The Journal of Systems & Software 203 (2023) 111734

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

GitHub Copilot AI pair programmer: Asset or Liability?☆

Arghavan Moradi Dakhel [a,*,1], Vahid Majdinasab [a,*,1], Amin Nikanjam [a], Foutse Khomh [a], Michel C. Desmarais [a], Zhen Ming (Jack) Jiang [b]

[a] Polytechnique Montreal, Montreal, Canada
[b] York University, Toronto, Canada

ARTICLE INFO

Article history:
Received 27 June 2022
Received in revised form 18 April 2023
Accepted 26 April 2023
Available online 2 May 2023

Dataset link: https://github.com/Copilot-Eval-Replication-Package/CopilotEvaluation

Keywords:
Code completion
Language model
GitHub copilot
Testing

ABSTRACT

Automatic program synthesis is a long-lasting dream in software engineering. Recently, a promising Deep Learning (DL) based solution, called Copilot, has been proposed by OpenAI and Microsoft as an industrial product. Although some studies evaluate the correctness of Copilot solutions and report its issues, more empirical evaluations are necessary to understand how developers can benefit from it effectively. In this paper, we study the capabilities of Copilot in two different programming tasks: (i) generating (and reproducing) correct and efficient solutions for fundamental algorithmic problems, and (ii) comparing Copilot's proposed solutions with those of human programmers on a set of programming tasks. For the former, we assess the performance and functionality of Copilot in solving selected fundamental problems in computer science, like sorting and implementing data structures. In the latter, a dataset of programming problems with human-provided solutions is used. The results show that Copilot is capable of providing solutions for almost all fundamental algorithmic problems, however, some solutions are buggy and non-reproducible. Moreover, Copilot has some difficulties in combining multiple methods to generate a solution. Comparing Copilot to humans, our results show that the correct ratio of humans' solutions is greater than Copilot's suggestions, while the buggy solutions generated by Copilot require less effort to be repaired. Based on our findings, if Copilot is used by expert developers in software projects, it can become an asset since its suggestions could be comparable to humans' contributions in terms of quality. However, Copilot can become a liability if it is used by novice developers who may fail to filter its buggy or non-optimal solutions due to a lack of expertise.

1. Introduction

Recent breakthroughs in Deep Learning (DL), in particular the Transformer architecture, have revived the Software Engineering (SE) decades-long dream of automating code generation that can speed up programming activities. Program generation aims to deliver a program that meets a user's intentions in the form of input–output examples, natural language descriptions, or partial programs (Alur et al., 2013; Manna and Waldinger, 1980; Gulwani, 2010).

Program synthesis is useful for different purposes such as teaching, programmer assistance, or the discovery of new algorithmic solutions for a problem (Gulwani, 2010). One finds different approaches to automatic code generation in the literature, from natural language programming (Mihalcea et al., 2006) and formal models (Drechsler et al., 2012; Harris and Harris, 2016) to Evolutionary Algorithms (Sobania et al., 2021b) and machine-learned translation (Rahit et al., 2019).

Novel Large Language Models (LLMs) with the transformer architecture recently achieved good performance in automatic program synthesis (Brown et al., 2020; Chen et al., 2021; Clement et al., 2020; Feng et al., 2020). One such model is Codex (Chen et al., 2021); a GPT-3 (Brown et al., 2020) based language model with up to 12 billion parameters which has been pre-trained on 159 GB of code samples from 54 million GitHub repositories. Codex shows a good performance in solving a set of hand-written programming problems (i.e., not in the training dataset) using Python, named HumanEval dataset (Chen et al., 2021). This dataset includes simple programming problems with test cases to assess the functional correctness of code. A production version of Codex is available as an extension on the Visual Studio Code development environment, named GitHub Copilot.[2] Copilot, as

☆ Editor: Prof. Raffaela Mirandola.

* Corresponding authors.
E-mail addresses: arghavan.moradi-dakhel@polymtl.ca (A. Moradi Dakhel), vahid.majdinasab@polymtl.ca (V. Majdinasab), amin.nikanjam@polymtl.ca (A. Nikanjam), foutse.khomh@polymtl.ca (F. Khomh), michel.desmarais@polymtl.ca (M.C. Desmarais), zmjiang@cse.yorku.ca (Z.M. Jiang).

[1] Both authors contributed equally to this research.

[2] https://copilot.github.com/

https://doi.org/10.1016/j.jss.2023.111734
0164-1212/

an "AI pair programmer", can generate code in different programming languages when provided with some context (called prompt), such as comments, methods names, or surrounding code.

Several studies focus on the correctness of code suggested by Copilot on the different types of problems such as linear algebra problems for an MIT course (Drori and Verma, 2021) or university level probability and statistical problems (Tang et al., 2021). The author in Finnie-Ansley et al. (2022) used Davinci (an API on a beta version of Codex) on different programming questions of a programming course and compared students' grades with the grade of the tool in solving the programming questions correctly. There are few studies that assess other aspects of Copilot besides the correctness of its suggestions. Nguyen and Nadi (2022) compared the complexity of Copilot's solutions in different programming languages for several LeetCode questions, besides their correctness. Authors in Vaithilingam et al. (2022) conducted a user study to understand how Copilot can help programmers complete a task. They studied how much time participants needed to complete a task using Copilot.

While these studies highlight some qualifications of Copilot, they neither examined the quality of the code produced by Copilot compared to humans nor did they investigate the buggy solutions suggested by Copilot and the diversity of its suggestions. Therefore, despite all these previous studies, we still do not know if–how Copilot, as an industrial component, can be leveraged by developers efficiently. We need to go beyond evaluating the correctness of Copilot's suggestions and examine how despite its limitations, it can be used as an effective pair programming tool.

The focus of our study is not on the type or difficulty level of programming tasks that Copilot can handle, but it is on the quality of the code that it will add to software projects if it is used as an AI pair programmer. We aim to investigate if the quality of code generated by Copilot is competitive with humans and if it can be used instead of a developer in pair programming tasks of software projects without impacting code quality. We highlight Copilot's limitations and competence with two different strategies and compared its suggestions with humans in different aspects. We also formulate suggestions on how developers can benefit from using Copilot in real software projects.

First, we assess Copilot's capabilities in solving fundamental algorithmic problems (i.e., searching and sorting) in programming. We study the correctness and reproducibility of Copilot's solutions to these problems. Secondly, we compare Copilot's solutions with human solutions in solving programming tasks, to assess the extent to which it can mimic the work of a human pair programmer. We use a dataset of different programming tasks containing up to 4000 in human-provided solutions (correct and buggy).

To conduct our study, we have chosen datasets for which Copilot is able to generate answers to their programming tasks. While such tasks may not be representative of all programming tasks that a professional developer performs, they allow us to assess Copilot's capabilities/limitations, and to list our suggestions to developers on how to benefit from this tool in real software projects. However, we acknowledge the limitations of generalizing the results to more complex tasks.

The results of our study show that Copilot is capable of providing efficient solutions for the majority of fundamental problems, however, some solutions are buggy or non-reproducible. We also observed that Copilot has some difficulties in combining multiple methods to generate a solution. Compared to human programmers, Copilot's solutions to programming tasks have a lower correct ratio and diversity. While the buggy code generated by Copilot can be repaired easily, the results highlight the limitation of Copilot in understanding some details in the context of the problems, which are easily understandable by humans.

Our finding shows Copilot can compete with humans in coding and even though it can become an asset in software projects if used by experts, it can also become a liability if it is used by novices, those who may not be familiar with the problem context and correct coding methods. Copilot suggests solutions that might be buggy and difficult to understand, which may be accepted as correct solutions by novices. Adding such buggy and complex code into software projects can highly impact their quality.

To summarize, this paper makes the following contributions:

- We present an empirical study on the performance and functionality of Copilot's suggestions for fundamental algorithmic problems.
- We empirically compare Copilot's solutions with human solutions on a dataset of Python programming problems.
- We make the dataset used and the detailed results obtained in this study publicly available online (Moradi et al., 2022) for other researchers and–or practitioners to replicate our results or build on our work.

The rest of this paper is organized as follows. We briefly review the related works in Section 2. Section 3 presents the design of our study to evaluate Copilot as an assistant to developers. We report our experiments to assess Copilot's suggestions for fundamental algorithmic problems and compare generated suggestions with what programmers do on specific programming tasks in Section 4. We discuss our results and potential limitations in Section 5. Threats to validity are reviewed in Section 6. Finally, we conclude the paper in Section 7.

2. Related works

A few studies empirically investigate the different capabilities of Copilot. Sobania et al. (2021a) compared Copilot with a Genetic Programming (GP) based approach that achieved good performance in program synthesis. Their findings show that GP-based approaches need more time to generate a solution. Moreover, training GP-based models is expensive due to the high cost of data labeling. Also, these approaches are not suitable to support developers in practice as GP usually generates code that is bloated and difficult to understand by humans (Sobania et al., 2021a).

Vaithilingam et al. (2022) conducted a human study involving 24 participant to understand how Copilot can help programmers to complete a task. They focused on 3 Python programming tasks: "1. edit CSV, 2. web scraping" and "3. graph plotting". Their finding shows that while Copilot did not necessarily improve the task completion time and success rate, programmers prefer to use Copilot for their daily tasks because it suggests good starting points to address the task. The tasks in this study involve less problem solving effort compared to the typical programming tasks in our study. They are mostly related to using programming language libraries. Also, they did not compare Copilot's suggestions with their participants' suggestions when working without the help of Copilot.

Drori and Verma (2021) studied Copilot's capability in solving linear algebra problems for the MIT linear algebra course. In the same line of work, Tang et al. examined Copilot's capability in solving university level probability and statistical problems (Tang et al., 2021). These two studies only focused on the correctness ratio of Copilot's solutions and did not examine its performance on programming tasks.

Finnie-Ansley et al. (2022) used Davinci (an API on a beta version of Codex) on two datasets. The first dataset includes 23 programming questions for a programming course, students' solutions for these questions, and their grades. This dataset is not publicly available. The second dataset is a set of different descriptions of a single well-known problem, rainfall, without

humans' solutions. For the programming questions, the paper focused on the grading of the solutions suggested by Codex: generating the correct solution for the problems after different runs (10 runs) and then comparing the grading with students. For the second dataset, besides the code correctness, they checked the variety of solutions by calculating the number of source lines of code (SLOC). Their results showed that Codex outperformed most students as evidenced by the grades received for their proposed solutions. Also, they observed that using the same input as a prompt on Codex can lead to different solutions, while Codex can generate correct solutions for different descriptions of the same problem.

Nguyen and Nadi (2022) evaluated Copilot on 33 LeetCode questions in 4 different programming languages. They used the LeetCode platform to test the correctness of Copilot's solutions. The questions in their study included different levels of difficulty. Although they evaluated the correctness of Copilot's solutions and compared their understandability, they did not assess whether Copilot successfully found the optimal solution for each task.

Another group of studies focuses on vulnerability issues to evaluate Copilot solutions. As mentioned before, Copilot is trained on a large volume of publicly available code repositories on GitHub which may contain bug or vulnerability problems. Pearce et al. (2022) conducted different scenarios on high-risk cybersecurity problems and investigated if Copilot learns from buggy code to generate insecure code. Another study investigates how Copilot can reproduce vulnerabilities in human programs (Asare et al., 2022). To do so, they first used a dataset of vulnerabilities generated by humans, then rebuilt the whole code before the bug and asked Copilot to complete the code. The completed section was manually inspected by three coders to determine if Copilot reproduced the bug or fixed it.

Moroz et al. (2022) examined the challenges and the potential of Copilot to improve the productivity of developers. They highlighted the copyright problems and the safety issues of its solutions. They discussed the non-deterministic nature of such models and the harmful content that could be generated by them.

Authors in Ziegler et al. (2022) surveyed 2631 developers about the impact of Copilot on their productivity and highlighted different metrics of users' interaction with Copilot that help predict their productivity. They relied on the SPACE (Forsgren et al., 2021) framework to generate 11 Likert-style questions in their survey. Also, they analyzed the usage data of Copilot of the participants who responded to this survey. They extracted different metrics from this data such as the acceptance rate of solutions, persistence rate, unchanged and mostly unchanged accepted solutions, etc. They found that the acceptance rate of solutions by developers is the most relevant metric that shows the impact of Copilot on the productivity of developers.

In another recent paper, the author discussed the opportunities and challenges of AI code generation tools in an educational context (Becker et al., 2022). This study is not an empirical study and there is no experiment or evaluation of Copilot's suggestions. The author was not specifically focusing on Copilot and its competence or limitations, but on the opportunities/challenges that such tools (code generation tools) bring into education such as exercise generation, illustrative samples, example solutions, etc.

The authors in Denny et al. (2023) evaluated Copilot's capabilities in solving elementary coding problems that are taught to students in an introductory programming course (CS1: Introduction to Programming) and investigated the effect of modifying the task descriptions, also known as prompt engineering, to address the unsuccessful attempts at solving the tasks using the original descriptions. They also categorized the area of failure on those programming tasks. The authors explored the impact of using different natural language descriptions in decreasing the number of failures or wrong suggestions. Their focus was only on the correctness (correct/fail) of Copilot's suggestions, not their quality if used as an AI pair programmer. They did not conduct a comparison between Copilot's suggestions and those generated by students.

In Wermelinger (2023), the author studied Copilot's capabilities on a handful of programming tasks designed to test students' knowledge of program design. The author also compared Copilot's outputs to those of Codex (Davinci, which is a model that Copilot is based on) and assessed the correctness of their suggestions alongside the diversity of their solutions. For diversity comparison, the author manually checked the different programming structures used in those solutions. The author also assessed Copilot's and Codex's abilities in generating test cases alongside explaining their own solutions. Compared to this research, we study Copilot which is based on Codex and has a more diverse set of algorithmic problems. We also analyze the quality of the generated solutions on multiple aspects such as optimality, reproducibility, and similarity compared to humans.

Authors in Leinonen et al. (2023) studied how Codex can be used to provide better programming error messages. The authors provided a faulty program alongside the prompt asking the model to explain why the program fails and to provide solutions. They analyzed the provided solutions in terms of whether the solution is correct, understandable, contains a fix to the fault, and whether it improves upon the original script.

To the best of our knowledge, none of these studies compared the quality of code generated by Copilot with human code in solving programming tasks. The majority of these studies focused on assessing the correctness of Copilot's solutions and highlighted its issues; e.g., the presence of vulnerabilities in generated solutions. In this study, we focus on fundamental algorithmic problems and compare the quality of code generated by Copilot with humans in solving programming tasks.

3. Study design

In this section, we present our methods to assess Copilot as an AI pair programmer and detail the experimental design to achieve our goals.

To solve a programming task, a developer usually builds the code on top of fundamental data structures (e.g., queues, trees, graphs) and algorithms (e.g., search, sorting) in computer science. Moreover, the developer needs to come up with ideas to achieve the goal(s) of the programming task efficiently.

We evaluate Copilot on (1) the adequacy of recommended code for fundamental algorithmic problems, and (2) the quality of recommendations compared to human-provided solutions. Specifically, we address the following research questions (RQs):

RQ1: Can Copilot suggest correct and efficient solutions for fundamental algorithmic problems?

RQ2: Are Copilot's solutions competitive with human solutions for solving programming problems?

In the rest of this section, we describe the research methods we followed to answer each of our RQs as illustrated in Fig. 1.

3.1. RQ1: Copilot on algorithm design

Our goal in RQ1 is to observe if Copilot can generate solutions for fundamental algorithmic problems given clear descriptions of the problem and do further analysis. Solving these fundamental algorithmic problems is important for developers contributing to a software project. Although these problems are not necessarily representative of all professional projects, the ability to correctly and efficiently handle fundamental programming problems is a

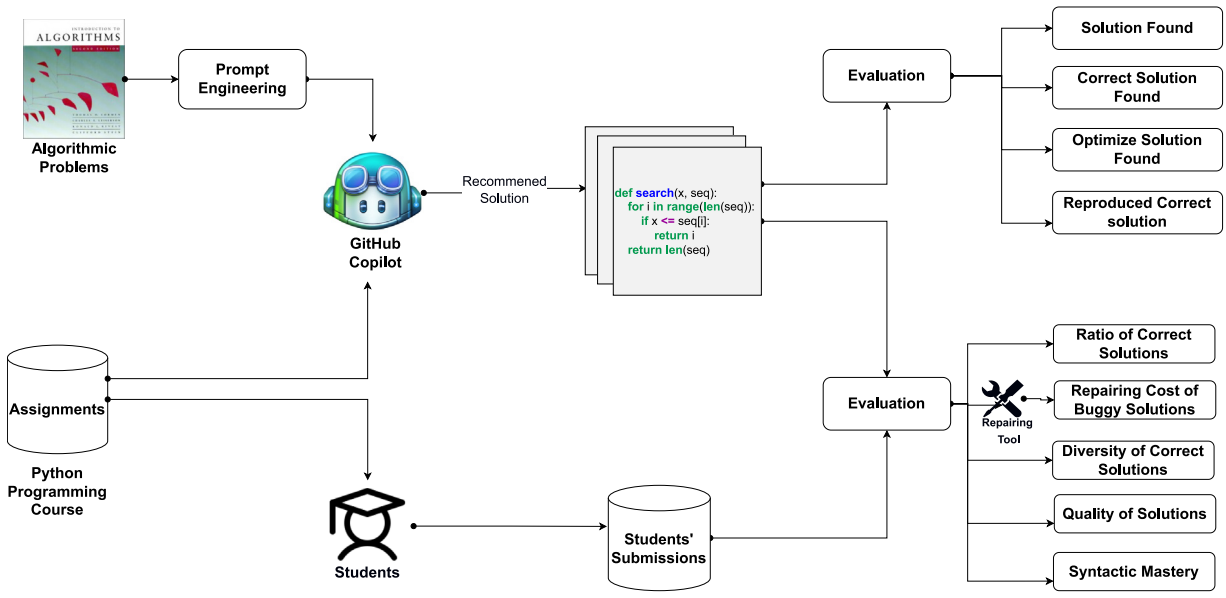


Fig. 1. The Workflow of proposed methods. The study includes two different methods to test Copilot in recommending code to solve programming problems. The first pipeline focuses on algorithmic problems collected from a well-known algorithm design book (Cormen et al., 2022a). The second pipeline focuses on the assignments of a Python programming course (Hu et al., 2019). It compares Copilot with students in solving programming problems in different aspects.

requirement for correctly and efficiently handling more complex programming problems. For example, Leetcode[3] is a website that represents different categories of such problems for the coding assessment which are often part of the interview questions for recruiting professional developers. Developers may not use these algorithmic problems outside of the assessment (i.e., in their daily tasks) directly, but understanding how to solve them with an optimal algorithm is essential. In this section, we plan to examine if Copilot is capable of solving these fundamental problems with optimal solutions. In this section, we plan to examine if Copilot is capable of solving these fundamental problems with optimal solutions.

3.1.1. Data set: Fundamental algorithmic problems

We selected the problems and their descriptions from Cormen et al. (2022a). We choose this resource because it is widely used for teaching algorithmic design fundamentals to computer science students (Cormen et al., 2022b). In this book, the authors explain the principal algorithms that computer engineers must be knowledgeable about by breaking them into categories. Since our problems were selected from this book, we followed its categorization, such that our tests on Copilot were conducted on 4 categories:

- **Sorting Algorithms:** Sorting algorithms are among the first algorithmic concepts that are taught to computer science students. These algorithms introduce the concept of time complexity and how inefficient code can make differences in more complex programs. Sorting algorithms are used in databases to segment large amounts of data that cannot be loaded entirely into memory or in numerical operations to determine which numbers should undergo operations first for the results to be produced as quickly as possible. From this section, we selected some well-known sorting algorithms which students are asked to learn and then implement. These algorithms are methods for sorting an array of numbers (integers or floats) in a descending or ascending manner. In these problems, time complexity, a measure of an algorithm's run-time as a function of input length, is an important factor to be considered.
  From the algorithms described in the book, we selected *bubble sort*, *bucket sort*, *heap sort*, *insertion sort*, *merge sort*, *quick sort*, *radix sort*, and *selection sort*. We selected these algorithms based on their implementation complexity, from easy to hard, based on Cormen et al.'s (2022a) descriptions.
- **Data Structures.** From this section, we selected the Binary Search Tree (BST). BST is a basic data structure that is taught in algorithm design. Each node of the tree contains a value (called *key*) that is greater than all the values in the left sub-tree and smaller than all the values in its right sub-tree. The implementation of BST involves multiple steps, namely:
  - Finding the minimum and maximum values in the tree before inserting any new value.
  - In-order tree walks to extract all the values in the tree in a sorted manner.
  - Finding the successor node. Given a node x, the successor of x is the node that has the smallest value which is greater than x.
- **Graph Algorithms.** From this section, we selected the *Elementary Graph Algorithms*. These algorithms are used to perform some elementary operations on a graph. Since graphs store information about how each node is connected to others, they can be used in implementing applications such as maps and social media user connections. We tested Copilot on the following graph algorithms problems:
  - Generating code for a simple graph data structure.
  - Breadth First Search (BFS) on a graph.
  - Depth First Search (DFS) on a graph.
  - Implementing Directed Acyclic Graphs (DAG). DAGs require a more complex implementation logic compared to simple graphs, since during initialization, based on the directions of the edges, we need to check if a cycle exists in the graph or not.
  - Finding reachable *vertices*. A pair of vertices are defined as reachable if both vertices can be reached from each other in a directed graph.

[3] https://leetcode.com/

- **Advanced Design and Analysis Techniques.** We selected the greedy algorithms from this section. Unlike the algorithms described above, the greedy technique is a general approach for solving optimization problems based on breaking problems down into multiple subproblems and selecting the best solution at the given time. As these solutions need to be evaluated in the context of a problem, we selected the "activity selection", an introductory problem to greedy algorithms as described in Cormen et al. (2022a).

3.1.2. Prompt engineering

Alongside code completion, Copilot can generate code from natural language descriptions in the form of comments. However, as noted by Li et al. (2022), if the description becomes too long or detailed, Copilot's performance degrades. Since the book that we are using to collect the problems (Cormen et al., 2022a) is a comprehensive educational book, each problem is described in detail and by building upon concepts that were explained in the previous chapters. As a result, some problem descriptions span multiple paragraphs and sometimes, pages.

However a summary description of our selected problems can be found in different resources, but the authors summarized the description of each problem in their own words to reduce the chance of memorization (Carlini et al., 2022) issue on Copilot. Therefore, our prompt engineering was done in two steps:

1. **Describing the problem:** We needed to summarize each problem's description to feed them to Copilot while staying as faithful as possible to the books. To make sure that our descriptions were understandable and did contain enough information about the algorithm being described, we cross-checked each of them with those on popular coding websites such as W3SCHOOLS (W3schools Team, 2022) and GEEKSFORGEEKS (Geeksforgeeks Team, 2022) as well. For cross-checking, the second author summarized Cormen et al.'s (2022a) algorithm descriptions while keeping in mind Copilot's limits on the length of the prompt. If there were differences in the descriptions (i.e., the description was missing some key elements in explaining the problem), the descriptions were revised.
2. **Cross validation of problem descriptions:** Cross-validation of problem descriptions: The second author created the input descriptions as explained above. After this, these descriptions were cross-checked with the first author to make sure that they were correct, understandable, and contained enough information about the problem being described. The first two authors both have taken the course "Introduction to Algorithms" during their education and have more than 5 years of experience in coding and program design. To assess the agreement, we have calculated Cohen's Kappa score (Cohen, 1960). While the score was 0.95 implying an **almost perfect agreement**, for cases where there were conflicts about the descriptions, the two authors met and discussed the conflicts to resolve them. In the end, the descriptions were also cross-checked with the third author who has more than 10 years of experience in teaching algorithm design. Therefore, the final input descriptions were what all three authors agreed on.

Excluding sorting algorithms, other problems require code to be generated using previous code as it is common practice in both functional and object-oriented programming For these problems, we followed exactly the book's example by asking Copilot to generate code for the underlying subproblems and then for the succeeding problems, we asked it to implement the solution using the code it had generated before. We have recorded all of our descriptions and Copilot's responses in our replication package (Moradi et al., 2022).

3.1.3. Solving fundamental algorithmic problems with Copilot

To generate solutions with Copilot, we feed the description of each algorithmic problem, call it prompt, to Copilot. At each attempt on Copilot with a selected prompt, it only returns up to the top 10 solutions. Thus, we do not have access to the rest of the potential suggestions. To inquire about Copilot's consistency in generating correct solutions, we repeat the process 6 times and each time collect its top 10 suggestions.

To assess whether Copilot's suggestions are consistent over time, we performed 2 trials within a 30 days time window. Each trial consists of 3 attempts for each prompt and each attempt contains up to 10 suggestions provided by Copilot. The collection of 3 first attempts is called "First Trial" and the collection of 3 last attempts which were conducted 30 days later is named "Second Trial".

Given that Copilot may try to consider the script's filename as a part of its query, to make sure that solutions were only generated from our descriptions, we gave the scripts unrelated names.

3.1.4. Evaluation criteria

Below, we briefly explain the 4 different metrics which we have used to evaluate Copilot and explain them in detail in the rest of this section. The metrics are calculated per each fundamental algorithmic problem.

1. **Response Received** $\in [0, 3] \in \mathbb{N}$. The number of attempts in each trial that Copilot was able to understand the problem and generate code content as its response.
2. **Correctness Ratio (%).** The percentage of correct solutions suggested by Copilot in each trial.
3. **Code Optimality** $\in [Yes, No]$. Whether at least one of the correct solutions suggested by Copilot in each trial has optimal time complexity [Yes or No].
4. **Code Reproducibility** $\in [Yes, No]$.
    - **Within a Trial:** Whether at least one of the correct solutions suggested by Copilot in one attempt was repeated in two other attempts, within a trial [Yes or No].
    - **Across Trials:** Whether at least one of the correct solutions suggested by Copilot in the first trial was repeated in the second trial [Yes or No].
5. **Code Similarity** $\in [0, 1] \in \mathbb{R}$.
    - **Within Trial:** The similarity degree between all correct solutions within a trial.
    - **Across Trials:** The similarity of correct solutions between two trials.

(1) Response received

Our observation shows that if Copilot is unable to provide solutions to the problem with the provided prompt, it will return irrelevant responses such as repeating user's prompts, code that only contains import statements or natural language responses. Thus, this metric helps us to evaluate if Copilot generates code for the summarized description of the program instead of the mentioned irrelevant responses.

We used the description of each problem in the form of comments and collected up to the top 10 suggestions of Copilot in 6 different attempts and two separate trials, as it is described in Section 3.1.3. To calculate this metric, if at least one of the suggested solutions in an attempt within a trial is code content, we consider it as a successful code generation attempt or "Response Received". Since we conduct 3 separate attempts in each trial, we report the value of this metric with a number $\in [0, 3] \in \mathbb{N}$ per trial.

(2) Correctness ratio

We report the correctness ratio as a fraction of solutions suggested by Copilot per problem that are functional and address the objective of the problem. To calculate this metric, we first need to evaluate the correctness of Copilot's suggestions for a problem. A suggested code is correct if it passed a set of unit tests.

However, in algorithmic problems, passing a set of unit tests to check the correctness of solutions is not enough. In this category, not only we need to verify a suggestion on passing a set of unit tests, but also we need to verify its chosen algorithm.

For example, in the "Sorting" problems, all problems have the same functionality: sorting a list of numbers. But the importance is the choice of the algorithm to address the problem and to check if Copilot is able to understand the structure of the solution from the given description. If Copilot implements the "Bubble sort" instead of the "Selection sort" algorithm or uses the Python built-in functions "sort" or "sorted", the code is still functionally correct and is able to sort the inputs. But the code is not addressing the algorithm described in the problem. That is the same situation for implementing the data structure of a BST or a graph.

We tackle this challenge of calculating the correctness ratio by following three steps:

1. We check the functional correctness of Copilot's suggestions on a set of unit tests.
2. We check if the selected algorithm in the solution follows the description that we gave to Copilot for that problem. To conduct this step, same as in Section 3.1.2, the two first authors separately checked the solutions suggested by Copilot for the problems. They compared the algorithm of the solutions (that is employed by Copilot to solve the problem) to the reference algorithms (ground truth). We collect the ground truth for each problem from the reference book (Cormen et al., 2022a) and from popular coding websites such as W3SCHOOLS (W3schools Team, 2022) and GEEKSFORGEEKS (Geeksforgeeks Team, 2022). We calculate Cohen's Kappa score to measure the agreement between the two authors.
3. The solutions per problem within a trial that passed the two above steps are labeled as correct. Then, we calculate the correctness ratio based on the fraction of the correct solutions within a trial.

(3) Code optimality

We report this metric because the problems in our dataset can be implemented with different algorithms. This choice of algorithm may impact their computation complexity for example using a nested loop, queue, or recursive functions to solve a problem. With this metric, we want to check if Copilot is able to suggest the optimal algorithm of a problem among its correct suggestions.

We cannot write a code to automatically check if the computation size of another code is optimal due to Turing's halting problem (Bera and Bera, 2020). Thus, same as in Section 3.1.2 and Correctness Ratio in this section, the two first authors check if there is a solution with an optimal algorithm between the correct solutions suggested by Copilot for a problem in a trial. They separately compared correct solutions with a reference optimal code for a problem (ground truth). If at least one of the correct solutions suggested by Copilot within a trial is optimal, they consider that Copilot is able to find an optimal solution for that problem [Yes] and otherwise [No]. We calculate Cohen's Kappa score to report the agreement of two authors on code optimality.

(4) Code reproducibility and similarity

While Copilot is closed-source and we have no information about its characteristics that may impact its behavior on our prompts, we want to study if this tool is able to reproduce a correct solution for a problem in different attempts and over time. We introduce "Code Reproducibility" as a metric for this measurement. For more clarification, we split our approach for measuring this metric into three subsets:

- We consider two different types for reproducing a code: the one that checks if a correct solution is reproduced across different attempts within a trial and calls it "Within a Trial", and the one that checks if a correct solution of a problem is reproduced over a time window among two trials and call it "Across Trials".
- To identify the correct solutions that are reproduced and measure their similarity, we have used the Abstract Syntax Trees (AST) similarity method described in Salazar Paredes et al. (2020). AST similarity is calculated by first building the AST of a code and then pruning the leaves that are related to variable or function names. Also, we ignore comments or any natural language text in each solution as they are not part of the code itself.
  AST similarity is bounded between 0 and 1 with 1 denoting structurally equivalent programs (regardless of their semantic similarity) and 0 denoting no equivalence between programs. It also returns 1 for "structurally equivalent recorded programs" where the programs are functionally identical but their instructions are executed in a different order, and "structurally equivalent renamed identical programs" where the programs are structurally the same with different variable names.
  Therefore, this similarity measure will not be affected by different statement orders or different variable names. However, this similarity will be different for semantically similar programs where the same concept is implemented in different ways. In Section 3.2.4, we explain in more detail why we need to apply this method to detect similar codes when we discuss Copilot's duplication solutions.
- To apply this comparison to correct solutions "Within a Trial", we compare the pairs of correct solutions across 3 different attempts within that trial. If at least one of the correct solutions in one attempt is reproduced in two other attempts (similarity equals 1), or in other words if at least one of the correct solutions within a trial occurs in all its 3 attempts, we consider that Copilot is able to reproduce the correct solution for that problem "Within a Trial" [Yes], otherwise, we consider that [No]. To apply it "Across Trials", we compare the pairs of correct solutions among two trials. If at least one of the correct solutions in the first trial is reproduced in the second trial (similarity equals 1), we consider that Copilot is able to reproduce the correct solution for that problem "Across Trials" [Yes], otherwise, we consider that [No].

Our observation shows that in some cases however Copilot's suggestions are not exactly the same but they are very similar. For example, Fig. 2 shows two code samples for "Insertion Sort". The differences between the two code samples are only syntactically in a few lines. Code Sample #1 calculates the length of the list within the range in for loop instructor. However, Code Sample #2 assigns the length of the list into a variable and then uses it to control the loop. Also, the comparison operator in the while loop condition is different in the two code samples. However, only the variables of the operator are switched and both are applying the same comparison.

Code Sample #1 for Insertion sort

```
def insertion_sort(array):
    for i in range(1, len(array)):
        key = array[i]
        j = i - 1
        while j >= 0 and array[j] > key:
            array[j + 1] = array[j]
            j -= 1
        array[j + 1] = key
    return array
```

Code Sample #2 for Insertion sort

```
def insertion_sort(arr):
    length = len(arr)
    for i in range(1, length):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

Fig. 2. Two different solutions suggested by Copilot for Insertion sort. There are a few lines in these two code samples that are syntactically different but both are addressing the same functionality. Code Sample #1 calculates the length of the list within the range in for loop instructor. Code Sample #2 assigns the length of the list to a variable and then uses it to control the loop. The comparison operator in the while loop condition is different in the two code samples. However, only the variables of the operator are switched and both are applying the same comparison.

Therefore, in addition to "Code Reproducibility", we report the "Code Similarity" as the average similarity between pairs of correct solutions for different fundamental algorithmic problems. To calculate the similarity, we follow the same AST similarity measure as explained above. For "Within a Trial", we compare all pairs of correct solutions in different attempts within a trial, and for "Across Trials", we compare all pairs of correct solutions between two trials. Finally, the average of these comparisons is reported for each problem.

3.2. RQ2: Copilot vs. Human

In this subsection, we aim to describe our research method for RQ2, on how to compare Copilot code with human written code in different quantitative metrics. First, we illustrate the dataset of programming tasks that we used in our experiments and explain why we select this dataset. Then, we explain how we employ Copilot to generate solutions for each task in this dataset. After that, we present how we selected students' solutions for this comparison. Finally, we discuss the criteria to compare Copilot with students in solving Python programming tasks from different aspects.

3.2.1. Dataset: Python programming tasks

To address RQ2, we choose a dataset of a Python programming course that includes students' submissions for 5 programming assignments (Hu et al., 2019). While the programming tasks in this dataset may not be representative of all the programming tasks that professional developers do, they provide us with an opportunity to assess the quality of Copilot's suggestions beyond code correctness. This dataset includes different students' submissions including buggy ones that support our RQ on investigating the bug-repairing cost of Copilot's suggestions. The bugs detected in students' code can also occur in professional development settings, as demonstrated by tools like FindBugs (Hovemeyer and Pugh, 2004). This tool, in the beginning, is designed to identify issues in student code, but it has also successfully detected bugs in production software systems (Hovemeyer et al., 2005). Additionally, the task descriptions in this dataset are human-written, reducing the chance of memorization issues (Carlini et al., 2022).

This dataset includes 2442 "Correct" and 1783 "Buggy" student submissions for 5 Python programming assignments in a Python course. Another study also used this dataset for characterizing the benefit of adaptive feedback for errors generated by novice developers (Ahmed et al., 2020). Table 1 shows the description of each programming task. Each task includes a description of the problem, one or more reference solutions, a different number of submissions by students that includes "Correct" and "Buggy" solutions, and different unit tests for each task, with an average of 9 tests per problem, to evaluate the functional correctness of solutions.

This dataset also contains a tool named "Refactory" to automatically repair the buggy submissions of students if applicable (Hu et al., 2019). In our study, we use this tool to repair buggy solutions generated by Copilot and students to evaluate the complexity of fixing bugs in code generated by Copilot compared to those of junior programmers. This tool matches each buggy program with the closest correct solution based on its AST structure. Then, it modifies different blocks of the incorrect program to repair its bug(s) and convert it to a correct solution if possible. This tool shows better performance than other state-of-the-art methods in repairing buggy programs such as Clara (Gulwani et al., 2018). Despite others that need a large and diverse range of correct solutions, this tool can repair buggy code even with one or two references (i.e., correct solutions).

Considering the choice of programming tasks and in order to have a fair comparison, we compare Copilot with junior developers. We acknowledge that the results of this comparison may not be generalizable to all developers. Still, they can provide valuable insights for future studies to conduct similar investigations on more advanced programming tasks and compare them to those written by more experienced developers.

3.2.2. Solving programming problems with Copilot

To generate solutions with Copilot, akin to Section 3.1.3, we feed the description of each programming task in Table 1, called prompt, to Copilot. At each attempt, Copilot only returns the Top-10 solutions for a prompt. Thus, we do not have access to the rest of the potential suggestions. To inquire about the Copilot's consistency in generating solutions, similar to the previous experiments, we repeat the process. In this setup, we repeat the process 5 times and each time collect its top 10 suggested solutions. Expressly, we ask Copilot to solve each programming problem in 5 different attempts and collect the top 10 suggested solutions in each one. Thus in total, we collect 50 solutions by Copilot for each problem.

As we already explained in Section 3.2.1, there are different test cases per task. To evaluate the functional correctness of Copilot's solutions, a solution is considered "'Correct" if it passes all the unit tests related to its problem. Otherwise, it is considered as "Buggy".

3.2.3. Downsampling student solutions

In each attempt on Copilot, we only have access to its top 10 suggestions while the average number of student submissions for these tasks is 689.8. One solution to have more suggestions by Copilot could be to increase the number of attempts on Copilot. But, increasing the number of attempts to more than 5 will increase the number of duplicate answers in Copilot's suggestions. We discuss the duplicate solutions in Section 3.2.4 with more details.

Thus, instead of increasing the number of attempts on Copilot, we downsample the students' submissions to the same size of Copilot solutions (50 in total) to have an equal number of solutions for students and Copilot.

Table 1
A summary of the dataset used to compare Copilot with the human in solving simple programming tasks. The Dataset includes the assignments and submissions of a Python programming course. It includes students' submissions for 5 Python programming tasks (Hu et al., 2019). The last two columns represent the number of students' submissions in two categories "Correct" and "Buggy".

	Task	Description	Correct	Buggy
q1	Sequential Search	Takes in a value "x" and a sorted sequence "seq", and returns the position that "x" should go to such that the sequence remains sorted. Otherwise, return the length of the sequence.	768	575
q2	Unique Dates Months	Given a month and a list of possible birthdays, returns True if there is only one possible birthday with that month and unique day, and False otherwise. Implement 3 different functions: unique_day, unique_month, and contains_unique_day.	291	435
q3	Duplicate Elimination	Write a function remove_extras(lst) that takes in a list and returns a new list with all repeated occurrences of any element removed.	546	308
q4	Sorting Tuples	We represent a person using a tuple (gender, age). Given a list of people, write a function sort_age that sorts the people and returns a list in an order such that the older people are at the front of the list. You may assume that no two members of the list of people are of the same age.	419	357
q5	Top_k Elements	Write a function top_k that accepts a list of integers as the input and returns the greatest k number of values as a list, with its elements sorted in descending order. You may use any sorting algorithm you wish, but you are not allowed to use sort and sorted.	418	108
Total			2442	1783

3.2.4. Evaluation criteria

For this part of our study, we consider different criteria to compare solutions suggested by Copilot and students to solve these programming tasks. We investigate the solutions on the following markers. In the rest of this section, we explain each metric in more detail.

1. Correctness Ratio (pass@Topk)
2. Repairing Costs
3. Diversity
4. Cyclomatic Complexity
5. Syntactic Mastery.

(1) Correctness ratio (pass@Topk)

A very common metric to evaluate programming language models is pass@k metric (Li et al., 2022; Chen et al., 2021). For example, calculating pass@100 shows the fraction of correct solutions out of 100 solutions. However, since Copilot returns only the Top 10 solutions in each attempt, we cannot accurately use this metric in our study.

In this study, what attracts our interest is the pass@Topk in all the attempts. It means that if we call Copilot n times for the same problem (the same prompt), n equals the number of attempts, and collect the Topk solutions of each attempt, then pass@Topk equals the fraction of these solutions that passed all the test units. As an example for pass@Top2, we collect all the Top2 suggested solutions for a problem in $n = 5$ different attempts ($\#solutions = k * n = 2 * 5 = 10$). Then pass@Top2 reports the fraction of these 10 solutions that passed all test units. We can calculate pass@K for Copilot but we cannot calculate it for students.

Another evaluation that comes to our attention is the Correctness Ratio (CR) of solutions. We calculate the correctness ratio of solutions the same as Section 3.1.4. Here by CR, we mean the fraction of correct solutions out of all solutions suggested by the Copilot or human for each problem. We calculate this fraction for each problem while collecting Topk suggestions of Copilot in different attempts. For students, we calculate the fraction of correct submissions out of all students' submissions for each problem.

Also, we calculate the distribution of the CR and its average in independent attempts on Copilot. We like to study how increasing the number of attempts (giving different chances to Copilot to solve the same problem) impacts the CR.

(2) Repairing costs

After computing the CR for Copilot and students, we aim to compare Copilot's buggy solutions with students' buggy submissions. Our observation shows that several buggy solutions generated by Copilot can be easily converted into a correct solution by applying small changes. We discuss this observation in detail in Section 4.2.2.

Repairing the cost of bugs in a software project is an important metric to show the quality of a code snippet (Kim and Whitehead, 2006). The long repairing time of a bug can be correlated with structural problems in a code snippet (Kim and Whitehead, 2006). One of the important factors that impact the repairing time of a bug is the code churn or the size of changes that are required to fix the bug (Zhang et al., 2012). Complex or low-quality code (in case of being buggy) need more time to be repaired, e.g., the developer needs to spend more time to detect the bug, or bigger patches are required for fixing the bug. Thus, to compare the quality of code generated by Copilot with students, we repair the buggy solutions and then compare them in terms of repair costs.

We use the repairing tool that we explained in Section 3.2.1. We choose an automated tool for repairing buggy code because:

1. Using an automated tool to fix bugs is very common in software projects. Software projects train their own tool for automatically fixing the bugs within the projects to save developers time (Arcuri, 2008).
2. By using an automated tool, we prevent our repairing process from being biased by one specific human expertise.

This tool reports three different metrics to evaluate the repairing cost of buggy solutions (Hu et al., 2019) including:

- **Repair Rate:** This metric shows the fraction of buggy code that passed all test cases after the repair process.
- **Avg. Repair Time:** This metric shows the average time taken to repair a buggy program in seconds.
- **Relative Patch Size (RPS):** This metric defines as the Tree-Edit-Distance (TED) between the AST of a buggy code and the AST of its repaired code, normalized by the AST size of the buggy code.

(3) Diversity

It is already shown in language models such as Codex that increasing the number of sample solutions for a programming

```python
def remove_extras(lst):
    new_lst = []
    for i in lst:
        if i not in new_lst:
            new_lst.append(i)
    return new_lst
```

(a) Sample code 1

```python
def remove_extras(lst):
    new_lst = []
    for item in lst:
        if item not in new_lst:
            new_lst.append(item)
    return new_lst
```

(b) Sample code 2

```python
def remove_extras(lst):
    # Duplicate elimination
    new_lst = []
    for item in lst:
        if item not in new_lst:
            new_lst.append(item)
    return new_lst
```

(c) Sample code 3

Fig. 3. Three different solutions were generated by Copilot for the q3: Duplicate Elimination Task in one attempt. There is no difference between the approach of these 3 solutions in solving the task. The only difference between (a) and (b) is in variable names, "i" and "item". The difference between (c) and (b) is the additional comment in (c). The differences between (c) and (a) are in variable names and comments.

task can increase the number of correct solutions that pass all test units (Chen et al., 2021; Li et al., 2022). However, they did not study if this increment is due to the increasing diversity of solutions or if the new correct solutions are just a duplication of previous ones.

Copilot claims that it removes duplicate solutions among the Top 10 suggested solutions in a single attempt. However, our observations show the appearance of duplicate solutions in the Top 10 suggestions of a single attempt. Fig. 3 shows three different solutions generated by Copilot for task q3: Duplicate Elimination at a single attempt. As we can see, the structure of all three codes are the same. The only difference between Figs. 3(a) and 3(b) is in the variable name, "item" and "i". Also, the solution in Fig. 3(c) is the same as the solution in Fig. 3(a) alongside comments. Since Copilot compares the sequence of characters to eliminate duplicates, it considers these three solutions as three unique suggestions in the Top 10 solutions of a single attempt.

To remove duplicate solutions in each attempt, we use the method discussed in Section 3.1.4 for reproducibility evaluation. We investigate if increasing the number of attempts and consequently increasing the total number of solutions will increase the number of unique solutions. Also, we compare the diversity of solutions (correct and buggy) provided by Copilot and students. This metric compares Copilot's novelty in generating solutions to that of students in solving a programming task.

To remark on the duplicate solutions, as we discussed in Section 3.1.4, we compare the AST of two codes. We eliminate the leaves in AST which are related to variable or function names. Also, we ignore comments or any natural language text in each solution. Then, we calculate a similarity between the AST of every two solutions for a problem by the method introduced in Salazar Paredes et al. (2020). If the similarity between two ASTs is equal to 1, then they are assumed to be duplicates. We keep just one of the solutions. Any value less than 1 represents a difference between the functionality of the two solutions.

(4) Cyclomatic complexity

A programming language is comprised of a set of programming keywords and built-in functions, methods, and types. Developers may solve a simple programming task in different ways. They may choose different programming keywords and built-ins to solve the same problem. However, even though flexibility in completing a programming task is desired, it can impact the efficiency, readability, and even maintainability of code in some cases (Maruping et al., 2009; dos Santos and Gerosa, 2018). These differences can also reflect developers' mastery of the programming language. For example, Fig. 4 shows two different solutions to a simple programming task, q4: Sorting Tuples, from Table 1. Code Sample #1 has more diverse programming syntax keywords and built-in functions, but Code Sample #2 is easier to understand and more readable.

Cyclomatic Complexity (McCabe's Cyclomatic Complexity C.C.) is another code quality metric that evaluates the understandability of a code snippet. C.C. shows the number of independent paths in a code component, specifically, the number of decisions that can be made in a source code (Ebert et al., 2016; Sarwar et al., 2013). Measuring the understandability of code snippets allows us to estimate the required effort for adding new features to the code or modifying it (Scalabrino et al., 2019).

There are studies that apply C.C. to measure the readability and understandability of small code snippets (Fakhoury et al., 2019; Dantas and Maia, 2021; Nguyen and Nadi, 2022). When comparing solutions for a problem, a lower C.C. indicates a more readable and understandable code. For example, in Fig. 4, the C.C. of code samples #1 and #2 are 4.13 and 1, respectively. While code sample #1 represents two nested for-loops to sort the list, code sample #2 simply calls sort and uses a lambda to loop over the list. Such an approach is more Pythonic and also more understandable.

To evaluate if Copilot's suggestions are as understandable as humans', we calculate the C.C. of Copilot's solutions and compare them to the C.C. of humans' solutions for the same problems. Thus, we can assess whether Copilot can provide understandable code that is easy to change and maintain (lower C.C.) or not if used as a pair programmer in a software project. We use a Python package, RADON,[4] to calculate it. C.C. close or above 10 is interpreted as not a best practice code.

(5) Syntactic mastery

As we already discussed in Section 3.2.4/(4), different syntax patterns and built-in functions, methods, and types in solving the same problem can reflect the developers' mastery as novice developers may not be familiar with all possible programming keywords and features in a programming language. While diversity in syntax patterns of a solution to address a specific task shows familiarity with more programming keywords and built-ins, these diverse solutions may not necessarily be the best practice to solve a problem. One of the goals of pair programming in industrial projects is to transfer such experiences from experts to novice developers (Plonka et al., 2015; Lui and Chan, 2006; Fronza et al., 2009). So, as another evaluation criterion, we compare the diversity of programming keywords and Python's built-in functions of Copilot's code to those of humans.

For example, the code in Fig. 4 are different solutions to solve the same programming task. Code sample #1 has more diverse programming syntax keywords such as {'FunctionDef', 'List[None]', 'for', 'if', 'BoolOp', 'else', 'break', 'elif', 'return'} and more diverse built-ins such as {'append', 'range', 'insert'}. Code sample #2 includes programming syntax keywords such as {'FunctionDef', 'Lambda', 'NameConstant', 'Subscript[Num]'} and built-in method, 'sort', which are less diverse than the first sample but more advanced and a less complex solution (in terms of cyclomatic complexity).

We follow the instructions suggested by Moradi Dakhel et al. (2021) to collect programming syntax patterns. We convert each solution to its AST and then walk through the syntax tree to collect nodes as programming keywords.

[4] https://radon.readthedocs.io/en/latest

Code Sample #1 for q4

```
def sort_age(lst):
a = []
for i in lst:
    if a ==[]:
        a.append(i)
        continue
    else:
        for k in range(0,len(lst)):
            if i[1]>a[0][1]:
                a.insert(0,i)
                break
            elif a[-1][1]>i[1]:
                a.insert(len(lst),i)
                break
            elif i[1] > a[k+1][1] and i[1]<a[k][1]:
                a.insert(k+1,i)
                break
return a
```

Code Sample #2 for q4

```
def sort_age(lst):
lst.sort(key=lambda x: x[1], reverse=True)
return lst
```

Fig. 4. Two different solutions to solve q4: Sorting Tuples. Code Sample #1 has more diverse syntax patterns and Python built-in functions compared to Code Sample #2. But #2 is more readable and less complex (in terms of C.C.) in understanding because of using more advanced programming syntax and built-in methods. The C.C. of Code Sample #1 (written by a human) is 4.13 while it is 1 for Code Sample #2 (suggested by Copilot).

To collect built-in functions within a code, first, we need to distinguish the built-in function from other function calls since all types of calls in Python, from built-in to local or public library, are a subset of a node named "Call" in AST. To do so, we extract a list of Python built-ins.[5] Then, we collect the node's name of the node "Call" if its "class_name" was in the list of Python built-ins. We compare the diversity of the keywords and Python built-ins in Copilot's and humans' code to study their capabilities in using Python's keywords and built-ins.

4. Empirical results

In this section, we present the results we obtained to answer our RQs, one by one.

4.1. RQ1: Copilot on algorithm design

In this section, we assess the capability of Copilot to solve algorithmic problems. To highlight the difference between our two trials which have been conducted 30 days apart from each other, for each marker, we have indicated the results of the solutions "Within a Trial" separately from each other as "First Trial" and "Second Trial". For this part of our study, we discuss the different evaluation criteria per each category of problems since our finding shows there is a correlation between the difficulty of the categories and the results.

4.1.1. Sorting algorithms

In this section, we discuss our findings on Sorting Algorithms. For those evaluation metrics where the manual inspection of authors was required (Response Received, algorithm validation on Correctness Ratio, and code Optimality), the authors achieved 89% of the Kappa agreement. We discuss the details in the following of this section.

5 https://docs.python.org/3/library/functions.html

(1) Response received

Our results in Table 2 on sorting algorithms show that when the algorithm gets more difficult and requires more details in implementation, Copilot struggles to generate solutions. For example, on the first trial, for Heap and Radix sort, Copilot generates code in only one of the 3 attempts within the trial. However, in the second trial, Copilot showed improvement as it generated code in all three attempts. The situation is the opposite for Merge sort. In the first trial, Copilot generates code in all three attempts. But in the second trial, it is responsive in only two of the three attempts.

(2) Correctness ratio

Copilot shows various behavior in generating correct solutions for sorting algorithms. The difficulty of problems impacts its ability to generate a correct solution and to use the correct algorithm for the implementation. However, Copilot shows different behavior in two different trials. For example in the first trial for bubble and bucket sort which are two easy sorting algorithms, 100%, and 85.71% of Copilot's suggestions were correct respectively. However, in the second trial, it generates no correct solutions for these two sorting problems.

Since implementing heap sort requires implementing a max heap, and then writing a sorting function, this algorithm is harder to implement. In the first trial, Copilot generates no correct solution for this problem. However, during our second trial, 9.09% of its suggestions for this problem are correct. In the second trial for Radix sort, Copilot showed improvement in solving the problem as it generated code in all three attempts (Response Received) but none of the generated code was correct.

Copilot shows some particular behavior for some of the sorting algorithms. For example, during the second trial where we asked it to generate code for Bucket sort, some of the generated code was calling the Quick sort function for sorting the buckets even though Quick sort had not been implemented in the code.

For validating the algorithm choice in solutions that passed all unit tests, two authors disagreed on the result for selection sort. The input prompt was summarized from the descriptions collected from the algorithm design book (Cormen et al., 2022a). The given prompt for this algorithm was "Create a function that accepts a list as input. The function should create two lists named sorted and unsorted. The function sorts an array by repeatedly finding the minimum element (considering ascending order) from an unsorted list and putting it at the beginning of the sorted list. Finally, it should return the sorted array". Given this description, the second author only accepted solutions that followed this exact description, mainly those which created the two empty *sorted* and *unsorted* lists. Upon review, however, the first and third authors mentioned that some solutions followed the selection sort algorithm, without following the exact steps mentioned in the description. After discussions, these solutions were considered as correct as well.

(3) Code optimality

Our result on code optimality shows that if Copilot is able to generate correct solutions for a sorting algorithm, it generates an optimal solution for that problem, too. In the first trial, Copilot generates correct solutions for 7 out of 8 sorting algorithms and it is able to generate optimal solutions for these 7 problems in the same trial, too. In the second trial, Copilot generates correct solutions for 4 sorting algorithms and it is able to generate optimal solutions within its correct solutions for these 4 sorting problems as well.

Since we have no correct solutions for example for Bubble sort or Bucket sort in the second trial, code optimality is not applicable for these cases. Thus, we show their results with "-".

Table 2
Results of Copilot's code generation ability on fundamental algorithmic problems. "Response Received" shows the number of attempts in each trial that Copilot can generate code for the proposed prompt. It ranges in $[0, 3] \in N$. **"Correct Ratio"** shows the percentage of correct solutions in each trial. **"Optimal"** status is "Yes" if at least one of the correct solutions in each trail is optimal. If at least one correct solution in one of the three attempts (Within a Trial) repeats in two other attempts (at the same Trial), then **"Reproduced"** is "Yes". **"Across Trials"** for **"Reproduced"** metric is "Yes" if at least one of the correct solutions from the First trial repeats in Second trial . If the metrics are not applicable then it presents by "-". For example, in "Second trial " of "Bubble Sort", we receive no (0) response from Copilot. Consequently, "Correctness Ratio" and "Optimal" in 'Second trial " and "Reproduced" in "Second trial " and "Across Trials" assign "-" for this algorithm.

Algorithm	Response received [0,3]		Correctness ratio [%]		Optimal [Yes/No]		Reproduced [Yes/No]		
	First trial	Second trial	First trial	Second trial	First trial	Second trial	First trial	Second trial	Across Trials
Sorting Algorithms									
Bubble Sort	3	3	100	0	Yes	–	Yes	–	–
Bucket Sort	3	3	85.71	0	Yes	–	Yes	–	–
Heap Sort	1	3	0	9.09	–	Yes	–	No	–
Insertion Sort	3	3	100	100	Yes	Yes	Yes	Yes	Yes
Merge Sort	3	2	33.34	0	Yes	–	No	–	–
Quick Sort	3	3	16.67	16.67	Yes	Yes	No	No	No
Radix Sort	1	3	10	0	Yes	–	No	–	–
Selection Sort	3	3	14.28	13.34	Yes	Yes	No	No	No
Binary Search Trees									
Data Structure	3	1	61.9	35.71	Yes	Yes	No	No	Yes
Min and Max Values in Tree	3	3	71.42	66.67	Yes	Yes	No	Yes	No
In-order Tree Walk	3	3	94.12	16.67	Yes	Yes	Yes	No	Yes
Finding The Successor Node	3	3	100	100	No	Yes	No	Yes	Yes
Elementary Graph Algorithms									
Simple Data Structure	2	2	50	0	Yes	–	No	–	–
Breadth First Search	3	3	100	100	Yes	Yes	Yes	Yes	Yes
Depth First Search	3	3	75	0	Yes	–	No	–	–
Directed Acyclic Data Structure	2	3	86.37	0	Yes	–	No	–	–
Finding Reachable Vertices	3	3	60	100	Yes	No	Yes	Yes	No
Greedy Algorithms									
Activity Class	2	3	0	0	–	–	–	–	–
Comparing Activities	3	3	9.52	0	Yes	–	Yes	–	–
Adding Activities to a Set	3	3	13.33	16.67	Yes	No	No	No	Yes
Generate All in one Prompt	1	3	0	0	–	–	–	–	–

However, this result on sorting algorithms is due to the fact that for some of the sorting algorithms such as bubble sort or heap sort, there is no other possible implementation than the optimal one (quadratic or log-linear).

We also observed that Copilot generates Pythonic code or uses Python's language-specific features instead of re-implementing the desired functionality to some extent. For example, alongside using list comprehensions (which are faster in Python than iterating over the list in explicit for-loops), Copilot generated code that uses built-in functions. An example of this can be observed in Fig. 5, code generated for the Quick sort where after dividing the input array into left and right subarrays, instead of generating code for sorting the arrays, Copilot used Python's built-in sort function. For sorting problems where iterating over the entire input was required, instead of using while loops, Copilot generates code with either explicit "for" loops or list comprehensions. Doing so removes the risk of getting trapped in an infinite loop and in the case of using list comprehensions can make a real difference in the program's running time.

(4) Code reproducibility and similarity

As the last evaluation metric in Table 2, we report if at least one of the correct solutions suggested by Copilot is exactly reproduced (similarity equals 1) within a trial and across two trials. Our result shows that correct solutions are not exactly reproduced for the majority of sorting algorithms.

However, the correct solutions are not exactly reproduced within a trial or across two trials, our results in Table 3 on the similarity degree between pairs of correct solutions show that they are very similar in some cases. For example, for Quick Sort in the second trial, the correct solutions are not exactly reproduced but based on Table 3, there is a 0.99 similarity between its correct solutions. As another example, for selection sort in the first trial and across two trials, the correct solutions are not exactly reproduced but the similarity degree between them equals 0.61 and 0.63 respectively.

Same as code optimality, if Copilot was not able to generate the correct solution for a problem within a trial, then the code reproducibility metric and similarity degree are not applicable. Also, if Copilot generates correct solutions for the sorting problems just in one of two trials, then reproducibility and similarity across trials are not applicable. We use "-" for nonapplicable cases.

Summary of results. In summary, Copilot is relatively capable of providing solutions for sorting problems. It is responsive (Response Received) for 6 out of 8 sorting problems of the first trial and for 7 out of 8 problems of the second trial.

On Correctness Ratio, in the first trial, Copilot generates correct solutions for all sorting problems except Heap sort, and on average, 51.42% of its solutions within this trial is correct. However, in the second trial, it generates correct solutions for only 4 sorting problems (out of 8) and the average correctness ratio within this trial is 34.77%. In both trials, if Copilot generates the correct solution for a problem, at least one of those correct solutions is optimal.

Finally, the correct solutions suggested by Copilot are not exactly reproduced for the 4th, 3rd, and 2nd sorting problems in the first trial, second trial, and across two trials respectively,

Table 3
Similarity ratios of the AST of Copilot's correct suggestions on fundamental algorithmic problems. To calculate the similarity, we removed the duplicate correct solutions in each attempt (three attempts within a trial). The results show however some of the correct solutions are not exactly reproduced in different attempts within a trial or between two trials, but they are very similar. The similarity is blank, "-", if it cannot be calculated (i.e. no correct solution or only one correct solution).

Algorithm	First trial	Second trial	Across trials
Sorting Algorithms			
Bubble Sort	0.93	–	–
Bucket Sort	1	–	–
Heap Sort	–	–	–
Insertion Sort	0.99	1	0.99
Merge Sort	–	–	–
Quick Sort	–	–	0.99
Radix Sort	–	–	–
Selection Sort	0.61	–	0.63
Binary Search Trees			
Data Structure	0.51	0.46	0.53
Min and Max Values in Tree	–	1	0.83
In-order Tree Walk	1	–	1
Finding The Successor Node	0.33	0.99	0.55
Elementary Graph Algorithms			
Simple Data Structure	0.25	–	–
Breadth First Search	0.54	0.72	0.45
Depth First Search	0.73	–	–
Directed Acyclic Data Structure	0.63	–	–
Finding Reachable Vertices	0.79	1	0.076
Greedy Algorithms			
Activity Class	–	–	–
Comparing Activities	1	–	–
Adding Activities to a Set	0.09	0.11	0.17
Generate All in one Prompt	–	–	–

Code Sample #1 for Quick sort

```
def Quick_Sort(arr):
if len(arr) <= 1:
    return arr
else:
    pivot = arr[0]
    less = [i for i in arr[1:] if i <= pivot]
    greater = [i for i in arr[1:] if i > pivot]
    return Quick_Sort(less) + [pivot] \
           + Quick_Sort(greater)
```

Code Sample #2 for Quick sort

```
def Quick_Sort(arr):
left = []
right = []
for i in range(len(arr)):
    if i % 2 == 0:
        left.append(arr[i])
    else:
        right.append(arr[i])
left.sort()
right.sort()
return left + right
```

Fig. 5. Two different solutions suggested by Copilot for Quick sort. Code Sample #1 is a recursive function. It picked the first element as a pivot to partition the given array and employed the correct "Divide and Conquer" algorithm to implement Quick Sort. However, Code Sample #2 randomly divided the given array into partitions. It is buggy, and it is not deploying the sorting properly, but it uses the Python built-in function, "sort" to sort each partition.

but the similarity degree between some of those non-reproduced correct solutions is above 0.6. In some trials and for some of the sorting problems the code optimality, reproducibility, and similarity are not applicable due to the lack of comparable correct solutions.

4.1.2. Binary search trees

In this section, we discuss our findings on Binary Search Trees (BSTs). The Kappa agreement between the two authors on evaluation metrics that needed manual inspection is 100%. For this problem, we first asked Copilot to generate the BST data structure which should comprise a class with the parent, right, and left nodes alongside the node's value. After that, we asked Copilot to generate a method that handles insertion per the BST insertion algorithm for the class. Then, we asked Copilot to create a method for deleting a node. These operations require the BST to be rebuilt in order to conform to the BST property. We also asked Copilot to implement a search method for finding if a value is present in the BST. These 3 methods comprise the base BST data structure. In the next steps, we asked Copilot to generate functions for finding the maximum and minimum value in a tree, performing an in-order tree walk, and finding the successor node of a child node. We discuss the details of our results in the following section.

(1) Response received

Our results show that Copilot is capable of understanding the BST problems in both trials. Only in the second trial, Copilot struggles in suggesting code in 2 out of 3 attempts for generating the data structure of a BST.

(2) Correctness ratio

Our results in Table 2 show that Copilot has inconsistent behavior in generating correct solutions for some BST problems in two trials. For example, considering the "In-order Tree Walk", 94.12% of Copilot's suggestions are correct in the first trial, but in the second trial, it reduces to 16.67%. However, for the two problems, "Min and Max Values in Tree" and "Finding The Successor Node", the correctness ratio on both trials are very close to each other. For example, for 'Finding The Successor Node", 100% of Copilot's suggestions are correct in both trials.

(3) Code optimality

It should be noted that, in a majority of the cases, Copilot was able to generate code consistent with optimal time complexities as required for an efficient BST problem. In addition, Copilot was able to generate multiple different versions (with iterative and recursive programming techniques) for "Finding maximum and minimum values in the tree", "In-order tree walk", and "Finding successor nodes" problems. For the "In-order Tree Walk" problem, Copilot generated functions inside the main function responsible for executing the walk. These functions were duplicate functions of those generated for finding minimum and maximum values in the tree. This is bad programming practice as it overcomplicates the code. However, since these functions were not used by the original function at all, the generated code was still optimal. Copilot tends to generate recursive functions when the solution can be solved using such an approach. For example, for the "In-order Tree Walk" and "Finding maximum and minimum values in the tree" problems, the generated code are all recursive functions.

Thus, for all the BST problems in both trials, except for "Finding successor nodes" in the first trial, at least one of the correct solutions suggested by Copilot has optimal time complexity.

(4) Code reproducibility and similarity

As it is shown in Table 2, in the first trial, Copilot exactly reproduces at least one of its correct solutions in 3 different attempts only for the "In-order tree walk" problem. Based on Table 3, the similarity between the pairs of its correct solutions is not greater than 0.51 for those correct solutions that are not exactly reproduced. For example, the similarity of correct solutions in different attempts of the first trial for "Data Structure" and "Finding successor nodes" are 0.51 and 0.33 respectively.

In the second trial, based on Table 2, the exact correct solutions are reproduced for "Finding maximum and minimum values in the Tree" and "Finding successor nodes" problems. The similarity for correct solutions of 'Data Structure" which is not exactly reproduced in this trial is 0.46.

Unlike sorting algorithms, reproducibility across two trials was not an issue on BST problems as Copilot reproduces at least one of the correct solutions from the first trial in the second trials for all BST problems except "Finding maximum and minimum values in the Tree". However, Table 3 shows that the similarity of the correct solution for this problem across two trials is 0.83.

Summary of results. In summary, Copilot is capable of understanding the description of BST problems in both trials, except for the "Data Structure" problem on the second trial.

Copilot has inconsistent behavior in generating correct solutions in two trials as 81.86% of its solutions are correct in the first trial but the correctness ratio equals 54.76% in the second trial. Copilot was able to generate optimal code for all the BST problems in both trials except for "Finding successor nodes" in the first trial.

Copilot struggled in exactly reproducing its correct solutions within each trial and the similarity of those solutions that are not exactly reproduced is not above 0.51. However, Copilot reproduces at least one of its correct solutions from the first trial in the second trial (Across Trials) for all BST problems except "Finding maximum and minimum values in the Tree". Although the correct solutions for this problem are not exactly reproduced across two trials, the similarity of its correct solutions is 0.83.

4.1.3. Elementary graph algorithms

In this section, we discuss our findings on Elementary Graph Algorithms. The Kappa agreement between the two authors on metrics that needed manual inspection is 83%. As our algorithms are becoming more complex, it is required for Copilot to generate code that uses the previous code that it has generated. We discuss the details of our results in the following section.

(1) Response received

Our results in Table 2 show that like BSTs, Copilot is adept at generating code for elementary graph algorithms. In the first trial, Copilot generates code in all 3 attempts for all graph problems except "Simple Data Structure" and "Directed Acyclic Data Structure" and in the second trial, it struggles only in one of the 3 attempts on "Simple Data Structure".

(2) Correctness ratio

As we can find in Table 2, same as BST problems, Copilot shows inconsistent behavior in generating correct solutions for some graph problems in two trials. For example, for "Simple Data Structure", "Depth First Search" and "Directed Acyclic Data Structure" in the first trial, 50%, 75% and 86.37% of Copilot's Suggestions are correct respectively. However, in the second trial, Copilot is not able to generate correct solutions for these problems. For the "BFS" problem, 100% of Copilot solutions are correct in both trials.

Our observation shows that during different attempts on Copilot to generate code for BFS and DFS, Copilot generated code for both algorithms regardless of our asking to do so only for one of them.

Even though Copilot was able to recognize and generate code for our description, some of the generated code had one flaw and since successor methods use the previous methods, this bug was present in every piece of generated code. This snow-balling effect has affected our Kappa score as well. This bug was a result of Copilot considering the nodes being named by integer numbers. As a result, if a node is created with a name that is not an integer (e.g. "A" or "Node1" instead of "1" or "2"), the code will fail to iterate through the list of nodes and generate a syntax error. However, since the code functioned correctly given the normal usage, we labeled them as correct.

(3) Code optimality

In the first trial, Copilot generated one optimal solution for each of the graph problems. However, in the second trial, out of 2 problems that Copilot addressed correctly, only one of them, BFS, includes the optimal solution within its correct solutions. Checking if a graph is cyclic, requires using a BFS or DFS approach. If Copilot does not use the code that it has generated for BFS and DFS during checking if a graph is cyclic, we will be left with code pieces that repeat the same operation over and over which is a bad practice in programming. We consider those suggestions as non-optimal.

We examined the solutions suggested by Copilot for constructing the graph data structure and observed that its solutions contain both list comprehensions and explicit "for" loops. In one of the correct solutions, the generated code constructs the nodes from the input using explicit "for" loops, and in another solution, it does so using list comprehensions. We accept the code that uses list comprehensions as optimal since if the input is large, there is a real running time difference between these two approaches. We also observed that some of the generated code are using an advanced Python feature called "operator overloading" in which a native Python function is rewritten by the programmer to behave differently depending on the arguments that it receives as input. Fig. 6 shows an example of operator overloading generated by Copilot.

(4) Code reproducibility and similarity

As we can find in Table 2, in the first trial, Copilot is able to reproduce at least one of its correct solutions for only two graph problems, "Breadth First Search" and "Finding Reachable Vertices". However, for other problems such as "Depth First Search" and 'Directed Acyclic Data Structure", the correct solution is not exactly reproduced by Copilot but their similarity equals 0.73 and

```
def __contains__(self, key):
    return key in self.graph

def __str__(self):
    return str(self.graph)
```

Fig. 6. Code sample of operator overloading. "operator overloading" is an advanced Python feature in which a Python built-in function is re-written by the programmer to behave differently depending on the arguments that it receives as input. "contains" and "str" are two Python native functions that Copilot re-wrote in graph problems.

0.63 respectively. In the second trial, Copilot is able to reproduce the correct solutions for those two problems that it addressed correctly. For across trials, Copilot is able to exactly reproduce the correct solutions only for the BFS problem. The similarity between correct solutions of "Finding Reachable Vertices" is very low across two trials, 0.076.

Summary of results. Our results show Copilot is adept at generating code for elementary graph algorithms. However, same as BST, Copilot shows inconsistent behavior in generating correct solutions for some graph problems in two trials. In the first trial, Copilot is able to generate correct solutions for all graph problems with an average correct ratio of 74.27%. However, in the second trial, it is able to generate correct solutions for only two problems and 100% of its correct solutions are correct. Copilot was able to generate optimal code for all problems that it addressed correctly in both trials except for "Finding Reachable Vertices" in the second trial. In the manner of reproducibility, it struggled to reproduce its correct solutions for all graph problems. However, the similarity between correct solutions for some problems is more than 0.6.

4.1.4. Greedy algorithms

In this section, we discuss our findings on the "activity selection" problem as a Greedy Algorithm. The Kappa agreement between the two authors on metrics that needed manual inspection is 100%. The "activity selection" problem requires the programmer to define a class for "activities". Each activity has a start and end time. The goal of this problem is: given a set of activities where each activity has its own start and ending time, return a set that contains the maximum number of activities that can be performed as long as they do not overlap. Overlapping is defined as:

- An activity's start time must be after a previous activity's end time.
- An activity should not happen during another activity.

For this problem, we asked Copilot to generate code for implementing the activity class, comparing activities, and finally checking for overlaps between activities to investigate if the generated solutions are "greedy".

(1) Response received

Our results in Table 2 show that Copilot is capable of understanding what the underlying problem is and can generate code for it. Our observations show that Copilot can even generate code when we give it the entire problem definition (activity class, comparing activities, and adding activities to a set) in one go.

(2) Correctness ratio

Even though Copilot is capable of understanding what we ask from it, the code that it generates for solving the problem is either buggy or incorrect. For example, given the prompt "implement a class called activity. Each instance of this class has two attributes: start-time and end-time. Both should be integer numbers between 0 and 24", the generated code has no functionalities for checking the input type or their boundaries. In another problem, when we asked Copilot to implement a method for comparing activities, we gave it the following prompt: "implement a function for comparing two activities. the function should return True if the first activity ends before the second activity starts. if the inputs have overlapping start times, return False". Here, Copilot implemented the description correctly. However, since this method is dependent on its inputs being instances of the activity class, this code will fail if the input is anything else. Type checking is important and a basic operation to do which Copilot fails to do here. Finally, for adding activities to a set of activities, Copilot was asked to create a method that accepts a set of activities alongside a start time and end time of activity. The method should first create a new activity instance with the given start and end time and then check if this new activity does not overlap with the activities in the set. Copilot was unable to generate the necessary code for this no matter how detailed the description was.

(3) Code optimality

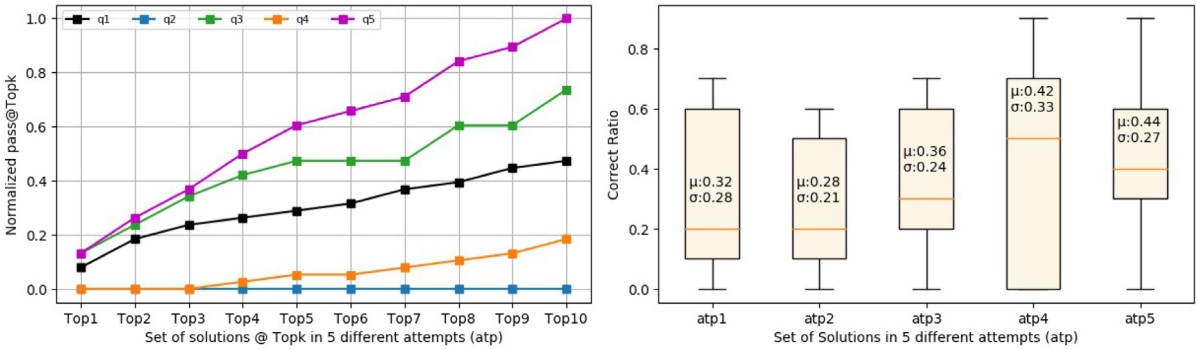
As Copilot was not able to generate correct solutions to most of the problems, we could only analyze the optimality of the solutions generated for "Comparing activities" and "Adding Activities to a Set". Here, the generated code was simple (As was the underlying problem) and the solutions required only checking the boundaries of class attributes or whether the output of a function was true or not.

(4) Code reproducibility and similarity

As Tables 2 and 3 show, Copilot was only capable of reproducing solutions to a problem for the "Adding activities to a set" problem across trials and these solutions were different from each other. As Table 3 shows, for the "Comparing Activities" problem, Copilot generated solutions that were exactly the same in the same trial. However, in the second trial, it was not capable of even producing a correct solution.

Summary of results. The activity selection problem was used as a proxy to see whether Copilot would be able to generate code for solving this problem with a greedy solution. However, Copilot was not able to generate solutions that satisfied the criteria of a correct solution. In particular, Copilot showed difficulties in understanding type checking and variable boundary checking even though such behaviors were explicitly required in the prompt.

Findings: Copilot is able to recognize fundamental algorithms by their names and generate correct, optimal code for them as long as the descriptions are short and concise. In some cases, the developers may need to invoke Copilot multiple times in order to receive solutions that are correct and tailored to their descriptions.
Challenges: Copilot is unable to generate code for type-checking variables. It also generates needlessly complicated code for some simple descriptions. Hence, Copilot still needs to be improved to truly be considered as a pair programmer.



(a) Normalized pass@Topk of 5 different attempts

(b) CR of solutions in 5 attempts

Fig. 7. Evaluation of correct solutions generated by Copilot. Plot (a) shows the normalized values for pass@Topk metrics against different values of k. It shows the fraction of correct solutions between Topk solutions of 5 different attempts. Plot (b) shows the distribution, average and standard deviation of the Correctness Ratio (CR) in each attempt for different programming tasks.

4.2. RQ2: Copilot vs. Human in solving programming problems

In this section, we discuss our findings to answer RQ2. We discuss the results for each criterion of our evaluation separately.

4.2.1. Correctness ratio of Copilot's suggestions and students' submissions

As explained in Section 3.2.4, we calculate the pass@Topk for solutions generated by Copilot for each programming task. The pass@Topk shows the fraction of correct solutions among the Topk solutions, collected from 5 different attempts. We normalized the values to report this metric for the programming tasks.

Fig. 7(a) shows the normalized values for pass@Topk of each programming task for Copilot. TopK solutions range between Top1 to Top10 because each attempt on Copilot includes only the Top10 suggestions. Based on this result, Copilot cannot find correct solutions for "q2: Unique Dates Months". This task asks for *"...solve the problem by implementing 3 different functions..."*. Copilot could not understand this point within the task description and tried to solve the problem in one function. Thus, all of Copilot's solutions for this task failed the test cases because the test units of this task are based on implementing 3 different functions.

There are no correct solutions in Copilot's Top3 suggestions for "q4: Sorting Tuples" in 5 different attempts. It increases to 0.02 in the set of Top4 solutions. For "q1", "q3", and "q5", the pass@Top1 is equal to 0.08, 0.13, and 0.13, respectively. For some questions, the pass@Topk, at different values of k, shows greater values than the other questions. For example, "q5" has the greatest values for pass@Top4 and above. Also, "q4" has the lowest pass@Topk, for different values of k, after "q2".

In general, pass@Topk increases by increasing the k. It means collecting a larger number of solutions suggested by Copilot increases the number of correct solutions and this growth can be different for different programming tasks.

In addition, Fig. 7(b) shows the Correctness Ratio (CR) of solutions in each attempt independently. However, the distribution of CRs in different attempts is varied, but adding new attempts can increase the average CR of solutions. For example, the average CR in the first attempt (atp1) is equal to 0.32 while it increases to 0.44 in the last attempts (atp5). It shows if we ask Copilot to solve the same problem multiple times (here 5 attempts), there is a chance to increase the CR among new Top10 suggested solutions on average. However, this is not correct for all questions. For example for "q1", the CR in "atp4" is 0.7 but it decreases to 0.4 in "atp5". But, for "q5", the CR in the first attempt is equal to 0.7 and it increases to 0.9 in the last attempt.

Since we cannot calculate pass@Topk for students, in Table 4, we compare the CR of solutions generated by Copilot with the CR of students' submissions. For this comparison, we calculate three different CRs for Copilot. The first, CR@Top1, reports the number of correct solutions out of all Top1 solutions in 5 different attempts for each programming task. CR@Top5 calculates the fraction of correct solutions out of all Top5 solutions suggested by Copilot in 5 different attempts. Finally, CR@Top10 represents the number of correct solutions generated by Copilot out of all its 50 solutions for a programming task. Collecting more solutions decreases the CR of Copilot since it increases the fraction of wrong solutions. For some of the questions, CR@Top1 and CR@Top5 of Copilot are greater than students' CR. For all questions, the CR of students' submissions is greater than CR@Top10 for Copilot's suggestions. On average for all the programming tasks, the Correctness Ratio (CR) of students' submissions is greater than the CR of Copilot's suggestions.

4.2.2. Repairing costs of buggy solutions generated by Copilot and students

In this part, we compare the repair cost of buggy solutions for Copilot with students. As we already discussed, our observation shows there are buggy solutions that are generated by Copilot and are very similar to correct solutions. A small change can convert them into a correct solution. Therefore, we attempt to quantify our observation by calculating the intersection between Copilot's correct and buggy solutions for each problem using the BLEU score (Papineni et al., 2002). The comparison has been done in a pairwise manner between each correct and each buggy solution. For example, if out of 50 solutions, 40 are correct and 10 are buggy, we end up with 400 pairwise comparisons.

BLEU is used in evaluating program synthesis approaches such as text-to-code, code summarization, and code prediction. BLEU score uses the n-gram overlap between tokens of two contents and penalizes length difference. It returns a value between 0 and 1 (Tran et al., 2019). BLEU measures how well two texts match or are similar to each other. Ren et al. (2020) introduces a new metric, called CodeBLEU, that measures the BLEU score on syntax and semantics of code. As a part of this new metric, they measure CodeBLEU between AST of code.

To measure the overlap between correct and buggy solutions, we measure the BLEU score between the AST of the buggy and correct. We omit the buggy code that has syntax errors and cannot be converted into AST. For example, the BLEU score of more than 0.7 between the AST of several correct and buggy pairs of solutions implies a high similarity between these two solutions. It can give us an estimation of the number of changes that we need to apply to a buggy solution to repair it.

Table 4
The Correctness Ratio (CR) of Copilot's solutions while collecting Top1, Top5, and Top10 solutions in all 5 attempts compared to the CR of students' submissions.

		Copilot			Students
	Task	CR@Top1	CR@Top5	CR@Top10	CR
q1	Sequential Search	**0.6**	0.44	0.36	0.57
q2	Unique Dates Months	0.00	0.00	0.00	**0.40**
q3	Duplicate Elimination	**1**	0.72	0.56	0.64
q4	Sorting Tuples	0.00	0.08	0.14	**0.54**
q5	Top-k Elements	**1**	0.92	0.76	0.79
	Total	0.52	0.43	0.35	**0.59**

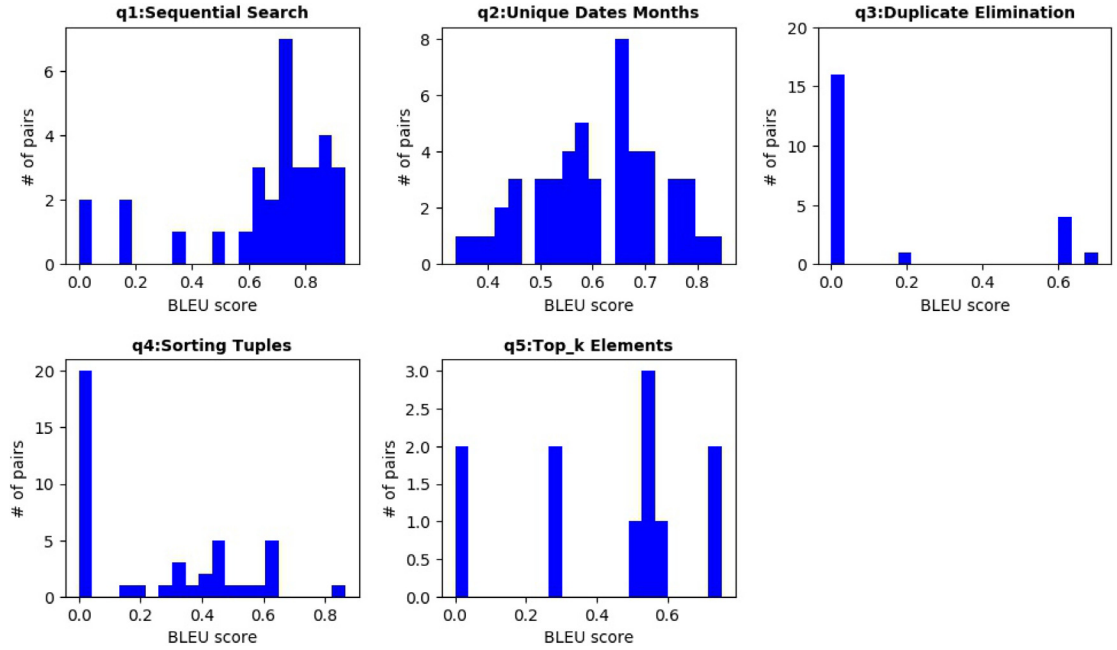


Fig. 8. Distribution of BLEU score among the pair of correct and buggy solutions generated by Copilot. This chart shows a histogram of the BLEU Score on pairs of correct and buggy solutions generated by Copilot. The BLEU score of 0.75 and above represents a great similarity between the AST of a correct and buggy pair. The BLEU score between several pairs of the buggy and correct solutions is greater than 0.7, in different programming tasks. This supports our observation that several buggy solutions can be corrected with small changes.

Fig. 8 shows the density distribution for the BLEU score among pairs of the buggy and correct solutions generated by Copilot for different programming tasks. As we can see in this figure, there are pairs of correct and buggy solutions with BLEU scores of 0.75 or greater. It shows that sometimes a small change in a buggy solution generated by Copilot can easily convert it into a correct solution, for example, changing "$>$" (greater) to "$\geq$" (greater equal).

Now that some of the buggy solutions generated by Copilot are very similar to the correct solutions, we are interested in comparing the repairing cost of Copilot's buggy solutions with students' buggy submissions. As we have explained in Section 3.2.3, for this comparison, we need to downsample students' submissions to the same size as Copilot's suggestions. Fig. 9 shows the distribution of repairing time for repairing students' buggy submissions. There are a high number of submissions with low repairing time and few with high repairing time. Thus, to keep the distribution of repairing costs in the sample set close to the entire population, we repeat the downsampling process 5 times and report all repairing metrics for students' submissions based on the average of all 5 sampleset.

As we can find in Table 5, the average repair rate for Copilot's buggy solutions is greater than students', which are 0.95 and 0.89 respectively. This means that on average, 95% of buggy solutions generated by Copilot have been fixed after the repair process. For example, for "q4: Sorting Tuples" and "q5: Top-k Elements", all buggy solutions of Copilot (100%) have been fixed while the repairing rate of students' submissions for these two tasks is equal to 85%.

In addition, the average repair time for Copilot's buggy solutions is less than the students'. This means that not only the repairing tool can fix the majority of Copilot's buggy solutions but also it can fix them faster than student buggy submissions. The average repairing time for Copilot's buggy solutions is 4.94 s while it is equal to 6.48 s for the students. The reason is that on average, the Relative Patch Size (RPS) of Copilot's buggy solutions that need to be repaired is smaller than students'. As we can find in Table 5, the average RPS for Copilot and students are 0.33 and 0.35, respectively.

We can conclude that however on average, the CR of students' submissions is greater than Copilot's solutions, but the repairing costs of buggy solutions of Copilot are less than students. With a repairing tool, we can repair the majority of buggy solutions generated by Copilot and increase its CR.

Thus, if Copilot, as a pair programmer in a software project, suggests buggy solutions, it is less expensive to fix its bugs compared to bugs that may be produced by junior developers when solving the same programming task.

4.2.3. Diversity of Copilot's suggestions and students' submissions

The diversity of solutions shows the novelty of Copilot and students in solving different problems. Also, it shows that while

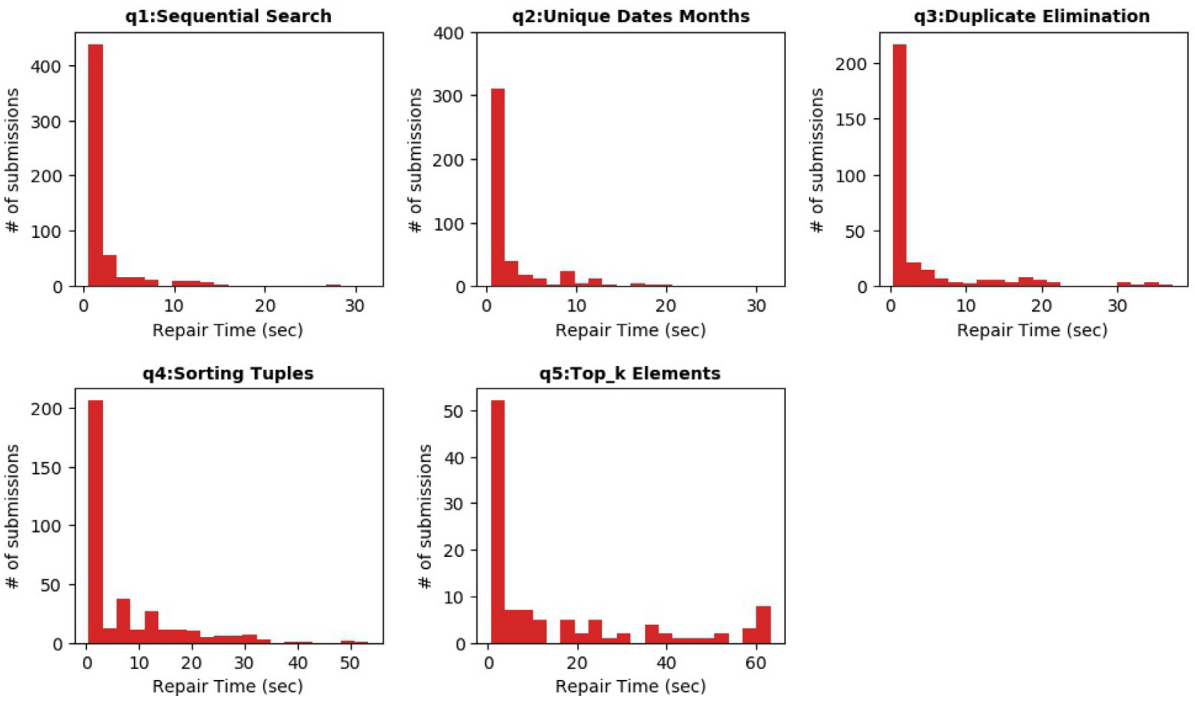


Fig. 9. The distribution of repairing time for students' buggy submissions. This chart shows a histogram of students' buggy submissions based on their repairing time. It shows that there are more buggy submissions with low repairing time than buggy submissions with high repairing time. We repeat the downsampling process on students' submissions 5 times to observe the same distribution in samplesets.

Table 5
Comparing the Repairing Cost of Copilot's suggestions with students's submissions.

		Copilot			Students		
	Task	Rep Rate	Avg Rep Time(sec)	Avg rps	Rep Rate	Avg Rep Time(sec)	Avg RPS
q1	sequential search	0.94	9.61	0.48	0.98	2.58	0.40
q2	unique dates months	0.92	3.26	0.28	0.82	3.81	0.44
q3	duplicate elimination	0.91	0.64	0.26	0.96	4.35	0.30
q4	sorting tuples	**1.00**	**0.78**	**0.15**	0.85	8.82	0.29
q5	top-k elements	**1.00**	10.40	0.50	0.85	12.84	0.30
	Total	**0.95**	**4.94**	**0.33**	0.89	6.48	0.35

increasing the number of sample codes increases the fraction of correct solutions, this increment is due to the diversity of correct solutions or duplication. As we discussed in Section 3.2.4, we observe duplicate solutions in a single attempt and across multiple attempts on Copilot to solve a problem. On the other hand, we observe duplicate solutions among students' submissions as well. For example, for "q1: Sequential Search", after comparing the ASTs of students' correct submissions, 54.32% of their submissions are identified to be duplicated.

To compare the diversity among students' submissions and Copilot's solutions, we randomly downsample 10 student submissions in 5 different sample sets and consider them as 5 different attempts. Then, in each attempt on Copilot and for each sample set of students' submissions, we eliminate duplicate correct and buggy solutions. There are a few buggy solutions for Copilot and for student solutions involving syntax errors that cannot be converted into AST (3 solutions). We consider them as non-duplicate buggy solutions.

Fig. 10 shows the cumulative distribution of Correct (C) solutions, None Duplicate Correct (NDC) solutions, Buggy (B) solutions, and None Duplicate Buggy (NDB) solutions by Copilot and students across different tasks. In this figure, for example, in "q3: atp3", the number of Correct (C) solutions suggested by Copilot is 17 but the number of Non-duplicate Correct (NDC) solutions is only 2. This means that after generating more solutions and running more attempts, Copilot repeats these 2 correct solutions several times. However, out of 14 Correct (C) solutions generated by students in the third attempt (atp3), 13 solutions are non-duplicate. That is the same observation for buggy solutions. Increasing the number of attempts on Copilot leads to a jump in the number of correct solutions for "'q1" and "q5" from 2 to 18 and 7 to 38 respectively. However, for "q3" and "q4", this growth is smaller. The number of None Duplicate Correct (NDC) solutions of Copilot is less than or equal to the number of Correct (C) solutions in each attempt for each task. This is the same story for Buggy solutions. However, it shows that despite Copilot's claims that it removes the duplicate solutions, there are still duplicates in the Top 10 solutions of each attempt.

The difference between C and NDC in student submissions is less than Copilot. For example, in "q3", the cumulative number of C solutions generated by Copilot in different attempts is greater than students' submissions in different samplesets. However, it is the opposite for NDC solutions. In "atp5" the cumulative number of C solutions generated by Copilot equals 28 and it equals 22 after the 5 sampleset on students' submissions. However, the cumulative NDC solutions at these attempts equal 2 (out of 28) for Copilot and it equals 21 (out of 22) for students. It shows more diversity between correct and even buggy submissions of students compare to Copilot's solutions.

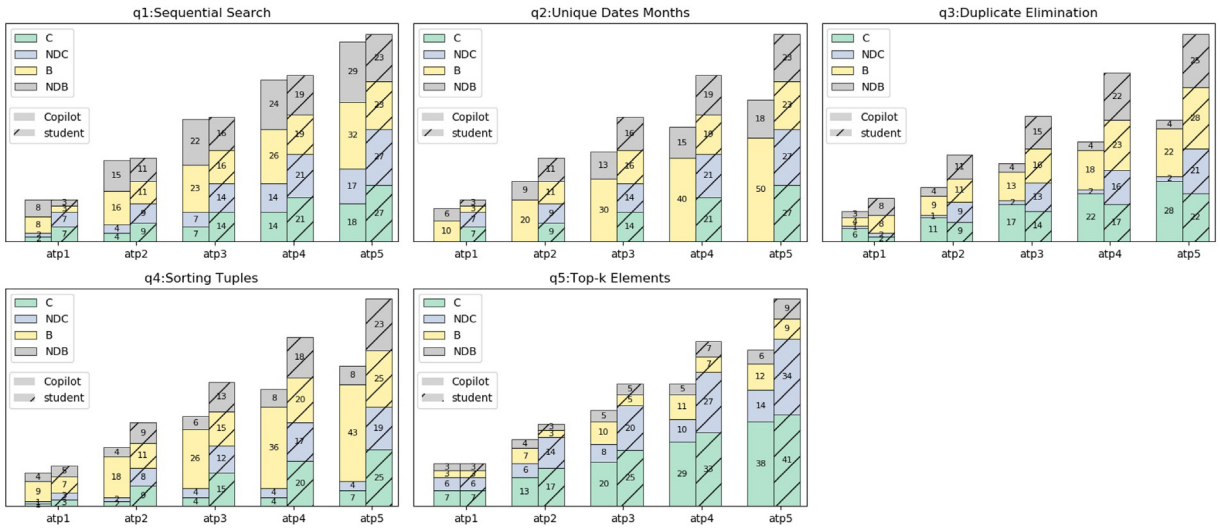


Fig. 10. The cumulative distribution of solutions by Copilot and students. It shows the cumulative distribution of Correct (C), Non-duplicate Correct (NDC), Buggy (B), and Non-duplicate Buggy (NDB) solutions for Copilot and students. Attempts (atp) for students equal to a random sampleset of their submission. Each value on the stack represents the number of solutions in each of the 4 categories. The growth of NDC solutions for Copilot's solutions decreases or stops for some programming tasks while the number of its Correct (C) solutions increases. Students' submissions are more diverse than Copilot's solutions.

Table 6
The Cyclomatic Complexity (C.C.) of Copilot's solutions compare to students' submissions.

	Question	C.C. Copilot	C.C. Students
q1	Sequential Search	5.8 ± 1.94	4.63 ± 2.1
q2	unique dates Months	–	4.18 ± 1.03
q3	Duplicate Elimination	3 ± 0.01	3.12 ± 0.5
q4	Sorting Tuples	1 ± 0	4.13 ± 1.03
q5	Top_k Elements	1.44 ± 0.69	3.3 ± 1.46
	Total	**2.81**	3.87

As another example for Copilot, there is no more NDC solution after "atp3" for "q3" and "q5". This means that by increasing the number of solutions generated by Copilot for these two questions, the CR increases due to the duplication of correct solutions not generating new ones.

In general, the diversity of correct and buggy submissions for students is more than Copilot. While there is no guarantee that all non-duplicate solutions are optimized, students solved these 5 tasks with more diverse and novel solutions.

4.2.4. The cyclomatic complexity of code

In this section, we calculate the Cyclomatic Complexity (C.C.) of code generated by Copilot and students. Table 6 shows the average and the standard deviation of C.C. for the correct solutions generated by Copilot and students. It is worth mentioning that we use the sampling method explained in Section 3.2.3 to collect students' correct solutions.

On average, the correct solutions suggested by Copilot are found to be less complex than students' solutions. However, we should consider that for example, for "q2", Copilot has no correct solutions, or the CR of Copilot for "q4" is only 8%. Also, for "q5", Copilot used Python built-in functions "Sort" and "Sorted", however, it was asked in the description to not use them.

The low cyclomatic complexity in our results is primarily attributed to the ease of the tasks in our dataset, however, the observed minor variances are attributed to poor coding practices in the students' solutions, like the example demonstrated in Fig. 4.

Although Copilot is not able to match the diversity observed in student solutions according to the results discussed in Section 4.2.3, its solutions are more understandable than solutions suggested by students in terms of their cyclomatic complexity.

4.2.5. Syntactic mastery

As discussed in Section 3.2.4/(5), different developers can solve a programming task with different solutions. Consequently, this can impact the readability and maintainability of the code if it is not an efficient solution.

In this section, we compare the diversity of syntax keywords and the usage of built-in functions between the solutions generated by Copilot and those written by humans for different programming tasks. Fig. 11 shows the diversity of syntax keywords and built-ins that we observed in both Copilot's and students' solutions with normalized values. Students used more diverse keywords and built-ins in comparison to Copilot.

For example, for q3: Duplicate Elimination, the only Python built-in function in Copilot's solutions is "append". However, students included more diverse built-ins such as {'count', 'remove', 'index', 'copy', 'append', 'reverse', 'pop'} in their solutions. As another example, in q5: Top-k elements, Copilot used {'sort', 'append', 'remove'} as built-in functions in all of its solutions but students used {'copy', 'pop', 'remove', 'append', 'sort', 'extend', 'reverse', 'clear'}. The using of programming keywords by Copilot and students is similar to built-ins. For example, for q4: Sorting Tuples, there are solutions provided by students that iterate over the list of tuples to sort them causing diverse syntax patterns in their solutions such as {'Tuple', 'Lt', 'Add', 'Expr', 'Continue', 'Eq', 'Break', 'Gt', 'BoolOp', 'And', 'UnaryOp', 'USub', 'LtE'}. We cannot find these programming patterns in Copilot's solutions as it only used the built-in function "sort" in the majority of its solutions.

Students used more diverse syntax patterns and built-ins to solve the same problem compared to Copilot. This may be the result of students not being familiar with advanced Python features as opposed to Copilot which uses such features frequently. However, this diversity could stem from the diversity of student submissions as discussed in Section 4.2.3, or it could be the result of restriction in some assignments' descriptions, for example in q5: Top_k elements that not using the built-in functions sort and sorted is requested which, unlike the students, Copilot was not able to understand this restriction.

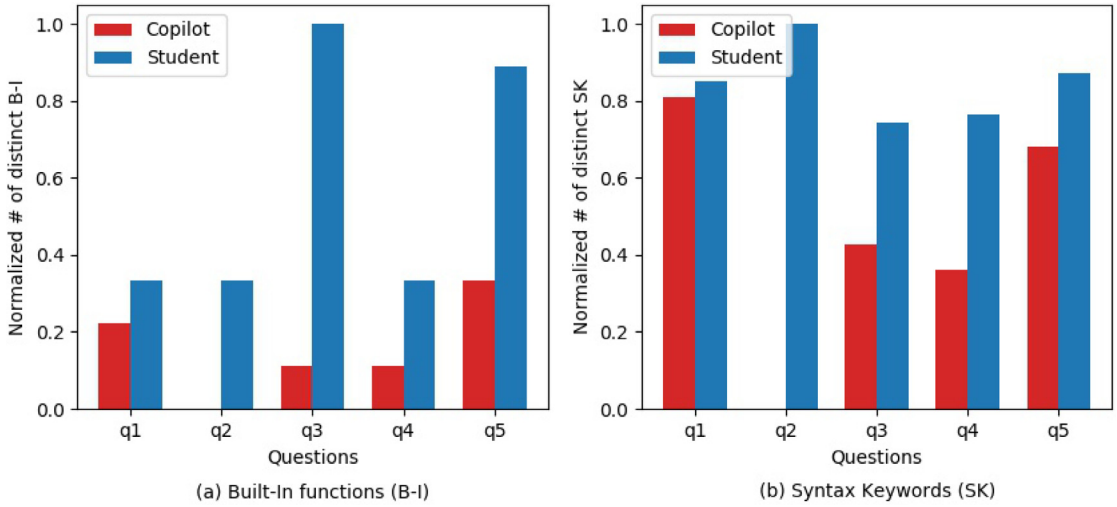


Fig. 11. Diversity of programming Syntax Patterns in Solutions generated by Copilot and Students. Plot (a) shows the normalized value for the distinct number of Python built-in functions in Copilot's solutions compared to students' for different questions. Plot (b) shows the normalized value for the distinct number of Python Syntax keywords in Copilot's solutions compared to students.

Findings: In general, Copilot suggests solutions that compete with students' submissions in different aspects. The correctness ratio and diversity of students' submissions are greater than Copilot's. However, the cost of repairing buggy solutions generated by Copilot is less than students'. In addition, the complexity of Copilot's generated code is less than students'.
Challenges: Copilot has difficulty understanding some requirements in the description of tasks. This affects the correctness ratio of its solutions. However, students understand those details and consider them in their submissions.

5. Discussion and limitation

In this section, we discuss the boundaries of Copilot and how to make it more beneficial in real programming tasks despite its limitations.

5.1. Description of problems (prompts)

Our results show that Copilot cannot understand some details in the description of problems that are understandable by humans. For example, in q5, "Top-k Elements", it is asked in the description to "... not use Python's built-in functions sort and sorted ...". Copilot cannot understand this detail and uses these two built-in functions in all of the correct solutions. However, the majority of students avoided using these built-in functions. Instead, they wrote a sorting algorithm and then called it for sorting tasks or used other built-in functions such as "append", "remove" and "max" in their solutions. As our results in Section 4.1 show, Copilot suggests correct solutions for different sorting algorithms (meaning that Copilot is familiar with different sorting algorithms such as "Bubble Sort" or "Merge Sort"), but it did not use them in q5 because it could not figure out the requirements of the problem. But students apply their knowledge about sorting algorithms to solve this problem. Thus, since in the prompt, we cannot limit Copilot to NOT using certain functions, instead, it is better to clarify our task by defining functions that it is allowed to use.

In q4, "Sorting Tuple", it is asked to return the list of tuples in an order that "... older people are at the front ...". Copilot cannot understand this part. In 92% of suggestions, it returned the sorted tuples in the default order: *ascending*. However, students considered this point in their submission. We even checked some of the buggy submissions by students. Our observations show that even in buggy submission, students considered the correct order of sorting. It means that they fully understood what the point of sorting tuples is in a way that "...older people are at the front...".

Copilot shows similar limitations on algorithmic problems. For example, when asking Copilot to implement the "activity" class in Section 4.1.4, Copilot cannot understand putting limits on variables even though it was asked to do so explicitly. Another limitation is its difficulties in understanding long descriptions which are also observed by Li et al. (2022). Throughout our testing in Sections 4.1 and 4.2, we observed that Copilot might misunderstand the problem entirely if the description contains multiple sentences (whether short or long).

5.2. Experimental suggestions

Furthermore, for more exploration on how to change prompt to meet the target solution, we performed some experiments by applying different scenarios and discussing their impacts on the results.

Scenario#1: In this scenario, we changed "...older people are at the front..." to "...descending order..." in the description of q4 and repeated the process with Copilot to generate solutions. This small change improves the CR from 14% to 79%. This improvement shows there are some details/keywords in the description of problems that seem obvious to humans, but Copilot cannot understand those details in natural language. If we change those details into programming specific/technical keywords such as "descending", it can help Copilot recommend relevant solutions.

Scenario#2: We have a similar observation for q2, "Unique Birthday", where the Copilot cannot understand the requirements mentioned in the description, however, all students considered it. In this question, it is asked for "...implement 3 different functions unique_day, unique_month and contains_unique_day...", to address the problem. Copilot could not understand this condition. Unit tests for q2 are testing all 3 functions. Thus, the CR of Copilot for q2 equals zero because all 50 solutions in different attempts have failed on some of the test units.

So, in this scenario, we gave 3 separate descriptions to Copilot for unique_day, unique_month, and contains_unique _day functions in the same source file. Here is the revised description that we used:

- **unique_day:** Given a day and a list of possible birthday dates return True if there is only one possible birthday with that day, and False otherwise.
- **unique_month:** Given a month and a list of possible birthday dates, return True if there is only one possible birthday within that month, and False otherwise.
- **contains_unique_day:** Given a month and a list of possible birthday dates, return True if there is only one possible birthday with that month and day, and False otherwise.

We start with the description of unique_day at the first line of the source file. Then, we accepted the first solution suggested by Copilot. We continued with the description of unique_month in the next line and accepted the first suggested solution and followed the same instruction for contains_unique_day. We repeat the process 50 times to generate 50 solutions that contain 3 separate functions. Copilot even calls the unique_day function in some of its suggestions for the contains_unique_day function. You can find sample solutions in the replication package. Since there are separate unit tests to test each function separately, we run related tests against each function. In this scenario, the CR of unique_day, unique_month, and contains_unique_day are 88%, 0%, and 40% respectively.

While the original description was clear to students, Copilot could not understand it. Instead of asking Copilot to solve the problem with different functions, we divide a problem into 3 different problems. It increases the CR for unique_day and contains_unique_day. However, the CR of unique_month is still zero. In the following, we investigate this case with a different scenario.

Scenario#3: Since Copilot could not find any correct solutions for unique_month, we manually checked its suggested solutions. We found that in all buggy solutions, Copilot refers to the second item of the "birthday" tuple in the list of birthday dates as the month. However, unit tests consider month as the first item of tuples to test the functionality of the method. For example, consider below unit test:

- unique_month (Month = "January", Birthdays = [("January","1"), ("January", "2")]).

In each tuple in the list of birthdays, for example, ("January", "1"), Copilot referred to the second item as a month, however, the first item in the tuple is the birthday month.

In the description of "unique_month", we added the above unit test as a sample input, at the end of the description. It improves the CR of "unique_month" from 0% to 91%. It shows that adding sample input or sample unit test in the description of problems can help Copilot to generate more correct solutions.

In addition, we randomly checked 20% of students' submissions (both correct and buggy). Our observation shows that none of them assumed any wrong structure for the input data, while the structure of input is not clear in the description of the question. Thus, we assume that there is some extra clarification between students and the lecturer about the structure of the input.

6. Threats to validity

We now discuss the threats to the validity of our study following the guidelines provided by Wohlin et al. (2012) for experimentation in software engineering.

6.1. Internal validity

The threat to internal validity comes from the fact that Copilot is closed-source. We cannot analyze our results based on the characteristics (and expected behavior) of Copilot's trained model. This is also the case for Copilot's training data, hence we are not able to indicate whether it memorized the solutions to these inquiries from its training set or whether it generates a unique solution. Similar to other researchers (Nguyen and Nadi, 2022; Imai, 2022; Vaithilingam et al., 2022; Chen et al., 2021), we can only investigate Copilot's functionality in suggesting code for the provided prompt.

Also, as our experiments have shown, Copilot's suggestions change over time and are not always consistent. This may come from the inconsistency stemming from the nature of LLMs and also the continuous improvement of Copilot's engine as an ML product, perhaps by feeding new code samples or learning from new queries submitted to Copilot. As a result, we cannot guarantee that other researchers will receive the same suggestions and results that we obtained by performing the same experiments.

6.2. External validity

The lack of a dataset that comes from an industrial context and contains programming task statements along with their corresponding code drives us to follow the path of other research in software engineering using classical programming tasks to study Copilot's competence (Vaithilingam et al., 2022; Sobania et al., 2021a; Nguyen and Nadi, 2022; Drori and Verma, 2021; Tang et al., 2021). There are different advantages to these types of programming tasks that we discussed in Sections 3.1 and 3.2.1. To highlight two advantages, first, Copilot is able to generate answers corresponding to these task descriptions. Thus, we could apply our assessments beyond the correctness of the suggested solutions. Also, the task descriptions in our datasets are human-written and it decreases the possibility of the memorization issue in LLMs. But these programming tasks are not representative of the whole programming tasks in real software projects.

Considering the choice of programming tasks and to have a fair comparison, we compared Copilot with students in a Python programming course. While we have no information about the background and characteristics of the participants, we assume that they are good representatives of junior developers in real software projects, but they may not be representatives of the whole population.

6.3. Conclusion validity

To mitigate the threats to the validity of our conclusions, we choose different quantitative metrics, based on other studies in software engineering, to compare Copilot's code with humans' (Fakhoury et al., 2019; Nguyen and Nadi, 2022; Kim and Whitehead, 2006). Even though these quantitative metrics reduce the chance of having biased conclusions, they do not enable us to conduct any qualitative assessment such as how humans interact with the tool.

6.4. Construct validity

The threat to the construct validity of our study stems from the fact that all the features and capacities of a good AI pair programmer cannot be captured by quantitative metrics. Since pair programming is an interaction between human–human and human–tool, the opinion of humans about their experience in using such tools as an AI pair programmer is also required for a comprehensive study. For example, someone may prefer a pair programming tool that accepts voice commands to a tool that suggests a list of possible solutions because they like the discussion part of pair programming more than seeing a list of suggestions.

7. Conclusion

In this paper, we have studied Copilot's ability to code generation and compared its generated code with those of humans. Our results show that Copilot is able to generate correct and optimal solutions for some fundamental problems in algorithm design. However, the quality of the generated code depends greatly on the conciseness and depth of the prompt that is provided by the developer. Furthermore, our results indicate that Copilot still needs more development in fully understanding natural language utterances in order to be able to fill in the position of a pair programmer. Copilot may occasionally be unable to generate code that satisfies all the criteria described in the prompt, but the generated code can be incorporated with little to moderate changes to the provided prompt or the code.

Although Copilot suggests solutions that are more advanced in programming than solutions provided by junior developers, and even though those solutions are comparable to humans' in correctness, optimality, reproducibility, and repair costs, an expert developer is still required to detect and filter its buggy or non-optimal solutions. Thus, Copilot can be an asset in software projects if it is used by expert developers as a pair programmer but it can turn into a liability if it is used by those who are not familiar with the problem context and correct coding methods.

Given that Copilot has recently been released as a commercial product, a new wave of developers will have access to it. This will undoubtedly enrich Copilot's training dataset and will also expose more of its shortcomings. However, Copilot solutions can be troublesome if novice developers/students fully trust them, on the other hand, we hypothesize that Copilot's suggestions may help them in improving their programming skills. Therefore, as future work, a tool or a layer on top of Copilot that can filter out buggy and non-optimal suggestions will reduce the liability of using this tool in software projects. Future works can also use our study design and explore more diverse programming tasks with heterogeneous participants in a human-centered study, to more comprehensively compare Copilot with humans as an AI pair programmer.

CRediT authorship contribution statement

Arghavan Moradi Dakhel: Conceptualization, Methodology, Software, Data collection, Visualization, Validation, Investigation, Writing – original draft, Writing – review & editing. **Vahid Majdinasab:** Methodology, Data collection, Writing – original draft, Writing – review & editing. **Amin Nikanjam:** Conceptualization, Methodology, Writing – review & editing. **Foutse Khomh:** Supervision, Reviewing and editing. **Michel C. Desmarais:** Supervision, Reviewing and editing. **Zhen Ming (Jack) Jiang:** Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

We shared our code and data as a replication package: https://github.com/Copilot-Eval-Replication-Package/CopilotEvaluation.

Acknowledgments

This work is partially supported by the Fonds de Recherche du Québec (FRQ), the Canadian Institute for Advanced Research (CIFAR), and the National Science and Engineering Research Council of Canada (NSERC).

References

Ahmed, U.Z., Srivastava, N., Sindhgatta, R., Karkare, A., 2020. Characterizing the pedagogical benefits of adaptive feedback for compilation errors by novice programmers. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training. pp. 139–150.

Alur, R., Bodik, R., Juniwal, G., Martin, M.M., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A., 2013. Syntax-Guided Synthesis. IEEE.

Arcuri, A., 2008. On the automation of fixing software bugs. In: Companion of the 30th International Conference on Software Engineering. pp. 1003–1006.

Asare, O., Nagappan, M., Asokan, N., 2022. Is GitHub's copilot as bad as humans at introducing vulnerabilities in code? arXiv preprint arXiv:2204.04741.

Becker, B.A., Denny, P., Finnie-Ansley, J., Luxton-Reilly, A., Prather, J., Santos, E.A., 2022. Programming is hard–or at least it used to be: Educational opportunities and challenges of AI code generation. arXiv preprint arXiv:2212.01020.

Bera, R.K., Bera, R.K., 2020. Fundamental limits to computing. In: The Amazing World of Quantum Computing. Springer, pp. 171–206.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al., 2020. Language models are few-shot learners. Adv. Neural Inf. Process. Syst. 33, 1877–1901.

Carlini, N., Ippolito, D., Jagielski, M., Lee, K., Tramer, F., Zhang, C., 2022. Quantifying memorization across neural language models. arXiv preprint arXiv:2202.07646.

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

Clement, C.B., Drain, D., Timcheck, J., Svyatkovskiy, A., Sundaresan, N., 2020. PyMT5: multi-mode translation of natural language and python code with transformers. arXiv preprint arXiv:2010.03150.

Cohen, J., 1960. A coefficient of agreement for nominal scales. Educ. Psychol. Measur. 20 (1), 37–46.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., 2022a. Introduction to Algorithms, fourth ed. MIT Press.

Cormen, T.H., Leiserson, C.E., Ronald L Rivest, C.S., 2022b. Introduction to algorithms reviews. https://www.goodreads.com/book/show/58064696-introduction-to-algorithms.

Dantas, C.E.C., Maia, M.A., 2021. Readability and understandability scores for snippet assessment: An exploratory study. arXiv preprint arXiv:2108.09181.

Denny, P., Kumar, V., Giacaman, N., 2023. Conversing with copilot: Exploring prompt engineering for solving CS1 problems using natural language. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education Vol. 1. pp. 1136–1142.

dos Santos, R.M., Gerosa, M.A., 2018. Impacts of coding practices on readability. In: Proceedings of the 26th Conference on Program Comprehension. pp. 277–285.

Drechsler, R., Harris, I.G., Wille, R., 2012. Generating formal system models from natural language descriptions. In: 2012 IEEE International High Level Design Validation and Test Workshop. HLDVT, IEEE, pp. 164–165.

Drori, I., Verma, N., 2021. Solving linear algebra by program synthesis. arXiv preprint arXiv:2111.08171.

Ebert, C., Cain, J., Antoniol, G., Counsell, S., Laplante, P., 2016. Cyclomatic complexity. IEEE Softw. 33 (6), 27–29.

Fakhoury, S., Roy, D., Hassan, A., Arnaoudova, V., 2019. Improving source code readability: Theory and practice. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension. ICPC, IEEE, pp. 2–12.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al., 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.

Finnie-Ansley, J., Denny, P., Becker, B.A., Luxton-Reilly, A., Prather, J., 2022. The robots are coming: Exploring the implications of OpenAI codex on introductory programming. In: Australasian Computing Education Conference. pp. 10–19.

Forsgren, N., Storey, M.-A., Maddila, C., Zimmermann, T., Houck, B., Butler, J., 2021. The SPACE of developer productivity: There's more to it than you think. Queue 19 (1), 20–48.

Fronza, I., Sillitti, A., Succi, G., 2009. An interpretation of the results of the analysis of pair programming during novices integration in a team. In: 2009 3rd International Symposium on Empirical Software Engineering and Measurement. IEEE, pp. 225–235.

Geeksforgeeks Team, 2022. GeeksForGeeks. https://www.geeksforgeeks.org.

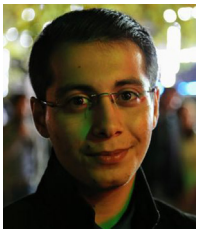
Gulwani, S., 2010. Dimensions in program synthesis. In: Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming. PPDP '10, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450301329, pp. 13–24. http://dx.doi.org/10.1145/1836089.1836091.

Gulwani, S., Radiček, I., Zuleger, F., 2018. Automated clustering and program repair for introductory programming assignments. ACM SIGPLAN Not. 53 (4), 465–480.

Harris, C.B., Harris, I.G., 2016. Glast: Learning formal grammars to translate natural language specifications into hardware assertions. In: 2016 Design, Automation & Test in Europe Conference & Exhibition. DATE, IEEE, pp. 966–971.

Hovemeyer, D., Pugh, W., 2004. Finding bugs is easy. SIGPLAN Not. (ISSN: 0362-1340) 39 (12), 92–106. http://dx.doi.org/10.1145/1052883.1052895.
Hovemeyer, D., Spacco, J., Pugh, W., 2005. Evaluating and tuning a static analysis to find null pointer bugs. In: Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. pp. 13–19.
Hu, Y., Ahmed, U.Z., Mechtaev, S., Leong, B., Roychoudhury, A., 2019. Re-factoring based program repair applied to programming assignments. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 388–398.
Imai, S., 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. pp. 319–321.
Kim, S., Whitehead, Jr., E.J., 2006. How long did it take to fix bugs? In: Proceedings of the 2006 International Workshop on Mining Software Repositories. pp. 173–174.
Leinonen, J., Hellas, A., Sarsa, S., Reeves, B., Denny, P., Prather, J., Becker, B.A., 2023. Using large language models to enhance programming error messages. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education, Vol. 1. In: SIGCSE 2023, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450394314, pp. 563–569. http://dx.doi.org/10.1145/3545945.3569770.
Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A.D., et al., 2022. Competition-level code generation with alphacode. arXiv preprint arXiv:2203.07814.
Lui, K.M., Chan, K.C., 2006. Pair programming productivity: Novice–novice vs. expert–expert. Int. J. Hum.-Comput. Stud. 64 (9), 915–925.
Manna, Z., Waldinger, R., 1980. A deductive approach to program synthesis. ACM Trans. Programm. Lang. Syst. (TOPLAS) 2 (1), 90–121.
Maruping, L.M., Zhang, X., Venkatesh, V., 2009. Role of collective ownership and coding standards in coordinating expertise in software project teams. Eur. J. Inf. Syst. 18 (4), 355–371.
Mihalcea, R., Liu, H., Lieberman, H., 2006. NLP (Natural Language Processing) for NLP (Natural Language Programming). In: International Conference on Intelligent Text Processing and Computational Linguistics. Springer, pp. 319–330.
Moradi, et al., 2022. Replication package. GitHub Repository, GitHub, https://github.com/Copilot-Eval-Replication-Package/CopilotEvaluation.
Moradi Dakhel, A., C. Desmarais, M., Khomh, F., 2021. Assessing developer expertise from the statistical distribution of programming syntax patterns. In: Evaluation and Assessment in Software Engineering. pp. 90–99.
Moroz, E.A., Grizkevich, V.O., Novozhilov, I.M., 2022. The potential of artificial intelligence as a method of software developer's productivity improvement. In: 2022 Conference of Russian Young Researchers in Electrical and Electronic Engineering. ElConRus, IEEE, pp. 386–390.
Nguyen, N., Nadi, S., 2022. An empirical evaluation of GitHub Copilot's code suggestions. In: Accepted for Publication Proceedings of the 19th ACM International Conference on Mining Software Repositories. MSR, pp. 1–5.
Papineni, K., Roukos, S., Ward, T., Zhu, W.-J., 2002. Bleu: A method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics. pp. 311–318.
Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R., 2022. Asleep at the keyboard? Assessing the security of GitHub copilot's code contributions. In: 2022 2022 IEEE Symposium on Security and Privacy (SP). SP, IEEE Computer Society, Los Alamitos, CA, USA, pp. 980–994. http://dx.doi.org/10.1109/SP46214.2022.00057, URL https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00057.
Plonka, L., Sharp, H., Van der Linden, J., Dittrich, Y., 2015. Knowledge transfer in pair programming: An in-depth analysis. Int. J. Hum.-Comput. Stud. 73, 66–78.
Rahit, K., Nabil, R.H., Huq, M.H., 2019. Machine translation from natural language to code using long-short term memory. In: Proceedings of the Future Technologies Conference. Springer, pp. 56–63.
Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., Ma, S., 2020. Codebleu: A method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297.
Salazar Paredes, P., et al., 2020. Comparing Python Programs Using Abstract Syntax Trees. Technical Report, Uniandes.
Sarwar, M.M.S., Shahzad, S., Ahmad, I., 2013. Cyclomatic complexity: The nesting problem. In: Eighth International Conference on Digital Information Management. ICDIM 2013, IEEE, pp. 274–279.
Scalabrino, S., Bavota, G., Vendome, C., Linares-Vasquez, M., Poshyvanyk, D., Oliveto, R., 2019. Automatically assessing code understandability. IEEE Trans. Softw. Eng. 47 (3), 595–613.
Sobania, D., Briesch, M., Rothlauf, F., 2021a. Choose your programming copilot: A comparison of the program synthesis performance of GitHub copilot and genetic programming. arXiv preprint arXiv:2111.07875.
Sobania, D., Schweim, D., Rothlauf, F., 2021b. Recent developments in program synthesis with evolutionary algorithms. arXiv preprint arXiv:2108.12227.
Tang, L., Ke, E., Singh, N., Verma, N., Drori, I., 2021. Solving probability and statistics problems by program synthesis. arXiv preprint arXiv:2111.08267.
Tran, N., Tran, H., Nguyen, S., Nguyen, H., Nguyen, T., 2019. Does BLEU score work for code migration? In: 2019 IEEE/ACM 27th International Conference on Program Comprehension. ICPC, IEEE, pp. 165–176.
Vaithilingam, P., Zhang, T., Glassman, E.L., 2022. Expectation vs. Experience: Evaluating the usability of code generation tools powered by large language models. In: CHI Conference on Human Factors in Computing Systems Extended Abstracts. pp. 1–7.
W3schools Team, 2022. W3schools. https://www.w3schools.com.
Wermelinger, M., 2023. Using GitHub Copilot to solve simple programming problems. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1. In: SIGCSE 2023, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450394314, pp. 172–178. http://dx.doi.org/10.1145/3545945.3569830.
Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. Experimentation in Software Engineering. Springer Science & Business Media.
Zhang, F., Khomh, F., Zou, Y., Hassan, A.E., 2012. An empirical study on factors impacting bug fixing time. In: 2012 19th Working Conference on Reverse Engineering. IEEE, pp. 225–234.
Ziegler, A., Kalliamvakou, E., Simister, S., Sittampalam, G., Li, A., Rice, A., Rifkin, D., Aftandilian, E., 2022. Productivity assessment of neural code completion. arXiv preprint arXiv:2205.06537.

Arghavan Moradi Dakhel is currently a Ph.D. Candidate in Software Engineering at Polytechnique Montréal. Her study is focused on modeling the expertise of developers and exploring characteristics and tools that impact developers' proficiency and productivity. She received her B.Sc. degree from University of Guilan in 2013 and her M.Sc. Degree from Shahid Beheshti University in 2016. Her research interest stands at the intersection of machine learning/deep learning, software engineering, mining software repositories, recommender systems, and human–computer/AI interaction.

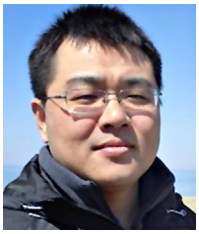


Vahid Majdinsasab received his Bachelor of Science in Information Technology from University of Tabriz. He received his Master of Science in Artificial Intelligence and Robotics from K.N. Toosi University of Technology. He is currently pursuing a Ph.D. degree in Software Engineering at Polytechnique Montréal, with a focus on using Artificial Intelligence for Software Engineering. Vahid's research interests include Reinforcement Learning, Multi-Agent Systems, Evolutionary Algorithms, and Large Language Models trained on code. His research aims to advance the state-of-the-art in applying machine learning to software engineering, with the goal of developing efficient, reliable, and intelligent software systems.

Amin Nikanjam is a research associate in the SWAT research team at Polytechnique Montréal. He is studying (1) how Software Engineering practices (like testing and fault localization) can be leveraged to Machine Learning Software Systems, and (2) how Machine Learning techniques can be applied for safety-critical systems in terms of reliability, robustness, and explainability. He received his Master's and Ph.D. in Artificial Intelligence from Iran University of Science and Technology, Iran, and his Bachelor's in Software Engineering from University of Isfahan. Before joining Polytechnique Montréal, he was an invited researcher at University of Montréal, and before that, he was an assistant professor at K.N. Toosi University of Technology, Iran. His research interests include Systems Engineering for Machine Learning, (Deep) Reinforcement Learning, and Multi-Agent Systems.

Foutse Khomh is a full professor, a Canada CIFAR AI Chair, and FRQ-IVADO Research Chair at Polytechnique Montréal, where he heads the SWAT Lab (http://swat.polymtl.ca/). He received a Ph.D. in Software Engineering from the University of Montreal in 2011. His research interests include software maintenance and evolution, cloud engineering, machine learning systems engineering, empirical software engineering, software analytics, and dependable and trustworthy AI/ML. His work has received four ten-year Most Influential Paper (MIP) Awards, and six Best/Distinguished Paper Awards. He has served on the program committees of several international conferences including ICSE, FSE, ASE, ICSM(E), SANER, MSR, ICPC, SCAM, ESEM and has reviewed for top international journals such as SQJ, JSS, EMSE, TSE, TPAMI, and TOSEM.

Michel C. Desmarais is a full professor at the Computer and Software Engineering Department of Polytechnique Montreal since 2002. After his Ph.D. in Psychology at the University of Montreal, he spent ten years at the Montreal Computer Research Institute (CRIM) as scientific lead of a research team. His research interests include Artificial Intelligence, Human–Computer Interfaces, Recommender Systems, Educational Data Mining and User Modeling, and Probabilistic and Cognitive modeling. He was editor of the Journal of Educational Data Mining from 2013 to 2017. Further information about him can be found at https://www.polymtl.ca/expertises/en/desmarais-michel-c.



Zhen Ming (Jack) Jiang is an associate professor at the Department of Electrical Engineering and Computer Science, York University. He received his Ph.D. from the School of Computing at Queen's University, MMath and BMath degrees from the David R. Cheriton School of Computer Science at the University of Waterloo. During his Ph.D. studies, he worked with the Performance Engineering team at BlackBerry (RIM). Tools resulted from his research are currently used daily to monitor and debug the health of several ultra-large commercial software systems within BlackBerry.