Measuring the Runtime Performance of Code Produced with GitHub Copilot

Daniel Erhabor
derhabor@uwaterloo.ca
University of Waterloo
Canada

Sreeharsha Udayashankar
s2udayas@uwaterloo.ca
University of Waterloo
Canada

Meiyappan Nagappan
m2nagapp@uwaterloo.ca
University of Waterloo
Canada

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

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 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 code with a significantly slower runtime performance.

1 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 [3] have been extended to what Xu et al. [35] have termed Natural-Language-to-Code (NL2Code) generators. Notably, Open AI's extension of the GPT-3 language model, Codex [4], and the production-ready product derived from it, GitHub Copilot [1], are popular examples of NL2Code tools in use today. While some studies have shown that developers generally may have a positive experience using GitHub Copilot [31], others have shown that it could generate potentially vulnerable code [18].

We present the first-ever evaluation of Copilot from a runtime performance perspective in systems programming. We conducted the first user study on Copilot to evaluate the runtime performance of the code generated when developers use it. With the results from our study, we hope to answer the following research questions:

RQ0: Does using Copilot influence program correctness?
RQ1: Is there a runtime performance difference in code when using GitHub Copilot?
RQ2: Do Copilot's suggestions sway developers towards or away from code with faster runtime performance?
RQ3: Do characteristics of Copilot users influence the runtime performance when it is used?

To answer our research questions, we conducted a user study with 32 system programmers. Each participant solved two programming problems in C++, one problem was solved with Copilot and the other was solved without it. One problem was related to I/O operations and one was related to concurrent programming. We selected problems related to these two domains as they have direct impact on the code runtime. We compared the runtime performance of Copilot-aided solutions with Copilot-unaided solutions, obtained survey responses from participants after they completed the problems, and analysed the video recordings of participants solving the problems.

Our findings indicate that using Copilot resulted in code with statistically significantly slower runtime performance. Specifically, Copilot-unaided solutions were 26% faster than Copilot-aided solutions for the problem related to I/O and 15% faster for the problem related to concurrent programming on average. In general, our expert solutions to the problems had up to 6 times faster runtime performance compared to the average of Copilot-aided solutions. Additionally, Copilot-aided solutions tended to tilt developers to code with slower runtime performance. Further, higher developer experience and familiarity with the C++ programming language were factors associated with code that had faster runtime performance as expected.

The paper is organized in the following way: We briefly go over some background related to GitHub Copilot and some related work in Section 2. The process of creating the problems that the participants would solve and the rationale behind choosing the problems is described in Section 3. Our model solutions to the problems are elaborated in Section 4 giving context to the problems. A summary of the participant recruitment process and the participants are described in Section 5. We then present the experiment design in detail in Section 6 where we cover the tasks that participants solved, how the tasks were split across the participants and the rationale behind it. Penultimately, we analyze and discuss the experiment's results, answering our research questions in Section 7. An overview of the methodology can be seen in Figure 1. Finally, in Section 8, we talk about the takeaways and limitations of our study and potential future directions.

2 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 when the extension is activated. In this way, users can receive suggestions by starting to write the code or by writing comments; either way, Copilot will suggest some snippets [1]. See Figure 2 for an example of Copilot in action.

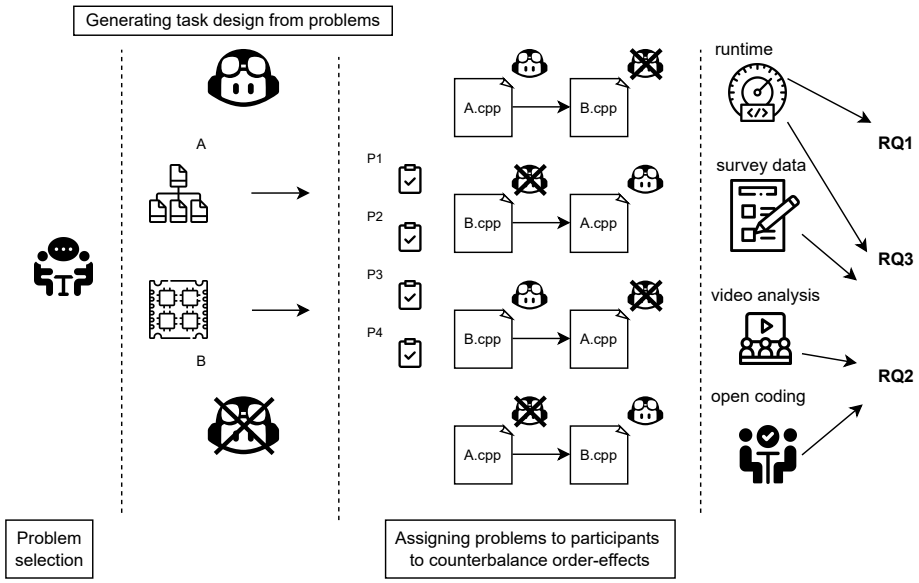

Figure 1: Overview of Methodology

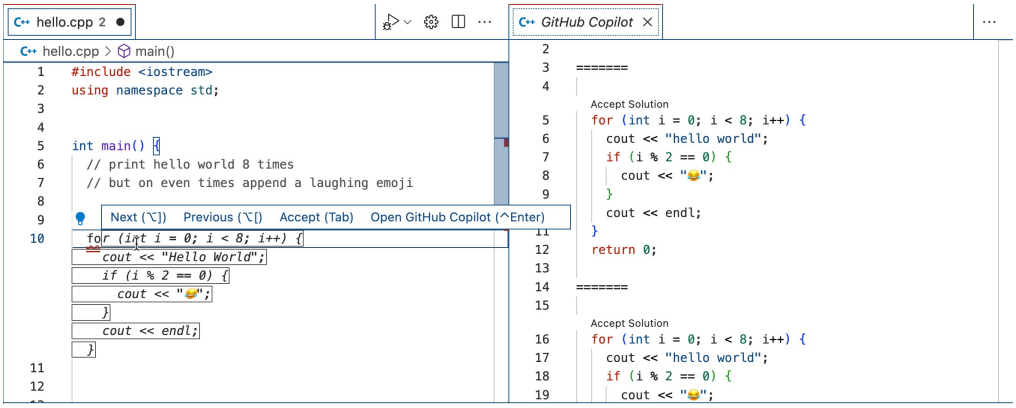

Figure 2: Copilot in Action

One of the early studies on Copilot was by Pearce et al. [18], where they wanted to understand how often suggestions from Copilot were vulnerable to security attacks and the contexts that made Copilot suggest vulnerable code. To achieve this, they prompted Copilot to suggest code in scenarios where the resultant suggestions could have been vulnerable or secure. Of the programs produced in response to the potentially vulnerable scenarios, 40% were vulnerable. In our work, we observe developers using Copilot instead of us generating code with Copilot.

A study by Sandoval et al. [20] in collaboration with Pearce from [18] assess 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 their previous study [18], their results did not show that Copilot had a conclusive impact on security. In contrast, our study has participants use the production-ready tool [1] as their NL2Code assistant instead of creating a custom environment that uses a custom model.

Vaithilingam et al. [31] conducted a user study on 24 participants to understand how programmers perceive and use Copilot; they found that programmers preferred to use Copilot in their day-to-day programming tasks and found it helpful as a starting point. In our study, the programming language was restricted to C++ and our participant pool was restricted to systems programmers. Additionally, a major difference compared to the previous three

studies is that we focus on the runtime performance implications of using Copilot.

3 PROGRAMMING PROBLEMS SOLVED BY PARTICIPANTS

Following in the same vein as Pearson et al. [18], we provided an "incomplete" code for participants to implement as a solution to a given problem. By "incomplete", we mean that we provided code stubs and accompanying documentation for the stubs that participants were asked to implement during the study. We call the stubs "problems" throughout this paper. These problems were provided to participants in the form of a CPP file that contained 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.

3.1 Problem selection

We chose two problem domains for our programming problems, file-system operations and multithreaded programming. We chose these two areas because problems in those domains tend to have a direct impact on application runtime performance. With file I/O operations accounting for about 30% - 80% of interactions in networked file systems [14], there is a need for file system operations to be fast on storage devices [23]. 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 multithreaded applications [15] and optimize accordingly. To reflect this, we chose a problem related to false sharing, a typical multi-threading optimization problem [5].

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.

3.2 Problem A: File System Operations

For this problem, participants were asked to read records from three text files. Each file is 1GB in size. The read operation is specified by `FileCombo`, a struct which 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.

A record is a sequence of 5000 bytes. For this problem, participants received a CPP file for problem A and three large text files. The full function signatures and the entirety of the CPP file with the accompanying documentation for problem A given to participants are in Appendix A.1 [8].

3.3 Problem B: Multi-threaded Optimization

For this problem, participants were asked to use a certain amount 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 repeatedly access and modify array items, potentially experiencing false sharing.

The full function signatures and the entirety of the CPP file with the accompanying documentation for problem B given to participants are in Appendix A.2 [8].

4 MODEL SOLUTIONS TO THE PROBLEMS

We created what we term "model" solutions to the problems. 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 corresponding to faster runtime performance than lower levels 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 [8].

4.1 Problem A Solutions

Level 0. A naive solution to Problem A where calls to `open`, `seek`, `read`, and `close` are made for each `fileCombo` in `fileCombos`.

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` in `fileCombos`. This optimization involves opening all the files in `FILE_NAMES` first and closing them only after each `fileCombo` in `fileCombos` has been processed. This avoids the repeated opening and closing of file descriptors, which is detrimental to runtime performance.

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.

Level 3. The combination of the L1 and L2 optimizations we outlined in Section 4.1 gives us the L3 optimization level, representing the best model solution to Problem A.

4.2 Problem B Solutions

Level 0. Consider a solution to Problem B using `THREAD_COUNT` concurrent threads. A naive solution to this problem is one where all threads starts at indices between 0–`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 line when modifying the `Item` within the `src` and `dst` arrays, leading to *cache thrashing*.

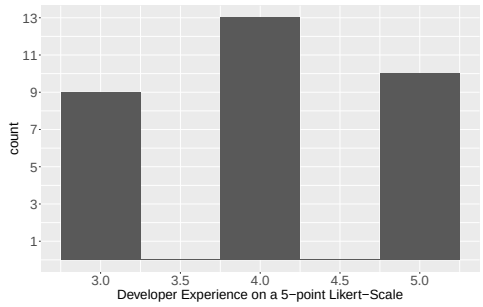

Figure 3: Distribution of Participants' Developer Experience from Screening Survey on a 5-point Likert-Scale from 1 (No experience) to 5 (10 years or more).

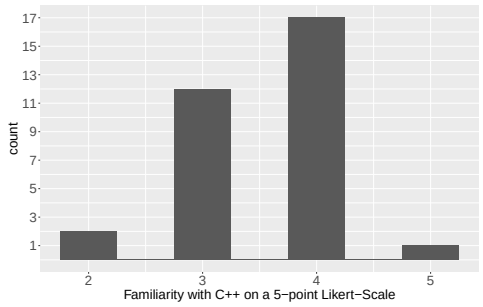

Figure 4: Distribution of Participants' Familiarity with C++ from Screening Survey on a 5-point Likert-Scale from 1 (Not familiar at all) to 5 (Extremely familiar).

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*.

Another solution to false sharing would be to add *padding* within the `Item` struct definition (See Appendix A.2 [8]), 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.

5 PARTICIPANTS

5.1 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++, have access to an internet browser and GitHub Copilot on Visual Studio Code at the time. Finally, participants could not be employed by OpenAI or 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 had read and signed the consent form declaring their intent to participate. The screening survey can be found in Appendix C [8].

5.2 Difficulties Recruiting Professionals

At the halfway point of our desired participant goal, we paused participant recruitment to analyze the preliminary data collected. Upon closer examination of participants' solutions to problem A, we observed that not a single participant had implemented any of the three levels of optimizations outlined in Section 4.1.

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 process of attempting to recruit professional systems developers started with contacting alumni of the affiliated university who were 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 only include projects with a dedicated social platform where interested parties connect, such as Discord [7] and Internet Relay Chat (IRC) [12].

While projects such as SerenityOS [21] and SkiftOS [22] had active Discord communities, there was a lack of interested potential participants in the study. Attempts to garner interest in the study within these projects were met with suggestions to reach out to other Discord communities such as the osdev (Operating Systems Development) [17] 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 reason for their unwillingness to participate in the study.

However, our persistent recruitment efforts paid off eventually, as we located professionals willing to participate in our study, enabling us to meet our desired participant goal.

5.3 Participant Summary

We recruited a total of 32 participants for this study, of which 25% were professionals in systems programming or contributors to open-source systems projects. Of the remaining participants, one

was a sessional lecturer with systems experience at the affiliated university, while the rest were graduate students with a systems research focus.

Figures 3 and 4 show the distribution of participants' experience and their familiarity with C++. Further details about the figures can be found in Appendix C [8]. Participants were compensated $50 for their time and the study was approved by Research Ethics Board (**REB# 44162**) at the affiliated university.

6 EXPERIMENT DESIGN

6.1 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 1.

The orders in Table 1 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 1: Possible Orders of *Mode* x *Problem*

6.2 Session Overview

We attempt to outline the steps carried out within each session in this section. Further details about the tutorial process can be found in Appendix D [8].

6.2.1 Pre-session orientation. The session was carried out remotely via an online conferencing platform. Each session began with the facilitator introducing the study and confirming the participant's consent to participate. Following this, the participant's consent was obtained to record their screen and audio during the session. Finally, the facilitator gave the participant a few basic tips for using Copilot such as accepting and rejecting suggestions.

6.2.2 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 online conferencing platform's chat feature or Google Drive if technical issues occurred with file uploads over the conferencing platform.

The participant was asked to extract the contents of the archive but not open them until the facilitator gave them the signal to do so. 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 were disabled except for the Copilot extension.[1](2) the participant 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.

6.2.3 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.

6.2.4 After each problem. Once the participant declared that they were done with a problem (or the timer ran out), the experimenter stopped the timer and notified the participants. They were then instructed to compress their solution and send it back to the facilitator via the online 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 participant was 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 initial survey's questions. Details of the first and second surveys are outlined in Appendix E [8].

6.2.5 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.

7 EVALUATION

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 SO-DIMMs), and 256 GB NVMe flash storage. The machine was running Ubuntu 20.04 and the code was compiled with gcc version 9.3.0 [11].

In order to minimize the effect of small runtime performance variations, we ran each participant's code 32 times. The filesystem cache was cleared between each of the 32 runs for every participant's code.

If the participant's code did not compile/compiled but encountered runtime errors, it was not run or analyzed. For instance, one participant's code produced a segmentation fault error even though it compiled successfully. However, if the participants' code did compile and run 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.

[1]keybinding related extensions like VSCode Vim [32] and SSH-related extensions like Remote - SSH [33] were the only exceptions allowed

7.1 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 2 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 2: List of Invalid Runs

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

7.2 RQ1 - Is there a runtime performance difference in code when using GitHub Copilot?

7.2.1 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 [34] `wilcox_test()` to compare the runtime performance.

7.2.2 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.

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 3: Summary Statistics of 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 3 highlights the summary statistics of the runtime performance for participants' valid solutions to the problems.

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 for 32 runs alongside the participants' solutions for a fairer evaluation.

In Table 4 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 4: Model Solutions Runtime Performance

7.2.3 Discussion. Our results suggest that **developers may benefit from Copilot-unaided 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 4.1 and Section 4.2.

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 7.1.

We note that their solution used the L3 optimization for problem A discussed in Section 4.1. 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` [16] "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 7.1: 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 [8]). This resembled the model L1 solution with some statement-level optimizations explained in Section 7.3 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.

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

7.3.1 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 4.

The author of this work and a collaborator separately looked through the source code for all participants and labelled each solution for problem A with 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 4 as concept-level optimizations from this point.

7.3.2 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 5 and Table 6 respectively.

7.3.3 Video Analysis. Using the themes generated in Table 5 and Table 6, 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.

7.3.4 Results. For problem A, where Copilot was used, 15 of the 16 correct solutions used the L0 naive implementation with the `<fstream>` [10] 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 4.1. 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.

7.3.5 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>` [30] and `<fcntl.h>` [9] 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>` [10] library. A snippet of P22's Copilot-aided solution to problem A can be found in Appendix F [8].

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 [8].

Some interesting categories for statement level optimizations in problem B in Table 6 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 [8]) with a mean runtime performance of **677.8 ms**. At a close second was P24 (Appendix F [8]) 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 [8]), 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 [8]) and P19's (See Appendix F [8]) 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.

7.4 RQ3 - Do characteristics of Copilot users influence the running time when it is used?

7.4.1 Approach. In addition to RQ1, we wanted to see if certain participant characteristics influenced the runtime performance when using Copilot. To this end, we used data from the screening survey that determined participant eligibility, the post-programming survey for the first and second problem, demographics questions that were included as part of the second programming survey, the runtime performance data we obtained in 7.2, and the time participants spent on each problem (6.2.2).

We generated a Spearman correlation matrix to observe any possible relationships. A summary of the data used in the correlation matrix is in Table 7. The data we used for this research question was only for valid solutions, i.e., solutions where the runtime performance was obtained (See Section 7).

7.4.2 Results. We present the Spearman correlation matrix showing the correlation coefficients. The correlation between mode (if Copilot was used as defined in 7) and runtime performance was **0.46** for problem A and **0.09** for problem B meaning that **using Copilot is associated with slower runtime performance**. Additionally, the correlation between runtime performance and developer experience was **-0.39** for problem A and **-0.30** for problem B. Likewise, the correlation between runtime performance and C++ familiarity was **-0.12** for problem A and **-0.10** for problem A. These results may imply that **developers with more experience and more familiarity with C++ can produce code with faster runtime performance.**

7.4.3 Discussion. While we notice some positive correlation between using Copilot and runtime performance code for A, our results suggest that problem B may have a weaker correlation. This observation suggests that using Copilot may have had less of an effect on runtime performance for problem B than it did for problem A. To explain this, we look at the runtime performance like we did in 7.2; however, in this case, we only consider the averages of all 32 iterations per participant, i.e., P22's snippet would be run 32 times,

Encoding	Summary
L*F	Used `<fstream>` [10] library for any of the concept-level optimizations L0, L1, L2, or L3
L*C	Used `<cstdio>` [6] library for any of the concept-level optimizations L0, L1, L2, or L3
L*U	Used `<unistd.h>` [30] and `<fcntl.h>` [9] libraries for any of the concept-level optimizations L0, L1, L2, or L3
NCLOSE	Did not close file
EXCEPT	Added `file.exceptions(...)` [26] to catch possible exceptions
ASSERT	Asserted that no error flags were set after file operations using `good()` [27] 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` [25] in call to `seekg()` [28]
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` [24] or similar
MAP	Used a `map` [29] to associate a file with all the `fileCombos` associated with that file

Table 5: 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` [19] 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 6: Table of Statement-level Optimizations & remarks for Problem B

Data from participants' solving		
runtime	Average of the running time of the 32 runs for each participant's code obtained in RQ1	positive floating-point
mode	Using Copilot (1) or not using Copilot (0)	1 or 0
time	How long participants took to solve the problem obtained in 6.2.2 and 6.2.3	minutes
Data from the screening surveys		
devExp	How much programming experience do you have?	5-Point Likert-scale
familiarCPP	How familiar are you with the C++ programming language?	5-Point Likert-scale
Data from the first and second post-programming surveys		
understandProblem	How well did you understand the programming problem?	6-Point Likert-scale
confidentSolution	How confident are you in your solution to the programming problem?	6-Point Likert-scale
timeDebugging	How much time did you spend debugging your program (in minutes)?	number between 0 and 30
timeBrowser	How much time did you spend on your browser while solving the problem (in minutes)?	number between 0 and 30
Data from demographics questions included from the second post-programming survey		
familiarCopilot	How familiar are you with Github Copilot?	6-Point Likert-scale
familiarVSCode	How familiar are you with Visual Studio Code (VSCode)?	6-Point Likert-scale

Table 7: Table of Input Data to Correlation Matrix

but we collect the average runtime performance of all 32 iterations as a single data point.

For problem A with Copilot, the "mean of means" (n1 = 16) runtime performance for valid solutions was **34.86 s** and without Copilot (n2 = 14) was **26.02 s** (note that this is the same as what was reported in 7.2). When we compared the runtime performance of Copilot (n1 = 16) versus not using Copilot (n2 = 14) via the same test in 7.2, we find the results still to be statistically significant (p = 0.0172).

For problem B with Copilot, the "mean of means" (n1 = 14) runtime performance for valid solutions was **1898 ms** and without Copilot (n2 = 14) was **1628 ms** (also reported in 7.2). However, comparing runtime performance for Copilot (n1 = 14) versus without Copilot (n2 = 14) was not statistically significant (p = 0.667). This

explains why there was such a strong positive correlation between mode and runtime performance for problem A but a weak positive correlation for problem B.

This difference may also be due to the nature of the solutions themselves. The solutions to problem B involve the usage of a helper function while problem A does not need one unless sorting was invoked. A participant with higher C++ expertise may not need helper functions and simply use C++ lambdas [13] both for problem A (if using L2 or L3 concept-level optimization) and problem B.

8 LIMITATIONS AND TAKEAWAYS

8.1 Limitations

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 well-represented in many undergraduate and graduate level Operating Systems courses.

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. The takeaway from this is that participants were satisfied with their solution at least 10 minutes before their time was up.

It may seem that if participants were explicitly directed to optimize their code, the results of our study would have been slightly different; however, we argue otherwise. The crux of our research question is NOT how well Copilot can generate highly performant code compared to human developers but if they are better off without it. The former research question would have required a participant pool of system developers who were performance experts as well as a different experiment design and analysis.

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.

8.2 Takeaways

This work evaluated the performance of code generated by the self-proclaimed AI programming assistant GitHub Copilot by conducting a user study on systems programmers. While our study suggests that there may be some value in using Copilot to generate correct code, it also suggests that developers are better off without it when it comes to runtime performance. Even though we recommend that more experienced systems developers are better off without Copilot in their day-to-day tasks, we acknowledge that Copilot and others like it are a step in the right direction for programmers. With GitHub Copilot being one of the first production-ready programming assistants that are gaining ubiquity in modern software development, more research is required to acknowledge not only its limitations but also its strengths. As our study is one of the first to evaluate Copilot from a runtime performance perspective within systems programming, we hope future work can build off our experiment design to analyze Copilot's suggestions at a finer granularity when used by developers.

9 DATA AVAILABILITY

In order to respect participant privacy and anonymity as well as avoid potential ethical ramifications, 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. However, we provide the data set of the runtime performance of participants' solutions as well as the runtime performance of the model solutions [2].

REFERENCES

[1] About GitHub Copilot 2023. *About GitHub Copilot.* Retrieved 2023-01-31 from https://docs.github.com/en/copilot/overview-of-github-copilot/about-github-copilot

[2] Anonymous. 2023. *Participants' Solutions Runtime Performance Data with Model Solutions Runtime Performance Data.* https://doi.org/10.5281/zenodo.7600851

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. https://doi.org/10.48550/ARXIV.2005.14165

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. https://doi.org/10.48550/ARXIV.2107.03374

[5] Common Systems Programming Optimizations & Tricks 2019. *Common Systems Programming Optimizations & Tricks.* Retrieved 2023-01-31 from https://paulcavallaro.com/blog/common-systems-programming-optimizations-tricks/

[6] cstdio - C library to perform Input/Output operations 2022. *cstdio - C library to perform Input/Output operations.* Retrieved 2023-01-31 from https://cplusplus.com/reference/cstdio/

[7] Discord 2023. *Discord.* Retrieved 2023-01-31 from https://discord.com/

[8] Daniel Erhabor. 2023. Supplementary Material for Measuring the Runtime Performance of Code Produced with GitHub Copilot. https://doi.org/10.5281/zenodo.7921650

[9] fcntl.h - file control options 1997. *fcntl.h - file control options.* Retrieved 2023-01-31 from https://pubs.opengroup.org/onlinepubs/7908799/xsh/fcntl.h.html

[10] fstream - Input/output file stream class 2022. *fstream - Input/output file stream class.* Retrieved 2023-01-31 from https://cplusplus.com/reference/fstream/fstream/

[11] gcc-9 9.3.0-17ubuntu1 20.04 source package in Ubuntu 2020. *gcc-9 9.3.0-17ubuntu1 20.04 source package in Ubuntu.* Retrieved 2023-01-31 from https://launchpad.net/ubuntu/+source/gcc-9/9.3.0-17ubuntu1~20.04

[12] Internet Relay Chat (IRC) 2016. *Internet Relay Chat (IRC).* Retrieved 2023-01-31 from https://www.irchelp.org/

[13] Lambda expressions 2023. *Lambda expressions.* Retrieved 2023-01-31 from https://en.cppreference.com/w/cpp/language/lambda

[14] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. 2008. Measurement and Analysis of Large-Scale Network File System Workloads.

In *USENIX 2008 Annual Technical Conference* (Boston, Massachusetts) *(ATC'08)*. USENIX Association, USA, 213–226.

[15] Sheheeda Manakkadu and Sourav Dutta. 2014. Bandwidth Based Performance Optimization of Multi-threaded Applications. In *2014 Sixth International Symposium on Parallel Architectures, Algorithms and Programming*. 118–122. https://doi.org/10.1109/PAAP.2014.51

[16] memcpy - Copy block of memory 2022. *memcpy - Copy block of memory*. Retrieved 2023-01-31 from https://cplusplus.com/reference/cstring/memcpy/

[17] Operating Systems Development 2022. *Operating Systems Development*. Retrieved 2023-01-31 from https://wiki.osdev.org/Expanded_Main_Page

[18] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2021. An Empirical Cybersecurity Evaluation of GitHub Copilot's Code Contributions. *CoRR* abs/2108.09293 (2021). arXiv:2108.09293 https://arxiv.org/abs/2108.09293

[19] pthreads(7) — Linux manual page 2021. *pthreads(7) — Linux manual page*. Retrieved 2023-01-31 from https://man7.org/linux/man-pages/man7/pthreads.7.html

[20] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2022. Security Implications of Large Language Model Code Assistants: A User Study. https://doi.org/10.48550/ARXIV.2208.09727

[21] Serenity OS 2022. *Serenity OS*. Retrieved 2023-01-31 from https://serenityos.org/

[22] Skift OS 2021. *Skift OS*. Retrieved 2023-01-31 from https://skiftos.org/

[23] Yongseok Son, Heon Young Yeom, and Hyuck Han. 2017. Optimizing I/O Operations in File Systems for Fast Storage Devices. *IEEE Trans. Comput.* 66, 6 (2017), 1071–1084. https://doi.org/10.1109/TC.2016.2635644

[24] std::ios_base::openmode 2022. *std::ios_base::openmode*. Retrieved 2023-01-31 from https://en.cppreference.com/w/cpp/io/ios_base/openmode

[25] std::ios_base::seekdir 2021. *std::ios_base::seekdir*. Retrieved 2023-01-31 from https://en.cppreference.com/w/cpp/io/ios_base/seekdir

[26] std::ios::exceptions 1997. *std::ios::exceptions*. Retrieved 2023-01-31 from https://cplusplus.com/reference/ios/ios/exceptions/

[27] std::ios::good 2022. *std::ios::good*. Retrieved 2023-01-31 from https://cplusplus.com/reference/ios/ios/good/

[28] std::istream::seekg 2022. *std::istream::seekg*. Retrieved 2022-11-12 from https://cplusplus.com/reference/istream/istream/seekg/

[29] std::map 2022. *std::map*. Retrieved 2023-01-31 from https://cplusplus.com/reference/map/map/

[30] unistd.h - standard symbolic constants and types 1997. *unistd.h - standard symbolic constants and types*. Retrieved 2023-01-31 from https://pubs.opengroup.org/onlinepubs/7908799/xsh/unistd.h.html

[31] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. 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* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 332, 7 pages. https://doi.org/10.1145/3491101.3519665

[32] Vim - Visual Studio Marketplace 2022. *Vim - Visual Studio Marketplace*. Retrieved 2023-01-31 from https://marketplace.visualstudio.com/items?itemName=vscodevim.vim

[33] Visual Studio Code Remote - SSH 2023. *Visual Studio Code Remote - SSH*. Retrieved 2023-01-31 from https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.remote-ssh

[34] Wilcoxon Rank Sum and Signed Rank Tests 2023. *Wilcoxon Rank Sum and Signed Rank Tests*. Retrieved 2023-01-31 from https://stat.ethz.ch/R-manual/R-devel/library/stats/html/wilcox.test.html

[35] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-IDE Code Generation from Natural Language: Promise and Challenges. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 29 (mar 2022), 47 pages. https://doi.org/10.1145/3487569