2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

Measuring the Runtime Performance of C++ Code Written by Humans using GitHub Copilot

Daniel Erhabor
University of Waterloo
Waterloo, Canada
derhabor@uwaterloo.ca

Sreeharsha Udayashankar
University of Waterloo
Waterloo, Canada
s2udayas@uwaterloo.ca

Meiyappan Nagappan
University of Waterloo
Waterloo, Canada
m2nagapp@uwaterloo.ca

Samer Al-Kiswany
University of Waterloo
Waterloo, Canada
alkiswany@uwaterloo.ca

***Abstract*—GitHub Copilot is an artificially intelligent programming assistant used by many developers. While a few studies have evaluated the security risks of using Copilot, there has not been any study to show if it aids developers in producing code with better runtime performance. We evaluate the runtime performance of C++ code produced when developers use GitHub Copilot versus when they do not. To this end, we conducted a user study with 32 participants where each participant solved two C++ programming problems, one with Copilot and the other without it and measured the runtime performance of the participants' solutions on our test data. Our results suggest that using Copilot may produce C++ code with (statistically significant) slower runtime performance.**

I. Introduction

Advances in natural language processing and deep learning have resulted in large language models (LLMs) that can generate code from free-form text. With this, language models like GPT-3 [1] have been extended to what Xu et al. [2] have termed Natural-Language-to-Code (NL2Code) generators. Notably, Open AI's extension of the GPT-3 language model, Codex [3], and the production-ready product derived from it, GitHub Copilot [4], are popular examples of NL2Code tools in use today. In a recent StackOverflow survey, 44% of developers state that they use LLM-based tools in their development process already, and 26% plan to use such tools soon [5]. While some studies show that developers may have a positive experience using GitHub Copilot [6], others show that it could generate potentially vulnerable code [7].

We present the first-ever evaluation of Copilot from a runtime performance perspective in systems programming. We focus on runtime performance as it is critically important in large-scale systems. Google notes that a few additional seconds of page load latency can increase customer bounce rates by 90% [8]. Amazon reports that 100 milliseconds of latency cost them millions of dollars in revenue [9]. Each millisecond of additional latency costs financial firms $100 million every year [10]. Thus, large-scale systems designed to maximize performance measure and report metrics such as their tail latencies and throughputs [11], [12].

We conducted the first user-based study on Copilot to evaluate the runtime performance of the C++ code generated when developers use it. With the results from our study, we answer the following research questions:

RQ0: Does using Copilot influence program correctness?

RQ1: Is there a runtime performance difference in C++ code when using GitHub Copilot?

RQ2: Do Copilot's suggestions sway developers towards or away from C++ code with faster runtime performance?

To answer these questions, we conducted a user study involving 32 participants with systems programming experience. Each participant solved two programming problems in C++; one was solved with Copilot and the other was solved without it. The problems were related to I/O operations and concurrent programming. We selected problems related to these two domains as they directly impact the code runtime. We compared the runtime performance of Copilot-aided solutions against Copilot-unaided solutions, obtained survey responses from participants after they completed the study, and analyzed the video recordings of participants solving the problems.

Our findings indicate that using Copilot resulted in C++ code with (statistically significant) slower runtime performance. Specifically, Copilot-unaided solutions were 26% faster than Copilot-aided solutions on average for the I/O-related problem and 15% faster for the concurrent programming problem. Our expert solutions to the problems had up to 6× faster runtime performance compared to the average Copilot-aided solution. Additionally, Copilot's aid tended to tilt developers towards code with slower runtime performance. Finally, as expected, higher developer experience and familiarity with the C++ programming language were correlated with faster runtime performance.

The rest of this paper is organized as follows: We provide background related to GitHub Copilot and related work in Section II. The process of creating the problems solved by the participants and the rationale behind choosing the problems is described in Section III. Our model solutions are elaborated in Section IV. A summary of the participant recruitment process and the participants is described in Section V. We present the experiment design in detail in Section VI where we cover the tasks that participants solved, how tasks were split across the participants, and the rationale behind it. We analyze and discuss the results of our study, answering our research questions in Section VII. We include a discussion on the participants and their familiarity with the problems in Section VIII. Penultimately, we discuss the threats to validity of our study in Section IX. Finally, in Section X, we discuss the takeaways and potential future directions.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00059

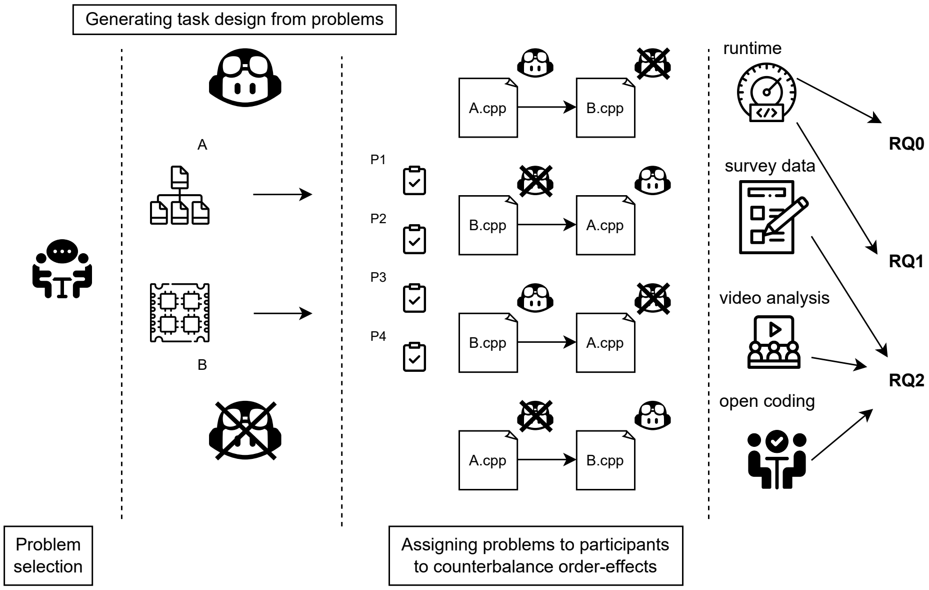


Fig. 1: Overview of Methodology

II. Background and Related Work

GitHub Copilot, the production-ready tool based on the Codex model by Open AI, can be used as a Visual Studio Code extension to suggest code snippets to users. Users can receive suggestions by starting to write code or by writing comments; either way, Copilot will suggest some snippets [4].

A. Related Work

Runtime performance of code generated by Copilot or ChatGPT. A recent study [13] analyzes the correctness and runtime performance of solutions produced by Copilot using problems from the HumanEval [14] dataset. They focus on comparing the multiple solutions suggested by Copilot. Doderlein et al. [15] and Elnashar et al. [16] analyze the impact of prompt engineering on the runtime performance of solutions generated by Copilot and ChatGPT respectively. Mastropaolo et al. [17] analyze the solutions generated by Copilot for different semantically equivalent task descriptions. Nascimento et al. [18] evaluate the runtime performance of code produced by ChatGPT [19] for various LeetCode problems to their human-written solutions provided within LeetCode. All of the past studies focus on the solutions generated by Copilot or ChatGPT on their own and do not look at the scenario where humans are using such tools to help write code. In our study we examine code written by humans who use Copilot. This is a key difference between past work and our work.

Security. Several studies within previous literature examine the security aspects of solutions generated by Copilot. One of the earliest such studies by Pearce et al. [7] sought to understand how often suggestions from Copilot were vulnerable to security attacks and the contexts which made Copilot suggest vulnerable code. To achieve this, they prompted Copilot to generate code in scenarios where the resulting solutions could be either vulnerable or secure. 40% of the programs produced in these scenarios were discovered to be vulnerable.

A study by Sandoval et al. [20] assesses the security of code written by student programmers when assisted by an NL2Code assistant (OpenAI's `code-cushman-001` model) like Copilot. They conducted a between-subjects study with 58 computer science students where participants were tasked with implementing operations of a Singly-Linked List in C. Contrary to the previous study [7], their results showed that Copilot had no conclusive impact on security.

Asare et al. [21] use a previously curated set of common C/C++ vulnerabilities from human developers [22] to assess whether Copilot introduces similar vulnerabilities when presented with the same scenarios. They conclude that while Copilot is susceptible to introducing a few previously seen vulnerabilities, it fares better than human developers in a majority of the cases.

Other factors influencing runtime performance. Numerous studies examine the impact of other parameters, such as software refactoring [23] and specific code changes [24] on the runtime performance of open-source software repositories. These are orthogonal to our paper as we focus on evaluating the code generated by GitHub CoPilot.

CoPilot for alternate problem domains. Drori et al. [25] evaluate the effectiveness of Copilot when generating programmatic solutions to university-level linear algebra problems. Tang et al. [26] use Copilot to tackle university-level probability and statistics problems.

Dakhel et al. [27] report the correctness ratio of solutions generated by Copilot for fundamental algorithmic problems. In addition, they compare Copilot's solutions against student submissions for 5 Python programming assignments. They found that while Copilot often generates ”buggy” code, its repair costs are less than those of similar ”buggy” student solutions. While they report the optimality of Copilot’s solutions to the algorithmic problems, they do not report the runtime performance of it’s solutions to the Python assignments or compare them against human submissions.

Imai et al. [28] compare Copilot against human pair programming by having participants develop a text-based minesweeper game in Python. Nguyen et.al [29] evaluate Copilot using 33 randomly chosen questions from LeetCode, primarily focusing on solution correctness and comprehensibility. Liu et al. [30] characterize the correctness and maintainability of ChatGPT’s solutions for 2000 programming tasks. Choudhuri et al. [31] look at the benefits and challenges faced by students when using ChatGPT for software engineering tasks. While this study does compare participants who used other resources with students who used ChatGPT, they did not look at the runtime performance of code.

Sobania et al. [32] compared Copilot’s solutions against programs synthesised using genetic programming. They found that genetic programming models are more expensive to train and can sometimes result in solutions which aren’t easily comprehensible for humans. They primarily examine the correctness of solutions generated using Copilot and not their runtime performance.

Experiment with Humans using Copilot: Unlike a majority of past studies [25], [26], [28], [29], [32] that focus on the solutions generated by Copilot on its own we focus on the scenario where humans are using such tools to help write code. Copilot was never meant to work without a human, at least to date. Therefore, these related studies do not examine Copilot in its intended environment and do not analyze the impact that such tools can have on software developers. Our study on the other hand is a more realistic experimental setting of how humans will use these tools. Finally, studies using Copilot without humans are limited to a set of simpler problems that can be auto-generated in full by Copilot. Thus, we are also able to examine more complex problems. Neither of our problems can be solved by Copilot with just a prompt and without human intervention.

> To the best of our knowledge, we are the first study to carry out a controlled experiment of the runtime performance of code written by humans working in tandem with Copilot.

III. Programming Problems Solved by Participants

Following in the same vein as Pearson et al. [7], we provided *incomplete code* for participants to implement as a solution to a given problem i.e., we provided code stubs and accompanying documentation for the solutions participants were asked to implement. We call the stubs *problems* throughout this paper. These problems were provided to participants as a CPP file containing the function declaration, the unimplemented function definition that participants were expected to implement, i.e., the primary function, initialization functions and sanity checks to verify correctness. A main function was also provided as an entry point to call the initialization functions, the primary function, and the sanity checks in the appropriate order.

A. Problem selection

We chose two problem domains for our programming problems; file-system operations and multi-threaded programming. We chose these areas because problems in those domains directly impact application runtime performance. With file I/O operations accounting for about 30% - 80% of interactions in networked file systems [33], there is a need for file system operations to be fast on storage devices [34]. Choosing a problem related to file systems reflects this demand. Additionally, since modern computing is moving towards a more parallel domain, there is a need to understand the bottlenecks of multi-threaded applications [35] and optimize accordingly. To reflect this, we chose a problem related to false sharing, a typical multi-threading optimization problem [36].

We chose problems that fit the following criteria: (1) the problem must have more than one solution where each solution differs not in correctness but runtime performance, (2) The problem should be solvable with or without Copilot assistance in 30 minutes.

B. Problem A: File System Operations

For Problem A, participants were asked to read records from three large text files. A record is a sequence of 5000 bytes; each file was 1GB. The read operation is specified by `FileCombo`, a struct that specifies which file to read from and at what offset. The `FileCombo` struct also has a buffer to hold the record read from a file.

For this problem, participants received the CPP file and the three text files. The full function signatures, the CPP file, and the accompanying documentation given to participants for Problem A are shown in Appendix A.1 [37].

C. Problem B: Multi-threaded Optimization

For Problem B, participants were asked to use a certain number of threads to set all the values in a source array buffer to zero while setting all the values in a destination array buffer to a particular value. However, they were not allowed to use assignment operations, i.e., move and copy semantics were not allowed on either the source array buffer or the destination array buffer. Participants were only allowed to increment or decrement the values in the respective array buffers. This restriction was in place because we wanted threads to access and modify array items repeatedly, potentially experiencing false sharing.

The full function signatures, the CPP file, and the accompanying documentation given to participants for Problem B are shown in Appendix A.2 [37].

IV. Model Solutions to the Problems

We created *model solutions* to each problem. Because there was more than one solution to each problem, each solution we derived differed only in performance and not correctness.

We itemize our solutions here and categorize them into Levels 0 – 3 (L0 – L3) for Problem A and Levels 0 – 1 (L0 – L1) for Problem B. Higher levels correspond to faster runtime performance i.e. L3 has a faster runtime than L0. Details about each of these implementations for both Problem A and Problem B can be found in Appendix B [37].

A. Problem A Solutions

1) Level 0: A naive solution to Problem A where calls to `open`, `seek`, `read`, and `close` are made for each `FileCombo`.

2) Level 1: Using the knowledge that only three files are being interacted with, we do not need to `open` and `close` a file for each `FileCombo`. This optimization involves first opening all the files in `FILE_NAMES` and closing them only after all `FileCombos` have been processed. This avoids the repeated opening and closing of file descriptors, which is detrimental to runtime performance.

3) Level 2: Within this optimization, we sort the `FileCombos` by `fileId` and break ties by `offset` before reading the files from storage. As a result, reading records within each specific file will be sequential and not random. Such sequential accesses reduce disk response times, thereby improving program runtime performance.

4) Level 3: The combination of the L1 and L2 optimizations we outlined above gives us the L3 optimization level, representing the best model solution to Problem A.

B. Problem B Solutions

1) Level 0: Consider a solution to Problem B using `THREAD_COUNT` concurrent threads. A naive solution to this problem is one where all threads start at indices between 0 and `THREAD_COUNT`-1 in the `src` and `dst` arrays. Each thread then decrements and increments one `Item` in `src` and `dst`, respectively. After processing their respective `Items`, each thread moves `THREAD_COUNT` steps until the next index and processes the `Item` therein. For instance, with a `THREAD_COUNT` of 4, threads would start at indices 0–3, increment and decrement their respective `Items`, before moving 4 steps ahead to their next index.

This is a naive solution because it promotes false sharing. Due to the contiguous nature of the `src` and `dst` arrays, threads working on `Items` with neighboring indices would be operating on the same 64-byte cache lines. As a result, these threads would clash by invalidating each other's cache lines when modifying the `Item` within the `src` and `dst` arrays, leading to *cache thrashing*.

2) Level 1: False sharing can be avoided by dividing each array (`src` and `dst`) into `THREAD_COUNT` slices and assigning a single thread to process each `Item` within a slice. This reduces the probability of mutual cache line invalidation greatly, reducing *cache thrashing*.

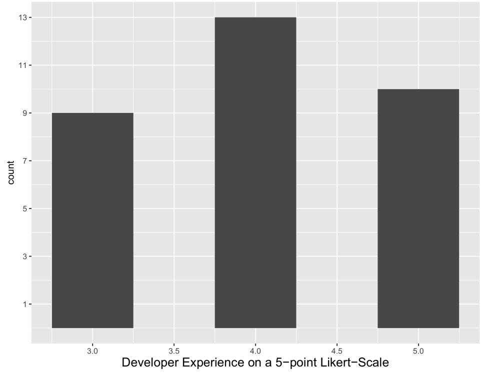


Fig. 2: Distribution of Participants' Developer Experience from Screening Survey on a 5-point Likert-Scale from 1 (No experience) to 5 (10 years or more).

Another solution to false sharing would be to add *padding* within the `Item` struct definition (See Appendix A.2 [37]), bringing its size up to 64 bytes (the cache line size). This would place consecutive `Items` within different cache lines, reducing *cache thrashing*. However, we chose not to allow participants to modify the struct definition as this could lead to longer debugging times, potentially violating the time limit constraint for the problem.

V. Participants

A. Participant Recruitment

Participants were recruited mainly via the mailing list for computer science graduate students and snowballed to other interested participants. We primarily targeted participants with experience in systems programming. We considered participants who met one or more of the following conditions to have satisfied this requirement:

- The participant has been involved professionally in the Systems / Networking domain, either via industry experience or open-source contributions to systems projects.
- The participant has been actively involved in a research project within the Systems / Networking areas.
- The participant has taken one or more university courses within the Systems domain including but not limited to Operating Systems, Distributed Systems, or Computer Networking.

Additionally, potential participants needed to be familiar with C++, and have access to a web browser as well as GitHub Copilot on Visual Studio Code at the time. Finally, participants could not be employed by OpenAI / GitHub or involved with the development of GitHub Copilot at the time.

To check if potential participants were eligible to participate, they were sent a Qualtrics screening survey after they signed the consent form declaring their intent to participate. The screening survey can be found in Appendix C [37].

B. Difficulties Recruiting Professionals

At the halfway point of our desired participant goal, we paused participant recruitment to analyze the preliminary data

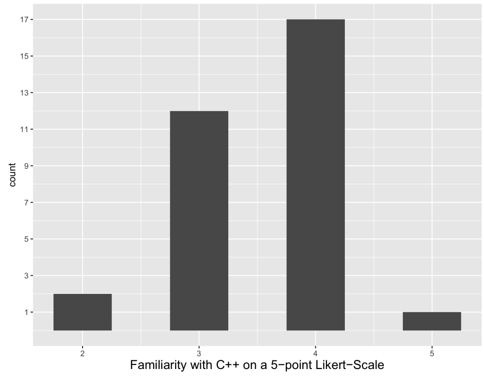


Fig. 3: Distribution of Participants' Familiarity with C++ from Screening Survey on a 5-point Likert-Scale from 1 (Not familiar at all) to 5 (Extremely familiar).

collected. A majority of the preliminary participants thus far had been graduate students with systems experience, i.e., they were part of systems-focused research groups. We decided to diversify our participant pool by including professional systems developers.

The initial recruitment process for professional systems developers started with contacting University of Waterloo alumni working within systems-related roles. Additionally, we looked for contributors to open-source systems projects on GitHub which were primarily implemented in C++. The advanced search feature was used to find projects that contained the keywords `systems`, `operating systems`, or `databases`. We also narrowed our search to include only projects with a dedicated social platform where interested parties connect, such as Discord [38] and Internet Relay Chat (IRC) [39].

While projects such as *SerenityOS* [40] and *SkiftOS* [41] had active Discord communities, their members were disinterested in the study. Attempts to garner interest within these communities were met with suggestions to reach out to other Discord communities such as the *osdev* (Operating Systems Development) [42] discord channel and the associated IRC. Within the *osdev* communities on Discord and IRC, there was a general unwillingness to participate in the study. Community members cited potential copyright issues with Copilot and other negative perceptions of GitHub Copilot, GitHub, and Microsoft as the primary reasons for their unwillingness to participate in the study.

However, our persistent recruitment efforts eventually paid off, as we located willing professional participants, enabling us to meet our desired goal.

C. Participant Summary

We recruited a total of 32 participants for this study, of which 25% were systems programming professionals or contributors to open-source systems projects. Of the remaining participants, one was a sessional lecturer with systems experience at the University of Waterloo, while the rest were graduate students with a systems research focus.

Figures 2 and 3 show the distribution of participants' experience and their familiarity with C++. Further details about the figures can be found in Appendix C [37]. Participants were compensated $50 for their time and the study was approved by the Research Ethics Board (**REB #44162**) at the affiliated university.

VI. Experiment Design

A. Order of Solving the Problems

Given our within-subjects experimental design where one participant solves one problem with Copilot and then the other problem without it, we needed to ensure that any order effects are counterbalanced across all 32 participants. To this end, we present all the possible orders of the *Problems* (**A** and **B**) with the *Modes* (**C** and **NC**) which indicate using Copilot and not using Copilot respectively. The four possible orders of *Mode* × *Problem* are shown in Table I.

The orders in Table I enforced a requirement that our participant pool be a multiple of four. Hence, we recruited a total of 32 participants for the study.

#	*First*	*Second*	*Participant ID*
1	C x A	NC x B	P1
2	C x B	NC x A	P2
3	NC x B	C x A	P3
4	NC x A	C x B	P4

TABLE I: Possible Orders of *Mode* x *Problem*

B. Session Overview

Within this section, we outline the steps carried out in each session. Further details about the tutorial process can be found in Appendix D [37].

1) Pre-session orientation.: The session was conducted remotely via an online conferencing platform. Each session began with the facilitator introducing the study and confirming the participant's consent to be a part of it. After this, screen and audio recording consent for the session was obtained as well. Finally, the facilitator gave the participant a few basic tips for using Copilot such as accepting and rejecting suggestions.

2) Session Goals.: Participants were given two C++ programming problems to solve during the session. Each prompt was self contained within a C++ file and participants were given a compressed archive containing this file. This compressed archive was sent to the participant via the conferencing platform's chat feature (or Google Drive if technical issues occurred).

The participant was asked to extract the contents of the archive but not open them until the facilitator gave them the signal. After verbally confirming that the participant was ready for the screen capture process to begin, they were asked to share their screen and view the C++ file.

The facilitator then confirmed that (1) all extensions except for the Copilot extension were disabled.[1](2) the participant

[1] keybinding related extensions like VSCode Vim [43] and SSH-related extensions like Remote - SSH [44] were the only exceptions allowed

could easily switch between their browser and VSCode. The participants were also reminded that the browser and other online resources could be used in addition to GitHub Copilot.

3) Timing Constraints.: Before commencement, the participants were notified that they had 30 minutes to tackle each problem. Participants were also alerted at regular intervals such as when 20, 10 and 5 minutes were remaining for each problem.

4) After each problem.: Once the participant declared that they were done with a problem (or the timer ran out), the facilitator stopped the timer and notified the participant. They were then instructed to compress their solution and send it back to the facilitator via the conferencing platform, Google Drive or email.

Once this step was completed, participants were asked to deactivate Copilot (if activated) as well as to close their VSCode window, browser window, and any other references they had opened. This was done to prevent any learning effects that could come from Copilot or the participants (e.g., their browser tabs could contain previous search results or references) from carrying over to the second problem. The participants were sent a link to a survey to complete after which they were allowed a break before tackling the second problem.

The instructions and procedure for the second problem were the same as the first, differing only in the survey at the end. The second survey contained demographic questions in addition to the first survey's questions. Details of the first and second surveys are outlined in Appendix E [37].

5) Post session interview.: At the end of the session, participants were asked for their feedback about the study, GitHub Copilot or anything else they wanted to share.

VII. Evaluation

Testbed. Each participant's code was run on a Linux machine with eight-core Intel Xeon D-1548 at 2.0 GHz, 64GB ECC Memory (4 x 16 GB DDR4-2133), and 256 GB NVMe flash storage. The machine was running Ubuntu 20.04 and the code was compiled with gcc version 9.3.0 [45]. In order to minimize the effect of small runtime performance variations, we ran each participant's code 32 times with the filesystem cache cleared between each run.

Errors. If the participant's code did not compile / compiled but encountered runtime errors, it was not analyzed. For instance, one participant's code produced a segmentation fault error even though it compiled successfully. However, if the participants' code compiled, ran without errors but failed the sanity checks, the runtime was recorded but not used in the analysis. As a result, we have only considered correctly implemented solutions when examining the runtime performance.

A. RQ0 - Does using Copilot influence program correctness?

Out of our pool of 32 participants, 16 have attempted to solve Problem A with Copilot while the other 16 tackled the problem without its aid. Among the participants who used Copilot for Problem A, every solution passed the sanity checks. On the other hand, among the participants who tackled Problem A without Copilot, 4 out of 16 code snippets either did not compile (P15 and P7), ran and failed the sanity checks (P3), or ran with errors (P23).

Similarly, 16 participants have attempted to solve problem B with Copilot and 16 without its aid. However, in this case, we observed that only 14 code snippets passed the sanity checks both when Copilot was used and when it was not. The 2 "invalid" solutions where Copilot was used either did not compile (P15) or ran and failed the sanity checks (P32). On the other hand, the 2 "invalid" solutions where Copilot was not used compiled but failed the sanity checks (P30 and P6).

Table II summarizes these invalid solutions. The fields within the table are described below:

- *PartID* - The anonymized ID of the participant
- *Problem* - The problem type (`A` or `B`)
- *Mode* - Whether Copilot was used (`C`) or was not used (`NC`) when tackling the problem
- *Compiled* - Whether the solution was compiled (`TRUE`) or ran into compilation errors (`FALSE`)
- *Passed* - Whether the solution passed sanity checks (`TRUE`) or failed them (`FALSE`). This field has a value of `NULL` if the solution did not compile or ran into runtime errors.

#	*PartID*	*Problem*	*Mode*	*Compiled*	*Passed*
1	P3	A	NC	TRUE	FALSE
2	P7	A	NC	FALSE	NULL
3	P15	A	NC	FALSE	NULL
4	P23	A	NC	TRUE	NULL
5	P15	B	C	FALSE	NULL
6	P32	B	C	TRUE	FALSE
7	P6	B	NC	TRUE	FALSE
8	P30	B	NC	TRUE	FALSE

TABLE II: List of Invalid Runs

Our results suggest that **using Copilot leads developers to produce correct code in most cases**.

B. RQ1 - Is there a runtime performance difference in C++ code when using GitHub Copilot?

1) Approach: To answer this question, we compare the runtime performance of all 32 runs of the participants' source files for Problems A and B. We use the non-parametric Wilcoxon rank sum test in R [46] `wilcox_test()` to compare the runtime performance.

Problem	*Mode*	*Valid Runs*	*Mean*	*Median*	*Min*	*Max*
A	C	16 x 32	34.86 s	34.85 s	33.82 s	36.02 s
A	NC	12 x 32	26.02 s	34.47 s	4.045 s	35.84 s
B	C	14 x 32	1898 ms	945.4 ms	612.1 ms	7356 ms
B	NC	14 x 32	1628 ms	943.9 ms	494.9 ms	6761 ms

TABLE III: Summary Statistics of Runtime Performance

2) Results: On comparing the runtime performance of valid solutions to Problem A with and without Copilot (p = **3.4e-34**), we find the results to be statistically significant. We observe that solutions without using Copilot were about **29%** faster than the ones using Copilot when comparing the mean runtime performance.

Similarly, comparing the runtime performance of the valid solutions to Problem B with and without Copilot (p = **0.000058**), we also find the results to be statistically significant. Again, we observe that solutions without using Copilot were about **15%** faster than the ones using Copilot when comparing the mean runtime performance.

Table III highlights the summary statistics of the runtime performance for participants' valid solutions to the problems. From this we can see that while the mean runtime performance is quite different, the median runtime performance in both problems are closer when comparing solutions created with and without Copilot. Even though the values are closer, not using Copilot still has a marginally faster runtime than using Copilot. This observation from medians along with the min and max values tell us that there are outliers in the data. These outliers matter too.

We notice that the fastest solution to Problem A is when not using Copilot and is 8 times faster than the median solution to the same problem with or without Copilot. However the same participant who wrote the fastest code for Problem A without Copilot had an average runtime performance of approximately 905 ms for Problem B when using Copilot. This value is much closer to the median as we can see from Table III. Thus, we can see that the same participant when using Copilot did not write the same high performance code. Note also that the max times are always faster when not using Copilot than when using Copilot. From all these comparisons and the statistical testing, we can see a picture emerging where participants who used Copilot always wrote code that has slower runtime performance than than those who did not.

For further context into the runtime performance, we also ran our L1, L2, and L3 solutions to Problem A and our L1 solution to Problem B for 32 runs alongside the participants' solutions. In Table IV we see that our L1 solution to Problem A was **13%** faster and **16%** slower than participants' Copilot-aided and Copilot-unaided solutions respectively. Our L2 solution to Problem A was **129%** and **110%** faster than participants' Copilot-aided and Copilot-unaided solutions, respectively. Our L3 solution to Problem A was **147%** and **132%** faster than participants' Copilot-aided and Copilot-unaided solutions, respectively.

Similarly, our L1 solution to Problem B was **106%** and **95%** faster than participants' Copilot-aided and Copilot-unaided solutions. We did not run our L0 solutions because the participants already implement L0 solutions for both problems

Problem	*Level*	*Mean*
A	L1	30.59 s
A	L2	7.565 s
A	L3	5.228 s
B	L1	581.4 ms

TABLE IV: Model Solutions Runtime Performance

3) Discussion: Our results suggest that **developers may benefit from Copilot-unaided C++ code in terms of runtime performance**. We give further context to these results by highlighting some participants' Copilot-unaided solutions whose mean runtime performance was close to or better than the model solutions highlighted in Section IV-A and Section IV-B.

Problem A. While our model L3 solution had a mean runtime of 5.288 s, P31's noteworthy Copilot-unaided solution had a mean runtime of 4.547 s beating our best model solution by **15%**. Their solution is shown in Listing 1.

We note that their solution used the L3 optimization for Problem A discussed in Section IV-A4. Additionally, in lines 4 - 7 a map was used to associate each `fileId` with a vector of `fileCombos` for the associated file. The pre-processing in this step allowed them to sort each vector of `fileCombos` belonging to a file (line 9), open the file once (lines 11 - 12), process all the `fileCombos` (lines 13 - 16) and then close the file (line 17). While the fundamental concept of the L3 optimization is still present, some implementation details are slightly different and as such may have contributed to the observed speed-up.

It is also pertinent to mention that P31 had ideas to add other optimizations that could have potentially reduced the runtime performance of their code even further. However, they did not have sufficient time to do so and debug their solution. They outlined this optimization in code comments which have been removed from the Listing for clarity. The potential improvement involved the usage of `memcpy` [47] "to avoid overlaps".

```
1  bool compareByOffset(const FileCombo* a, const FileCombo* b) { return (a->offset < b->offset); }
2
3  void readFileCombos(std::vector<FileCombo> &fileCombos) {
4    std::map<int, std::vector<FileCombo*>> combosByFile;
5    for (FileCombo& combo : fileCombos) {
6      combosByFile[combo.fileId].push_back(&combo);
7    }
8    for (auto combos : combosByFile) {
9      std::sort(combos.second.begin(), combos.second.end(), compareByOffset);
10     int previousOffset = 0-RECORD_SIZE-1;
11     std::ifstream in;
12     in.open(FILE_NAMES[combos.first]);
13     for (FileCombo* combo : combos.second) {
14       in.seekg(combo->offset);
15       in.read(combo->buffer, RECORD_SIZE);
16     }
17     in.close();
18   }
19 }
```

Listing 1: P31's L3 Solution to Problem A without Copilot

Problem B. A noteworthy solution to Problem B was P17's Copilot-unaided solution (in Appendix F [37]). This resembled the model L1 solution with some statement-level optimizations explained in Section VII-C and was one of the closest-performing solutions to our L1. Their solution had a mean runtime performance of 636.4 ms which was only 9% slower compared to our model L1 solution, which had a mean runtime performance of 581.4 ms.

C. RQ2 - Do Copilot's suggestions sway developers towards or away from C++ code with faster runtime performance?

1) Approach: We wanted to understand how suggestions from Copilot swayed participants to produce code with slower or faster runtime performance. To this end, we took the last snapshot of the participants' submitted code and categorized each participant's code for problems A and B. We labelled participants' code according to the optimizations discussed in Section IV.

An author of this work and a collaborator separately looked through the source code for all participants and labelled each solution for Problem A as either L0, L1, L2, or L3 to indicate the levels of optimizations that participants used. Similarly, for Problem B, they were labelled as L0 or L1. Additionally, they also noted programming constructs that participants used that could potentially increase or decrease the runtime performance and tried to group similar constructs.

We term these "programming constructs" as statement-level optimizations and refer to the optimizations within Section IV as concept-level optimizations from this point.

2) Statement Level Optimizations & Open-coding: After the author and the collaborator finished labelling participants' source files with concept-level and statement-level optimizations, they came together to resolve disagreements and discuss emerging patterns in the statement-level optimizations and remarks. Upon resolving the disagreements, they came up with a set of themes to encompass the statement-level optimizations. A summary of these themes/categories of statement-level optimizations for Problem A and Problem B are in Table V and Table VI respectively.

3) Video Analysis: Using the themes generated in Table V and Table VI, the author went through all 32 screen-shared recordings of participants solving the problem when Copilot was used and tracked the accepted suggestions or series of accepted suggestions that participants accepted that swayed them to the solutions that fit their themes.

4) Results: For Problem A, where Copilot was used, 15 of the 16 correct solutions used the L0 naive implementation with the `<fstream>` [48] family of library functions and thus were categorized as L0F. Additionally, few remarks were made as most solutions only used the naive L0F implementation in IV-A1. Some solutions were remarked as NCLOSE because they failed to close the files after reading from them. Some solutions also landed in the BINARY category. From the video analysis, it would seem that Copilot largely gave L0F suggestions, and participants simply accepted them without editing. Participants also only confirmed that the sanity checks passed before declaring they were done with the problem.

In Problem B, we notice a relatively balanced use of concept-level optimizations and varied use of statement-level optimizations and remarks when using Copilot. From 14 (out of 16) source snippets with correct solutions, we note that 9 of those solutions used the L1 concept-level optimizations of avoiding false sharing. Notably, 1 of the 9 (P23) was classified as L1 because it avoided false sharing by using OpenMP to handle the multi-threaded execution. 2 of the 14 solutions were encoded as L0 even though false sharing was absent because they either used a single-threaded approach (P7) or used only one additional thread (P3) for the problem instead of `THREAD_COUNT` threads. 3 of the 14 (P4, P11 and P19) solutions were encoded as L0 because false sharing was present in their solutions. Additionally, statement-level remarks such as 2LOOPS or 1LOOP were prevalent in the solutions.

Moreover, ITER_NAIVE and ITER_FAST were also common categories that emerged. Rarer categories like OPENMP, ONET and NT also appeared in a few cases. From the video analysis, Copilot initially suggested incomplete snippets leaning toward L0. Participants would accept the snippets and try to get the rest of the solution to work by debugging. In other cases, participants wrote comments about dividing an array into `THREAD_COUNT` chunks, and Copilot would suggest snippets leaning towards L1.

5) Discussion: For Problem A with Copilot, there was an interesting case where P22 was swayed via Copilot's suggestions to use L1U (Level 1 optimization but using the `<unistd.h>` [49] and `<fcntl.h>` [50] I/O functions). From the video analysis, we observe that the participant was largely responsible for coming up with concept-level L1 optimization in that they only declared a vector of file descriptors before the suggestions to use L1U with NCLOSE came along, which the participant accepted. However, P22 remarked that they "had to do more post-hoc checking" instead of "figuring out how to solve the problems"; that it was "a different approach of how they would solve the problem". We also note that while their solution used the L1 concept-level optimization, the mean runtime for their solution was **35.48 s** which was **15%** slower than our model L1 solution. This difference may be due to differences in the I/O implementation details in the `<unistd.h>` and the `<fcntl.h>` libraries versus the `<fstream>` [48] library. A snippet of P22's Copilot-aided solution to Problem A can be found in Appendix F [37].

Within Copilot-aided solutions to Problem B, we noticed that the solution with the least mean runtime performance at **677.8 ms** was from P12, who used the L1 concept-level optimization, and landed in the 1LOOP and ITER_LESS_NAIVE themes for the statement-level remarks. From the video analysis, the initial incomplete solutions accepted by the participant were leaning towards 1LOOP, NT and the incorrect solution of MISSING_LOOP. P12 was primarily responsible for implementing the code in the ITER_LESS_NAIVE statement-level remark because they "didn't think Copilot understood them[me] well when they[I] told it to increment or decrement" and "just gave up and wrote it themself[myself]". However, the L1 suggestion to split the thread into slices was accepted by the participant without much editing. P12 also remarked that "Copilot was useful", and they "usually just google" what Copilot would have suggested. We also note that their solution was 16% slower than our model L1 solution which could be because the model L1 solution used ITER_FAST and 1LOOP statement level optimizations. See a snippet of P12's solutions to Problem B that was done with Copilot in Appendix F [37].

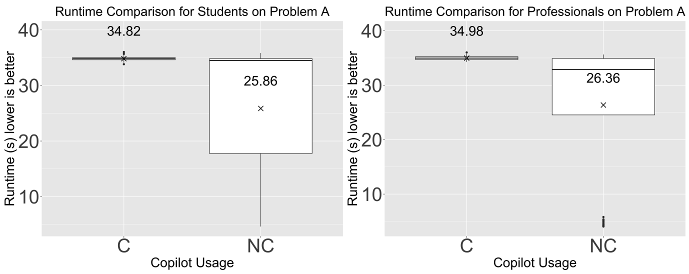
Some interesting categories for statement level optimizations in Problem B in Table VI are worth taking a closer look at, notably, 2LOOPS, 1LOOP and ITER_NAIVE and ITER_FAST. Our model L1 solution uses 1LOOP, ITER_FAST and also avoids false sharing and averages at a mean of **581.4 ms**. The closest Copilot-aided solution to the model solution in terms of runtime performance was P12's (Appendix F [37]) with a mean runtime performance of **677.8 ms**. At a close second was P24 (Appendix F [37]) with a mean runtime performance of **784.0 ms**, which avoided false sharing and used 1LOOP and ITER_NAIVE. This difference in runtime performance between the model L1 solution and P24's suggests that using ITER_FAST is better than using ITER_NAIVE to update the source and destination buffers when false sharing is avoided. If we also look at P27's Copilot-aided solution to Problem B (See Appendix F [37]), we notice that while it avoids false-sharing, it uses 2LOOPS and ITER_NAIVE which earns it a mean runtime performance of **925.1 ms**. Comparing P24's with P27's solution suggests that using 2LOOPS instead of 1LOOP to update the source and destination buffers when false sharing is avoided could result in slower runtime performance. On the other hand, if we look at solutions where false sharing was used, we note that both P11's (See Appendix F [37]) and P19's (See Appendix F [37]) Copilot-aided solutions had false sharing present. However, their solutions used 2LOOPS with ITER_NAIVE with a mean runtime performance of **1434 ms** and 1LOOP with ITER_NAIVE with a mean running time of **6202 ms**, respectively. This difference in runtime performance may suggest that using 1LOOP instead of 2LOOPS could result in slower runtime performance when false sharing is present, which is different from when false sharing is absent, as with P24's and P27's solutions.

VIII. Discussion

A. Comparison of results between students and professionals

To see if students and professionals who took part in our study had different outcomes, we split the data we had between students and professionals. Then we compare the runtime performance of the solutions written by students and professionals with and without Copilot for problems A and B. We compare the results using box plots and carry out the non-parametric Wilcoxon rank sum test for statistical significance.

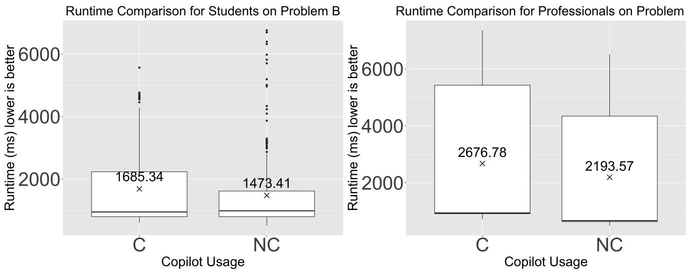
Problem A: From Figure 4 we can see the comparative run time performance of the solutions from students and professionals in Problem A. In both Figures. 4a and 4b we can see that the runtime performance of solutions when both students and professionals don't use Copilot is faster than when they use Copilot. The difference in the mean runtimes is approximately 8-9 seconds slower when using Copilot. When we test the data using the non-parametric Wilcoxon rank sum test, we find that the results are statistically significant (p = 4.99e-22 and p = 2.76e-15 for students and professionals respectively). Hence we can conclude that in Problem A, when taken as a whole and separately (as students and professionals), we consistently get the result that C++ code written with Copilot is slower than C++ code written without Copilot.



(a) Students

(b) Professionals

Fig. 4: Box Plots of runtimes for solving Problem A with Copilot (C) and Without Copilot (NC)



(a) Students

(b) Professionals

Fig. 5: Box Plots of runtimes for solving Problem B with Copilot (C) and Without Copilot (NC)

Problem B: In Figure 5, we can boxplots of the runtimes between students and professionals when they solve problem B with and without Copilot. We can again see that the mean runtimes when using Copilot is slower than when not using Copilot for both students (Figure 5a) and professionals (Figure 5b). The difference in means is approximately 0.2-0.5 seconds. When we test the data using the non-parametric Wilcoxon rank sum test, we find that the results are statistically significant for professionals (p = 2.61e-9) and not for students (p = 0.049). Therefore, while we cannot statistically conclusively say that the runtimes are slower for both students and professionals when considered separately, we see that the relationship between using and not using Copilot for Problem B is consistent with the data as a whole.

From Figures 4 and 5 we can conclude that our results from Section VII-B still hold when we consider students and professionals separately for both problems A and B.

B. Familiarity with the problem

Two of the questions we asked participants in the post-survey were whether they had previously seen the problem that they solved, and if they had solved it when previously seen. Out of the 64 combinations (32 participants each solving 2

Encoding	Summary
L*F	Used `<fstream>` [48] library for any of the concept-level optimizations L0, L1, L2, or L3
L*C	Used `<cstdio>` [51] library for any of the concept-level optimizations L0, L1, L2, or L3
L*U	Used `<unistd.h>` [49] and `<fcntl.h>` [50] libraries for any of the concept-level optimizations L0, L1, L2, or L3
NCLOSE	Did not close file
EXCEPT	Added `file.exceptions(...)` [52] to catch possible exceptions
ASSERT	Asserted that no error flags were set after file operations using `good()` [53] method and other assertions to ensure program correctness
READ_COMBO	Helper function for processing a single `fileCombo` in `fileCombos` and by calling `open()`, `seek()`, `read()`, and `close()` in order
BEGIN	Explicit seek from `std::ios_base::beg` [54] in call to `seekg()` [55]
OC_WITHIN	Opened and closed the files within the same loop as processing each `fileCombo` in `fileCombos`
BINARY	Added a ”binary” flag to the `open` call using `std::ios::binary` [56] or similar
MAP	Used a `map` [57] to associate a file with all the `fileCombos` associated with that file

TABLE V: Table of Statement-level Optimizations & remarks for Problem A

Encoding	Summary
NT	No threads used
ONET	Only one thread was used. Equivalent to not using threads
MISSING_LOOP	Failed to loop to decrement `src[i]` to zero and to increment `dst[i]` to `INIT_SRC_VAL`. This is an incorrect solution.
ITER_NAIVE	Made `SIZE X INIT_SRC_VAL` repeated calls to `dst[i].get()` or `src[i].get()` while decrementing `src[i]` and incrementing `dst[i]`
ITER_LESS_NAIVE	Made `SIZE` repeated calls to `src[i].get()` or `dst[i].get()` while decrementing `src[i]` and incrementing `dst[i]`
ITER_FAST	No calls to `src[i].get()` or `dst[i].get()` while decrementing `src[i]` and incrementing `dst[i]` but iterated up to `INIT_SRC_VAL`
2LOOPS	Decremented `src[i]` to 0 then incremented `dst[i]` to `INIT_SRC_VAL` instead of in lockstep
1LOOP	Decremented `src[i]` and incremented `dst[i]` in lockstep
SPLIT	`src[i]` is decremented using a separate thread and `dst[i]` is incremented using a separate thread
SPLIT2	Like SPLIT but `src[i]` decremented using 2 threads after being divided into 2 slices and `dst[i]` incremented using 2 threads after being divided into 2 slices
MANY_SPLIT	Spawned `SIZE` threads where each thread handled `src[i]` and `dst[i]`. There could be context switches since not enough threads on machine
LOCKS	Used locks.
RACET	Race conditions in thread spawning without locks leading to incorrect results
HARDT	Hardcoded thread spawning instead of dynamic based on `THREAD_COUNT`
PTHREAD	Used `pthread_create` and `pthread_join` [58] to create and join threads instead of `std::thread` methods
SPAWN_SEP	Spawned `THREAD_COUNT` threads to process `src[i]` then wait to finish then spawn another `THREAD_COUNT` threads for `dst[i]` then wait to finish
OPENMP	Used `parallel for` in OpenMP.

TABLE VI: Table of Statement-level Optimizations & remarks for Problem B

problems, one with Copilot and one without = 32x2), we found that only in 8 cases had participants either seen or solved the problem before (see Table VII. Therefore an overwhelming majority of them had neither seen the problem nor solved it before.

Problem	*Mode*	*Have Seen Problem*	*Have Solved Problem*	*Is a Professional*	*Runtime*	*Difference With Mean of the Setting (D)*
A	C	Yes	Yes	TRUE	35.01	0.03
A	NC	Yes	Maybe	FALSE	21.53	-4.33
A	NC	Yes	No	FALSE	34.95	9.09
B	C	Yes	Yes	FALSE	773.99	-911.35
B	C	Yes	No	FALSE	1433.83	-251.51
B	C	Yes	No	FALSE	3561.19	1875.85
B	C	No	Yes	FALSE	2245.17	559.83
B	NC	Yes	No	FALSE	1404.06	-69.35

TABLE VII: Participant familiarity with the problem and the solution. Column D at the end notes the difference between the runtime of the solution from a participant with the mean of the runtimes from all participants. A negative value indicates that the participant had a faster solution than the mean.

In addition from Table VII we see that in 4 cases participants had a faster solution and in 4 cases they had slower solutions compared to the mean runtime for that setting (Problem x Mode). Interestingly, in problem A we have a professional who has both seen and solved the problem before. They used Copilot to solve Problem A in our experiments and produced a solution with a slightly slower runtime than the mean runtimes when using Copilot to solve Problem A. This indicates that even when people have solved the problems, Copilot may lead them to a slower-than-average solution. On the other hand, students using Copilot to solve Problem B were evenly split. Two of them had a faster solution and two had a slower solution compared to the mean. And when the student in the last row on Table VII solved Problem B without Copilot, they had a faster runtime than the mean.

From the results in this analysis, we can see that (a) our experiments were not biased much with results from people who had seen or solved the problem before, and (b) even when they have, the results indicate that using Copilot may result in solutions with slower performance than when not.

IX. Threats to Validity

A. *External Validity*

Programming Language and Code Generation Tool. This study is explicitly about the runtime performance of C++ code written with the help of Copilot. We specifically chose C++ as a programming language for the experiment as C++ applications are typically performance-critical. We also specifically chose Copilot as it is used by more than a million developers [59]. We need more studies to examine runtime performance in other settings - different programming languages and different LLM-based code generation tools.

Number of Problems: We restrict our study to two problems as increasing the number of problems increases the number of participants required exponentially in our controlled experiment. To maintain the same experimental design we would need 384 participants for 4 problems and 11520 participants for 6 problems. The effort to run the experiment with more than 12-360 times the current number of participants is exponential in many ways: time to find participants, run the experiment, pay the participants, and analyze the data from the experiment. We do not know of any software engineering research study with a controlled experiment where there were more than 300 participants. There have been survey-based studies with more than 300 participants, but surveys are not high in effort like a controlled experiment. A recent paper using LLMs for code understanding also has 32 participants and 2 tasks [60].

Choice of Problems: We acknowledge the limitations of the representativeness of the programming problems used for this study. While file system operations and multi-threading programming concepts are critical, there could be other important domains that are not represented in our study that developers could have been more aware of. However, we argue that our chosen problem domains are typically part of the training of software engineers making it worth examining.

B. Construct Validity

Not explicitly asking participants to write low runtime performance code: Although it might seem that explicitly directing participants to produce performant code with or without Copilot, we argue that this directive would be part of a different study. This future study would answer the question of how well Copilot can generate highly performant code compared to human developers. Typically while run time performance is desired, we have not seen a case where every requirement in a software is explicitly asked to have lower runtime performance. Hence, our study explores whether code written without any explicit requirement for performance is different when using Copilot. Another possible study is when compared to developers with no performance experience, can directed prompts in Copilot produce higher-performance code. The participant pool, experiment designs and analysis of all of these studies are quite different. We leave these alternate studies as future work that needs to be studied too and argue that our study and its results are valuable too.

Choice of participants: We acknowledge that 75% of our participants are graduate students. We intended to have more than our current set of professionals (25%). However, as we state in Section V-A, we faced difficulty in recruiting professionals, especially systems programmers. However, all the graduate students who did participate are systems researchers who have extensive C++ and developer experience. Additionally, we split the results for students and professionals, and found that the findings remained the same - students and professionals wrote code that on average had a faster runtime performance when not using Copilot in comparison with using Copilot for both problems A and B. We have included the tables in our replication dataset online [37].

Finally, we acknowledge that there could have been some participant selection bias as we only selected participants that wanted to participate in the study. Our results may have been slightly different if the developers that were unwilling to participate actually took part in the study. A workaround to recruit such developers unwilling to participate due to negative perceptions about GitHub Copilot would be to omit details about using it until the session actually began. However, as there are significant ethical concerns with this type of deception in controlled human studies, this was infeasible.

C. Internal validity

As we were looking at the runtime performance of participants' code, another possible limitation could have been that participants did not have enough time to optimize their solution. However, on average all 32 participants spent approximately **17 minutes** of the 30 minutes allotted on problem A and **20 minutes** on problem B. Therefore the participants were satisfied with their solution at least 10 minutes before their time was up.

X. Conclusions and Takeaways

This work evaluated the performance of C++ code generated by the self-proclaimed AI programming assistant GitHub Copilot by conducting a user study on systems programmers. We present our main takeaways for different stakeholders.

- **Developers:** Developers must be careful about the code they get from Copilot, and not only review Copilot-generated code for functional correctness but also for non-functional aspects like runtime performance.
- **Maintainers:** Maintainers of Copilot need to evaluate and improve their tools and models to focus not only on functional correctness but also on other aspects of code that are just as important: performance, security, reliability, etc.
- **Researchers:** While benchmark datasets like HumanEval, MBPP, and SWE-Bench are available to evaluate the functional correctness, we need benchmark datasets for non-functional aspects too. With GitHub Copilot gaining ubiquity in modern software development, more research is required to scope its strengths and limitations.

XI. Data Availability

We provide the problems and prompts given to participants, the expert solutions we generated, the test scripts we used for evaluation, and the data set of the runtime performance of participants' solutions as well as the runtime performance of the model solutions [37]. However, to respect participant privacy and anonymity as well as to abide by the rules set by the ethics review board, we are unable to share participant responses to the screening survey, their video data, their responses to the programming surveys, and their entire unedited source code solutions. We share everything needed for anyone to be able to replicate this study with their own set of participants by ACM standards [61].

References

[1] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[2] F. F. Xu, B. Vasilescu, and G. Neubig, "In-IDE Code Generation from Natural Language: Promise and Challenges," *ACM Transactions on Software Engineering Methodologies*, vol. 31, no. 2, mar 2022. [Online]. Available: https://doi.org/10.1145/3487569

[3] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[4] (2023) About GitHub Copilot. [Online]. Available: https://docs.github.com/en/copilot/overview-of-github-copilot/about-github-copilot

[5] (2023) Hype or not? AI's benefits for developers explored in the 2023 Developer Survey. [Online]. Available: https://stackoverflow.blog/2023/06/14/hype-or-not-developers-have-something-to-say-about-ai

[6] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. Experience: Evaluating the Usability of Code Generation Tools powered by Large Language Models," in *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3491101.3519665

[7] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "An empirical cybersecurity evaluation of GitHub CoPilot's code contributions," *CoRR*, vol. abs/2108.09293, 2021. [Online]. Available: https://arxiv.org/abs/2108.09293

[8] Google, "Mobile page speed and industry benchmarks," https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/, 2017.

[9] Gigaspaces, "Amazon Found Every 100ms of Latency Cost them 1% in Sales," https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales, 2023.

[10] X. Tian, R. Han, L. Wang, G. Lu, and J. Zhan, "Latency critical big data computing in finance," *The Journal of Finance and Data Science*, vol. 1, no. 1, pp. 33–41, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405918815000045

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[13] B. Yetistiren, I. Ozsoy, and E. Tuzun, "Assessing the Quality of GitHub CoPilot's Code Generation," in *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 62–71. [Online]. Available: https://doi.org/10.1145/3558489.3559072

[14] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," 2021.

[15] J.-B. Döderlein, M. Acher, D. E. Khelladi, and B. Combemale, "Piloting CoPilot and Codex: Hot Temperature, Cold Prompts, or Black Magic?" 2023.

[16] A. Elnashar, M. Moundas, D. C. Schmidt, J. Spencer-Smith, and J. White, "Evaluating the Performance of LLM-Generated Code for ChatGPT-4 and AutoGen Along with Top-Rated Human Solutions."

[17] A. Mastropaolo, L. Pascarella, E. Guglielmi, M. Ciniselli, S. Scalabrino, R. Oliveto, and G. Bavota, "On the Robustness of Code Generation Techniques: An Empirical Study on GitHub CoPilot," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 2149–2160.

[18] N. Nascimento, P. Alencar, and D. Cowan, "Comparing software developers with ChatGPT: An Empirical Investigation," *arXiv preprint arXiv:2305.11837*, 2023.

[19] OpenAI, "GPT-4 Technical Report," 2023.

[20] G. Sandoval, H. Pearce, T. Nys, R. Karri, B. Dolan-Gavitt, and S. Garg, "Security implications of large language model code assistants: A user study," *arXiv preprint arXiv:2208.09727*, 2022.

[21] O. Asare, M. Nagappan, and N. Asokan, "Is Github's Copilot as bad as humans at introducing vulnerabilities in code?" *Empirical Software Engineering*, vol. 28, no. 6, p. 129, 2023.

[22] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 508–512. [Online]. Available: https://doi.org/10.1145/3379597.3387501

[23] L. Traini, D. Di Pompeo, M. Tucci, B. Lin, S. Scalabrino, G. Bavota, M. Lanza, R. Oliveto, and V. Cortellessa, "How Software Refactoring Impacts Execution Time," *ACM Transactions on Software Engineering Methodologies*, vol. 31, no. 2, Dec 2021. [Online]. Available: https://doi.org/10.1145/3485136

[24] D. G. Reichelt, S. Kühne, and W. Hasselbring, "PeASS: A Tool for Identifying Performance Changes at Code Level," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1146–1149.

[25] I. Drori and N. Verma, "Solving Linear Algebra by Program Synthesis," 2021.

[26] L. Tang, E. Ke, N. Singh, N. Verma, and I. Drori, "Solving Probability and Statistics Problems by Program Synthesis," 2021.

[27] A. Moradi Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. J. Jiang, "GitHub CoPilot AI pair programmer: Asset or Liability?" *Journal of Systems and Software*, vol. 203, p. 111734, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121223001292

[28] S. Imai, "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*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 319–321. [Online]. Available: https://doi.org/10.1145/3510454.3522684

[29] N. Nguyen and S. Nadi, "An Empirical Evaluation of Github Copilot's Code Suggestions," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 1–5.

[30] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le, and D. Lo, "Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues," 2023.

[31] R. Choudhuri, D. Liu, I. Steinmacher, M. Gerosa, and A. Sarma, "How Far Are We? The Triumphs and Trials of Generative AI in Learning Software Engineering," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639201

[32] D. Sobania, M. Briesch, and F. Rothlauf, "Choose Your Programming Copilot: A Comparison of the Program Synthesis Performance of GitHub CoPilot and Genetic Programming," ser. GECCO '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1019–1027. [Online]. Available: https://doi.org/10.1145/3512290.3528700

[33] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller, "Measurement and Analysis of Large-Scale Network File System Workloads," in *USENIX 2008 Annual Technical Conference*, ser. ATC'08. USA: USENIX Association, 2008, p. 213–226.

[34] Y. Son, H. Y. Yeom, and H. Han, "Optimizing I/O Operations in File Systems for Fast Storage Devices," *IEEE Transactions on Computers*, vol. 66, no. 6, pp. 1071–1084, 2017.

[35] S. Manakkadu and S. Dutta, "Bandwidth Based Performance Optimization of Multi-threaded Applications," in *2014 Sixth International Symposium on Parallel Architectures, Algorithms and Programming*, 2014, pp. 118–122.

[36] (2019) Common Systems Programming Optimizations & Tricks. [Online]. Available: https://paulcavallaro.com/blog/common-systems-programming-optimizations-tricks/

[37] Anonymous, “Supplementary material, Runtime performance data, and Reproducibility script,” Jul. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.13139261

[38] (2023) Discord. [Online]. Available: https://discord.com/

[39] (2016) Internet Relay Chat (IRC). [Online]. Available: https://www.irchelp.org/

[40] (2022) Serenity OS. [Online]. Available: https://serenityos.org/

[41] (2021) Skift OS. [Online]. Available: https://skiftos.org/

[42] (2022) Operating Systems Development. [Online]. Available: https://wiki.osdev.org/Expanded_Main_Page

[43] (2022) Vim - visual studio marketplace. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=vscodevim.vim

[44] (2023) Visual Studio Code Remote - SSH. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.remote-ssh

[45] (2020) gcc-9 9.3.0-17ubuntu1 20.04 source package in ubuntu. [Online]. Available: https://launchpad.net/ubuntu/+source/gcc-9/9.3.0-17ubuntu1~20.04

[46] (2023) Wilcoxon Rank Sum and Signed Rank Tests. [Online]. Available: https://stat.ethz.ch/R-manual/R-devel/library/stats/html/wilcox.test.html

[47] (2022) memcpy - Copy block of memory. [Online]. Available: https://cplusplus.com/reference/cstring/memcpy/

[48] (2022) fstream - Input/output file stream class. [Online]. Available: https://cplusplus.com/reference/fstream/fstream/

[49] (1997) unistd.h - standard symbolic constants and types. [Online]. Available: https://pubs.opengroup.org/onlinepubs/7908799/xsh/unistd.h.html

[50] (1997) fcntl.h - file control options. [Online]. Available: https://pubs.opengroup.org/onlinepubs/7908799/xsh/fcntl.h.html

[51] (2022) cstdio - C library to perform Input/Output operations. [Online]. Available: https://cplusplus.com/reference/cstdio/

[52] (1997) std::ios::exceptions. [Online]. Available: https://cplusplus.com/reference/ios/ios/exceptions/

[53] (2022) std::ios::good. [Online]. Available: https://cplusplus.com/reference/ios/ios/good/

[54] (2021) std::ios_base::seekdir. [Online]. Available: https://en.cppreference.com/w/cpp/io/ios_base/seekdir

[55] (2022) std::istream::seekg. [Online]. Available: https://cplusplus.com/reference/istream/istream/seekg/

[56] (2022) std::ios_base::openmode. [Online]. Available: https://en.cppreference.com/w/cpp/io/ios_base/openmode

[57] (2022) std::map. [Online]. Available: https://cplusplus.com/reference/map/map/

[58] (2021) pthreads(7) — Linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man7/pthreads.7.html

[59] (2023) The economic impact of the AI-powered developer lifecycle and lessons from GitHub CoPilot. [Online]. Available: https://tinyurl.com/copilot-marketshare

[60] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, “Using an LLM to help with code understanding,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[61] (2020) ACM badging terms. [Online]. Available: https://www.acm.org/publications/badging-terms