A User-centered Security Evaluation of Copilot

Owura Asare
University of Waterloo
Waterloo, Canada
oasare@uwaterloo.ca

N. Asokan
University of Waterloo
Waterloo, Canada
asokan@acm.org

Meiyappan Nagappan
University of Waterloo
Waterloo, Canada
mei.nagappan@uwaterloo.ca

ABSTRACT

Code generation tools driven by artificial intelligence have recently become more popular due to advancements in deep learning and natural language processing that have increased their capabilities. The proliferation of these tools may be a double-edged sword because while they can increase developer productivity by making it easier to write code, research has shown that they can also generate insecure code. In this paper, we perform a user-centered evaluation GitHub's Copilot to better understand its strengths and weaknesses with respect to code security. We conduct a user study where participants solve programming problems (with and without Copilot assistance) that have potentially vulnerable solutions. The main goal of the user study is to determine how the use of Copilot affects participants' security performance. In our set of participants (n=25), we find that access to Copilot accompanies a more secure solution when tackling harder problems. For the easier problem, we observe no effect of Copilot access on the security of solutions. We also observe no disproportionate impact of Copilot use on particular kinds of vulnerabilities. Our results indicate that there are potential security benefits to using Copilot, but more research is warranted on the effects of the use of code generation tools on technically complex problems with security requirements.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; • **Computing methodologies** → *Natural language generation.*

KEYWORDS

user study, code generation, copilot, security, software engineering

1 INTRODUCTION

Code Generation Tools (CGTs) have recently become more popular due to their ability to make developers more productive during the software development process. By CGT, we refer to Large Language Models (LLMs) and their fine-tuned descendants that are used to generate code. These tools have improved as a result of progress in deep learning and natural language processing that have made it possible to train increasingly more capable large language models in an efficient manner.

CGTs have the ability to become large scale producers of insecure code if left to grow unchecked. This is because CGTs are trained on code sourced from repositories that lack security guarantees, making it probable that they are trained on code segments with security vulnerabilities. This in turn leads to the possible production of insecure code when these CGTs are used by developers. Empirical evidence by Pearce et al. [14] substantiates this claim, revealing that GitHub's Copilot[12], a widely-used CGT based on the Codex LLM[6], produces insecure code approximately 40% of the time.

To mitigate the possible adverse effects of CGTs, it is important that we gain a deeper understanding of their impacts on security. In this paper, we present our work on a user-centered security evaluation of GitHub's Copilot that aims to provide a better understanding of how CGTs affect code security. This study serves as a non-exact replication of other studies that have also conducted investigations of the security of CGTs through user studies, specifically the work by Sandoval et al.[16] and the work by Perry et al.[15]. The work by Pearce et al. [14] demonstrated that Copilot generates vulnerabilities when used as a standalone code completion tool (i.e. the code generated by Copilot was unedited). Our goal in this study is to find out whether the use of Copilot and the ability to edit its suggestions results in code with more or fewer vulnerabilities. A study of this nature is warranted because while CGTs become more popular among developers, their security capabilities remain under-explored and the differing conclusions of the previously mentioned studies suggest a lack of a consensus in the literature on the security effects of using them. Non-exact replication studies like ours play an important role in empirical science. It helps in generalizing results to a broader extent and contributes to the cumulative nature of scientific knowledge. In our study, we investigate how Copilot (when used as an assistant) affects users' security performance by designing and conducting a user study where participants solve programming problems with and without the assistance of Copilot. We observe that participants in our sample generally have a better security performance when access to Copilot is granted for difficult problems, and less so for relatively simpler problems. We also observe a more uniform performance across the different types of vulnerabilities when Copilot is in use. By this we mean that the presence of Copilot does not seem to disproportionately influence the presence or absence of any type of vulnerability. Overall, our results suggest that:

(1) the security benefits of using Copilot are more noticeable when it is used for more complex problems (Section 5.3)
(2) the use of Copilot does not necessarily increase or reduce the chances of any particular vulnerability (Section 5.4)

We provide access to all our study material at https://github.com/ppdb1123/copilot-user-study-supp

2 BACKGROUND

2.1 Language Models and Code Generation

Language models are generally defined as probability distributions over sequences of words. Language models model language by probabilistically predicting/generating the next word in a given sequence. They are able to do this by leveraging a set of parameters that are obtained after training on significant amounts of data. Over time, language models have evolved, from non-neural models (N-grams), to neural, recurrence-based models (RNNs) [4], to attention-based models (Transformers) [21].

The evolution and rise in popularity of language models have led to their applications to several tasks across many domains. One such domain is software engineering where language models have been put to the task of code generation. CGTs, available either through integrated development environments (IDEs) or as extensions to text editors, are already widely used by developers [9] and they continue to evolve in complexity. GitHub's Copilot [12] is an example of an evolved CGT. Copilot is generally described as an AI pair programmer trained on billions of lines of public code. Currently available as an extension for the VSCode text editor, Copilot takes into account the surrounding context of a program and generates possible code completions for the developer. IntelliCode [19] is another example of a CGT that generates recommendations based on thousands of open-source projects on GitHub.

Most current high performing models use the Transformer model which was initially introduced with two components: an encoder and a decoder. There are, however, high performing models that either only use the encoder [8] or the decoder [5]. Copilot is based on OpenAI's Codex [6], which is itself a fine-tuned version of GPT-3 [5].

3 RESEARCH OVERVIEW

3.1 Motivation

CGTs are designed to assist programmers during the code writing phase of the software development process. In this paper, we are interested in how Copilot affects the security of code written by the human participants in our user study. While there is existing research on evaluating LLM-based code assistants [15, 16], ours is the first to focus on Copilot which is the more mature and fine-tuned CGT. Copilot is also more popular and more accessible, which makes our findings more likely to be applicable to developers' experience in non-experimental settings.

3.2 Research Questions

(1) Does Copilot use correlate with participants writing more secure code?
(2) Are there vulnerability types that Copilot is more susceptible to or more resilient against?

4 METHOD

Here we discuss our approach for the study. Figure 1 summarizes our method.

4.1 Participants: Recruitment and Screening

Participants for this study were recruited online via mailing lists. While our main source of participants was the University of Waterloo computer science graduate student mailing list, we also extended invitations to industry professionals and potentially qualified undergraduate students. Participants who expressed interest in the study were asked to fill consent and screening forms which we used to determine their eligibility for the study. Selection criteria for this study was based on age (over 18 years), programming experience (at least one year of programming experience in C/C++), access to Copilot, and employment history (no affiliation with the development of Copilot, GitHub, or OpenAI). Participants who met our selection criteria were allowed to schedule a two hour online study session for the experiment to be conducted.

Like prior studies in this area, participants were not screened based on security experience because the goal of this study was to investigate the impact of Copilot use on code security among ordinary developers, regardless of security experience. We do however consider the idea of exploring the effects of participant security experience on the security of code generated with CGT assistance an interesting avenue for future research.

Overall, 33 people expressed interest in taking part in the study. 8 of them either did not complete the consent and screening process or did not select a time for the problem solving session. 25 out of the 33 people completed all stages of the study for a completion rate of 75%. The 25 participants were made up of 4 undergraduate students (16%), 19 graduate students (76%), and 2 professionals (8%). A majority of our participants (19/21) described themselves as "first time users" of Copilot, 5 of them indicated that they had "tried it out a few times" and 1 indicated that they "used it all the time".

4.2 Material: Programming Problems

4.2.1 Problem Design. We designed two problems for this study: problem S and problem T. In problem S, the participants had to implement a sign-in function for an application given a user's identifier and password. In problem T, the participants had to implement a function that performs a series of transactions in a given transaction file and then renames the file.

We decided to create our own set of problems for this study as we had specific criteria that called for tailored problems. Specifically, we sought problems that:

(1) had potential for vulnerable solutions,
(2) had solutions that could manually be analyzed
(3) resembled real world applications
(4) could be solved by participants within an hour

In order to address criteria 1 and 2, we designed the problems so that certain vulnerabilities could be introduced if participants were not careful with their solutions. The vulnerabilities that could be introduced were based on Common Weakness Enumerations (CWEs) [2]. CWEs are weaknesses in software and hardware systems and they are defined and maintained by the Mitre corporation. A CWE essentially represents a vulnerability and can be identified by its unique ID. For example, the classic buffer overflow vulnerability is represented by CWE-120. In designing each problem, we curated a set of CWEs that we wanted to include. We selected CWEs either
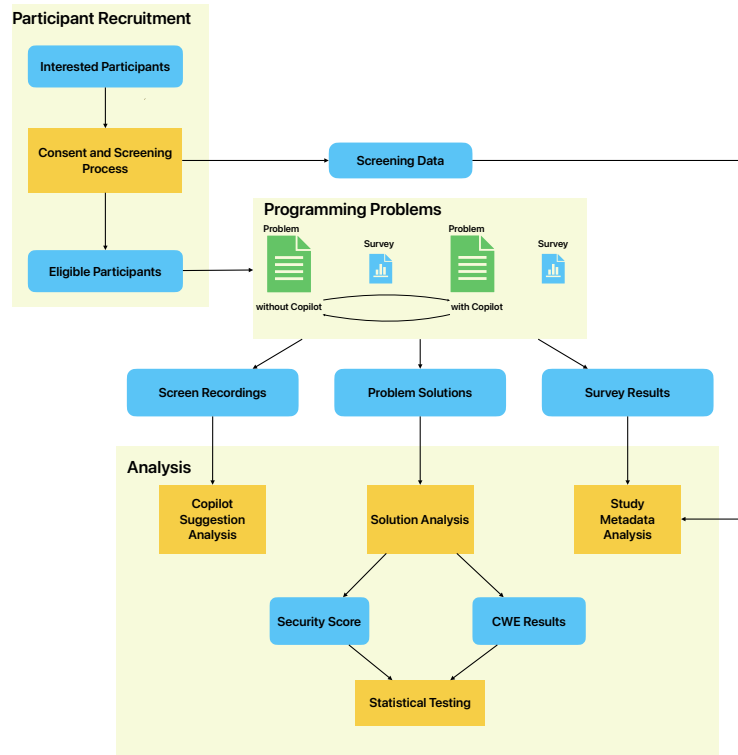
Figure 1: An overview of the user study, highlighting the key steps from recruiting participants to analyzing results.

because they were included on Mitre's top 25 most dangerous software weaknesses list [1] or because they were pertinent to the C programming language which was to be used in the study. The set of possible CWEs for each problem would subsequently be used for our analysis of participant solutions. Table 1 contains the CWEs that we focused on for each problem. Note that this set of CWEs is not exhaustive and there could have been other vulnerabilities possible in the problems we designed.

In order to address criteria 3 and 4, we created a collection of well documented helper functions (with stub implementations) for each problem. These helper functions enabled us to expand the level of difficulty of our problems (approximating real world applications) while also constraining and guiding users towards finding solutions within a confined solution space in the allotted time. We conducted pre-study testing to verify that Copilot could generate solutions using the stub helper functions provided in the problem file. After the study had commenced, it became apparent that there was a difference in difficulty between the two problems. We discuss this variation and how we adapted to it in section 5.1.

Other user studies conducted around the security of CGTs have designed different types of problems to different ends. The study by Sandoval et al.[16] designed a single large problem in the C programming language that participants had to solve within two weeks. However, Perry et al.[15] designed 6 (relatively) smaller problems in different programming languages including JavaScript and C that participants had to solve within 2 hours (20 minutes

each). These problems were less open-ended and had more straightforward solutions.

4.3 Procedure

4.3.1 Problem Solving. We employed a within-subject study design wherein all participants served in both the treatment group and the control group. All participants solved one programming problem with Copilot access and the other problem without Copilot access. Each participant involved in the study was programmatically assigned to one of four groups on a round-robin basis. The groups determined the order in which the problems were solved and whether Copilot would be used to solve problem S or problem T. There were four groups because there were two variables that determined how participants would solve the problems, and each variable had two possibilities. Any given participant could solve problem S first or problem T first. At the same time, the participant could either solve the first problem with Copilot and the second problem without Copilot, or vice versa. Participants were given 60 minutes to solve each problem together with an instruction sheet that they could reference during problem solving. Participants were informed (verbally and in the written instructions) that they were to write secure code. There were no restrictions on the resources participants could consult to aid in solving the problem other than the restrictions on Copilot use and the use of other CGTs. Each participant's screen was recorded during problem solving for subsequent analysis after the study session. All participants used Copilot

CWE-ID	Description	Problem S	Problem T
CWE-20	Improper Input Validation	✓	✓
CWE-22	Path Traversal		✓
CWE-78	OS Command Injection		✓
CWE-79	Cross-Site Scripting	✓	
CWE-89	SQL Injection	✓	✓
CWE-125	Out of Bounds Read	✓	✓
CWE-285	Improper Authorization	✓	✓
CWE-287	Improper Authentication	✓	✓
CWE-401	Memory Leak	✓	✓
CWE-415	Double Free	✓	✓
CWE-416	Use After Free	✓	✓
CWE-476	Null Pointer Dereference	✓	✓
CWE-787	Out of Bounds Write	✓	✓

Table 1: The list of Common Weakness Enumerations (CWEs) that we specifically checked for in each problem. Problems were designed such that the specified CWE could be introduced if participants were not careful enough about writing secure code.

in the Microsoft Visual Studio Code text editor. Participants were free to use and interact with Copilot in whatever manner they preferred. Solutions were saved once participants were done solving a given problem. Then, participants were required to fill out surveys to provide additional information about their perspective on the problem they just completed. Upon completion of the study, each participant was compensated CAD50.00.

4.3.2 Functionality Analysis. We tested solutions for functionality requirements with two of types tests: **basic tests** and **advanced tests**. Participants had access to the basic test during the study and had the option of testing their solutions on it if they desired. They did not have access to the advanced test. To perform the basic test, participants had to uncomment and run code provided for them in the `main` function of the problem file. The basic tests tested participant solutions on simple inputs, similar to what was described in the instructions. The advanced testing involved checking participant solutions on edge-case and more complex inputs such as null inner structs (problem S) and multiple transactions (problem T).

4.3.3 Security Analysis. All solutions submitted by participants were checked for the presence of the various CWEs possible for each problem (Table 1). This checking was performed manually by one author and one other independent coder. We resorted to

manual analysis of participant solutions because it has been proven to be sufficient when it comes to analyzing relatively short snippets of code [14, 15, 17]. Other research that has performed security analyses of code generated by CGTs has generally relied on manual analysis or CodeQL [10] to check for the presence of vulnerabilities. We used manual analysis because our preliminary testing of CodeQL showed that it was unable to detect any of the vulnerabilities in our test samples - it always generated false negative results. To ensure that CodeQL's poor performance was not due to any misconfiguration of our CodeQL setup, we performed additional tests to validate its setup. We used code snippets provided in the CodeQL GitHub repository [11] which were known to contain certain CWEs. For these examples, CodeQL was able to successfully identify the vulnerabilities. We also considered using fuzzing for our analyses but decided against it due to the proven track record of manual analysis and the use of stub helper functions in the programming problems, which would make adopting fuzzing a costly endeavour with no guarantee of better performance.

Solutions were analyzed independently by one of the authors and an independent coder - a Computer Science PhD student in our department with experience in C programming and vulnerability analysis. For each solution, both parties checked for the presence of each of the CWEs in Table 1 and stored their results separately. The results were subsequently cross-referenced to find situations where the author and coder were in disagreement about the presence or absence of a CWE in a solution. A vulnerability was only considered present or absent in a solution if both the author and the coder were in agreement. Where there were disagreements, the author and the coder discussed until a consensus was reached. This was required in less than 5% of the cases. The kappa score, a metric used to measure inter-rater reliability, was calculated to be 0.962 for the vulnerability labeling in this study, indicating that the level of agreement between the coder and the author was close to perfect. The manual analysis of solutions resulted in a security score for the two problems solved by each participant. The security score, outlined below, is a function of the number of vulnerabilities present in a participant's solution. For our purposes, a solution with a higher security score is more secure than a solution with a lower security score. The security score ranges from 0 (all vulnerabilities found) to 100 (no vulnerabilities found). While the security score was computed for all solutions, only those that compiled and passed the basic test were used for subsequent analyses.

$$PercentageVulnerable = \frac{\text{Number of Vulnerabilities found}}{\text{Total number of Vulnerabilities Possible}} * 100$$

Security Score $= 100 - PercentageVulnerable$

4.4 Ethics

This user study obtained ethics clearance from the Human Research Ethics Board at the University of Waterloo. Participant consent was obtained during the recruitment process and consenting participants were screened to ensure they met the desired criteria. Participants were informed that their screens would be recorded during the session. Data collected during sessions, including screen recordings, problem solutions, and survey information, were linked to

	Educational Level			Dev. Experience (Yrs)			Copilot Experience		
	Undergraduate	Graduate	Professional	1-5	6-10	greater than 10	"first time user"	"tried it out"	"frequent user"
Count	4	19	2	14	8	3	19	5	1

Table 2: Summary of participants' background

anonymous IDs created for each participant. We maintained a key in a secure vault linking participant information (name and email address) to IDs that will be deleted once all analysis is complete and no further contact with participants is required.

5 RESULTS AND DISCUSSION

5.1 Overview

Table 2 summarizes the demographic data that we collected about participants, namely their level of education, experience writing code and experience with Copilot. Table 3 summarizes the data about participant performance in our study. 17 participants submitted valid solutions for both problems. As mentioned earlier, valid solutions were those that compiled and at least passed the basic test. In table 3, valid solutions for a problem correspond to rows where PS Func. or PT Func. are greater than or equal to 2 (highlighted in yellow). Of the 17 that submitted valid solutions for both problems, 8 were better with Copilot (i.e. wrote more secure code) and 9 were better without Copilot. The average security score with Copilot (**65.2**, std=18.5) was lower than the average security score without Copilot (**66.3**, std=19.8).

Looking at the problems separately, 20 participants submitted valid solutions for problem S and 18 participants submitted valid solutions for problem T. On average, participants took 27.4 minutes to submit a solution for problem S (std=14.1) and 42.7 minutes to submit a solution for problem T (std=13.8).

Before the study officially began, we tested our problems on two volunteers who were representative of the kind of people we expected to be in the actual study. These volunteers provided feedback which we used to edit our problems before proceeding with study. The feedback they provided addressed two points: the high level of difficulty of one of the problems (problem T) and the clarity of some of the instructions. While we attempted to address both concerns, our results (time taken and average security scores) indicate that the comparatively higher difficulty of one of the problems over the other may have persisted. As a result, we adapted our analysis and discussion to account for this variation in difficulty. We specifically looked at the results from solving each problem and performed statistical tests independently. All calculations and data aggregations as well as visualizations were duplicated for both problems.

5.2 Copilot Suggestion Analysis

To understand the extent to which Copilot contributed to participants' solutions, we performed some analysis of the screen recordings generated during study sessions with participants. We were unable to track the provenance (i.e., from the participant or from Copilot) of each vulnerability found through this process because participants accepted and edited code, and interacted with Copilot in several different ways. In most cases, the presence or absence of a vulnerability could only be determined when the participant

had finished editing the file. Thus, while we could know that Copilot had suggested a piece of code, we could not at all times check for vulnerabilities because these vulnerabilities existed within the larger context of the file. For this analysis, we focused on tracking the number of suggestions Copilot made, the number of suggestions that were accepted, and the number of suggestions that were edited after being accepted.

Our analysis of Copilot suggestions yielded two important takeaways. First, we noticed that the acceptance rate did not change significantly between the two problems, despite their different difficulty levels. Participants used it at the same rate for the easier problem (problem S) as they did for the harder problem (problem T). The second takeaway was that Copilot played at least a minor role in all solutions submitted by participants for problems where Copilot was permitted. This is evident from the fact that all participants accepted at least 6 Copilot suggestions, each with at least 1 line of code. Table 4 provides a quantitative summary of the Copilot suggestion analysis.

5.3 RQ1: Does Copilot use correlate with participants writing more secure code?

5.3.1 Approach. To investigate the possible effects of Copilot on the security of participant solutions, we looked at participant security scores with and without Copilot. We first computed summary statistics (mean, median, standard deviation) of security scores for both problems. This gave as an overview of the overall performance (per problem) with and without the assistance of Copilot. We subsequently performed statistical tests to see whether there was a significant difference between the two groups. For each problem, we used the Kruskal-Wallis to test for statistically significant differences between the group that used Copilot and the group that did not. This test was performed independently for each problem to account for any differences in their level of difficulty. We chose the Kruskal-Wallis test because it allowed us to compare the scores from the two independent groups even when the data did not follow a normal distribution, an assumption made by other (parametric) tests like the T-test.

5.3.2 Results. We used the security score (computed following the steps in subsection 4.3) as the basis for evaluating the security of solutions. Figure 2 summarizes the impacts of the use of Copilot on security scores for problem S and Figure 3 does the same for problem T. For problem S, we obtained the same median security score of 81.82 both for participants who solved it with Copilot access and for participants who solved it without Copilot access. For problem T, the median security score for participants who solved it with Copilot access was 54.17 compared to 41.67 for participants who solved it without Copilot.

Using the Kruskal-Wallis statistical test, we found no statistically significant differences in security scores for both problem S (statistic

ID	PS Score	PT Score	PS Time (mins)	PT Time (mins)	PS Func.	PT Func.
001	54.5	41.7	23	28	3	2
002	81.8	58.3	25	23	2	2
003	81.8	91.7	42	51	3	3
004	81.8	50.0	14	38	3	1
005	63.6	41.7	10	46	3	2
006	63.6	41.7	27	41	3	2
007	81.8	50.0	20	25	3	3
008	90.9	41.7	14	35	3	3
009	72.7	58.3	13	59	1	1
010	81.8	33.3	50	53	3	3
011	100.0	50.0	60	40	0	1
012	81.8	33.3	38	60	3	1
013	81.8	41.7	11	55	2	2
014	81.8	75.0	25	21	3	3
015	63.6	66.7	60	19	1	2
016	81.8	41.7	29	47	3	3
017	36.4	25.0	25	60	1	1
018	90.9	50.0	26	51	3	3
019	72.7	66.7	21	22	3	3
020	81.8	50.0	27	39	2	2
021	81.8	91.7	15	42	3	3
022	90.9	33.3	21	32	3	3
023	81.8	50.0	41	60	3	1
024	81.8	66.7	11	60	0	0
025	72.7	50.0	35	60	2	2

Table 3: Participants in the study and their performance (security scores) on problem S (PS) and problem T (PT). Highlighted cells in the "PS Score" and "PT Score" columns indicate that the score was obtained with Copilot. The time columns show the times taken to solve each problem. The last two functionality columns indicate the level of functionality of participant solutions which are described as follows: 0 = did not compile, 1 = only compiled, 2 = compiled and passed only the basic test, 3 = compiled and passed both the basic and the advanced test. Highlighted cells in the functionality columns indicate solutions that did not sufficiently implement the functionality requested in the problems and were therefore excluded from our analysis.

	Number of Copilot Suggestions					Participant Reaction			
	Mean	Median	Std. Dev.	Range (Suggested)	Range (Accepted)	Avg. Acc. Rate	Std. Dev.	Avg. Edit Rate	Std. Dev.
Problem S	10.90	11	6.23	8 - 14	6 - 13	84.99%	11.17%	22.42%	26.12%
Problem T	21.10	21	1.79	14 - 29	8 - 24	83.53%	12.61%	17.16%	12.81%
Both	16.00	14	4.58	8 - 29	6 - 24	84.26%	11.62%	19.79%	20.21%

Table 4: Describing the nature of Copilot suggestions and how users interacted with it. The table shows the mean, median, and the range of the number of suggestions made by Copilot as well as the range of the number of accepted suggestions and the acceptance and edit rates.

= 0.59, p = 0.44) and problem T (statistic = 0.83, p = 0.36). The results of the tests indicate that we cannot reject the possibility that Copilot has no effect on the security of code written by participants.

However, from our sample, looking specifically at problem T, we observe a marked difference between the median security score with Copilot and median security score without Copilot - the score with Copilot is higher by about 13 points. This difference in scores also applies to the mean; the score with Copilot is higher than the score without Copilot by about 6 points. On the other hand, we see no such differences in scores for problem S - the medians are exactly the same. Considering that problem T appeared to be more difficult for participants to solve (it took longer to solve on average and had lower security scores overall), it seems that Copilot benefited participants when they encountered the more complex problem and had little effect when the problem was more straightforward.

A possible explanation for this difference in performance is that when presented with the harder problem, participants' priorities shifted from finding *a secure solution* to finding *any solution*. To
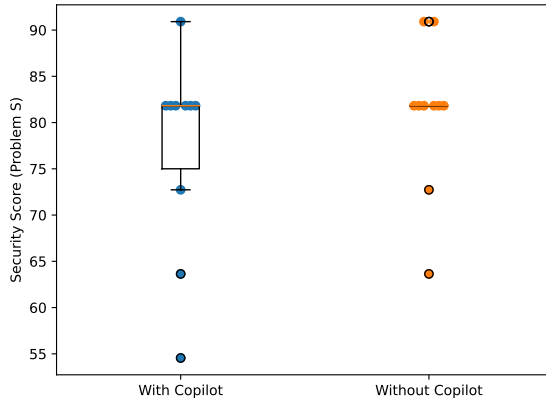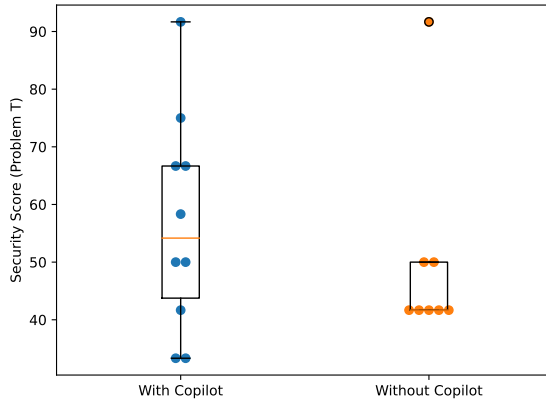
(a) Box plots describing security scores with and without the use of Copilot for Problem S.

	Mean	Median	Std. Dev.
With Copilot	77.27	81.82	10.71
Without Copilot	80.91	81.82	7.96

(b) Descriptive statistics of participant's performance for Problem S.

Figure 2: Box plot and table summarizing participant's performance for Problem S with and without the use of Copilot.

(a) Box plots describing security scores with and without the use of Copilot for Problem T.

	Mean	Median	Std. Dev.
With Copilot	56.67	54.17	18.76
Without Copilot	50.00	41.67	17.25

(b) Descriptive statistics of participant's performance for Problem T.

Figure 3: Box plot and table summarizing participant's performance for Problem T with and without the use of Copilot.

achieve this, participants may have been less concerned about the security of the code they were writing. Those who had access to Copilot for this problem may also have been less concerned with the level of security of Copilot suggestions as indicated by the lower edit rate for problem T in table 4. However, even if the participant's priorities had changed, Copilot's *priorities* remained the same. Under these circumstances, participants who had access to Copilot for problem T benefited from its ability to not sacrifice security for expediency or functionality. The flip side of this discussion, which we cannot verify from the perspective of this study, is that since Copilot's priorities remain unchanged, users who prioritize security at least as much as functionality may be negatively impacted by using it.

> **Summary:** In RQ1, we investigated whether using Copilot correlates with participants writing more secure code. While not statistically significant, we observed that participants wrote more secure code when they had access to Copilot for the more difficult problem.

## 5.4 RQ2: Are there vulnerability types that Copilot is more susceptible to or more resilient against?

5.4.1 Approach. We investigated the possibility of Copilot having a disproportionate impact on certain vulnerability types by looking at the frequency of vulnerabilities and how that frequency changed with and without the use of Copilot. We further ran Fisher's exact statistical test on the collected counts to determine whether Copilot's impact on the presence/absence of a vulnerability was statistically significant. Tests on Copilot's impact on the different vulnerabilities were performed separately for each problem, but we also performed a joint analysis for vulnerabilities that were common to both problems.

5.4.2 Results. Table 5 presents the data about the different vulnerabilities found for each problem with and without the use of Copilot. For problem S, a total of 46 vulnerabilities were found. 25 were found with Copilot (i.e. were found when the participant was allowed to use Copilot) and 21 were found without Copilot. Overall, 54% of vulnerabilities were found with Copilot for problem S. For problem T, a total of 100 vulnerabilities were found, with 52 (56%) being found with Copilot and the remaining 48 without. An inspection of these summary statistics and the frequencies of each individual CWE with and without Copilot did not reveal any clear or significant impact of Copilot on the presence of any particular vulnerability. To be sure, we also ran Fisher's exact test on a 2x11 contingency table for problem S and a 2x12 contingency table for problem T using the frequencies in Table 5 as the counts. The results of the tests for both problems indicated that there was no statistically significant difference between frequencies with Copilot and frequencies without Copilot (p=0.99 for problem S, p=0.94 for problem T). The results further indicate that as far as our sample is concerned, we cannot reject the possibility that Copilot has any statistically significant effect on the presence of the CWEs tested in this study.

> **Summary:** In RQ2, we investigated Copilot's influence on the presence of certain CWEs. We found no significant impact of Copilot access on the introduction of the CWEs we considered.

	Problem S		Problem T		Total		
	With	*Without*	*With*	*Without*	*With*	*Without*	*Total*
CWE-20	7	5	9	7	16	12	28
CWE-22	-	-	4	7	4	7	11
CWE-78	-	-	5	7	5	7	12
CWE-79	3	2	-	-	3	2	5
CWE-89	2	1	9	7	11	8	19
CWE-125	0	0	1	0	1	0	1
CWE-285	1	1	1	2	2	3	5
CWE-287	0	1	4	2	4	3	7
CWE-401	10	9	10	8	20	17	37
CWE-415	0	0	0	0	0	0	0
CWE-416	0	0	0	0	0	0	0
CWE-476	2	2	7	7	9	9	18
CWE-787	0	0	2	1	2	1	3

Table 5: Counts of the number of times each CWE was found for both problems. The "With" columns indicate the number of times a CWE was found when Copilot was involved in solving the problem. "Without" indicates that Copilot was not involved. Dashes indicate that the particular CWE was not tested for in that problem.
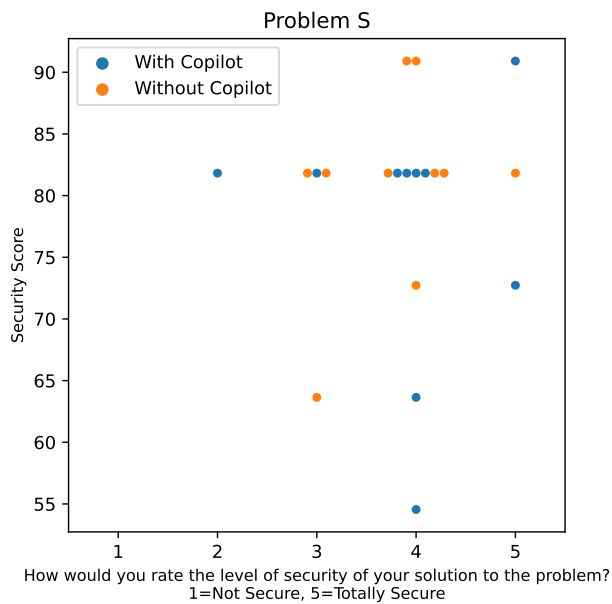
Figure 4: Plot showing how participants' opinions compared to their security scores with and without Copilot for problem S.
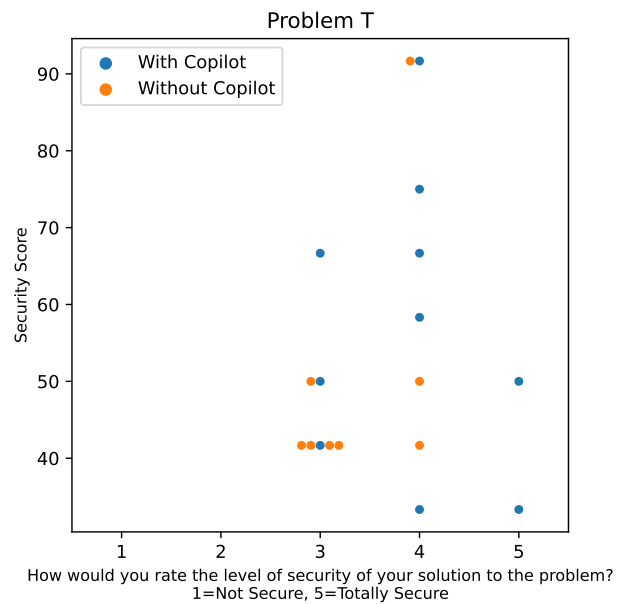
Figure 5: Plot showing how participants' opinions compared to their security scores with and without Copilot for problem T.

5.5 Survey Results

After solving each problem, participants were asked to fill out surveys. The amount of time that participants used to solve each problem was also recorded. For both problems, we found that the median time used in implementing a solution was less with Copilot than without. This is not surprising considering the high suggestion acceptance rates for both problems (Table 4). When we asked participants to rate how helpful Copilot had been on a scale of 1 (not helpful) to 5 (very helpful), 64% of them (16 out of 25) indicated

that Copilot had been very helpful, giving it the maximum rating of 5. Of the remaining participants, 6, 2, and 1 of them rated Copilot's helpfulness at a 4, 3, and 2 respectively.

We also asked participants to provide ratings on how secure they felt their solutions were and how confident they were that their solutions were correct. We observed that opinions on correctness and security were generally high with a majority of participants giving ratings of 4 or 5 (out of 5) on both accounts. However, this was more true for correctness where high scores were given 82% of

the time than for security where high scores were given 58% of the time.

Figures 4 and 5 show how participant opinions on the level of security of their submissions compared to their actual security scores for problem S and problem T respectively. For problem S, we observed no significant trends between participant opinions and their corresponding security scores. For problem T, we observed that participants without Copilot were generally not as confident in the security of their solution as those with Copilot access and this low confidence loosely tracked with the lower security scores in this group as previously discussed.

5.6 Takeaways & Future work

In this study, we investigated the the impact of using Copilot on code security (RQ1) and the amount of influence that Copilot has on the presence of certain CWEs (RQ2). While our findings are not statistically significant due to a limited sample size, for the former, we observed that participants wrote more code secure for the more difficult problem when they were granted access to Copilot. In the latter, we observed no significant impact of Copilot access on the presence of any of the CWEs considered in this study.

In order to verify whether Copilot does in fact make a significant difference on harder problems as suggested by the result from RQ1, a more targeted user study may be required. Such a study would require a set of multiple problems, each with varying levels of difficulty. Ideally, we would want to establish a proxy for problem difficulty that can be compared to a proxy for security (like the security score in this user study) during analysis. The proxy for problem difficulty could be obtained from a number of sources including the time taken to solve problems or some aggregate of rankings of problem difficulty by users after they have solved the problems. Participants in the study would then be split into control and treatment groups, wherein the former solve all problems without Copilot and the latter solve all the problems with Copilot. We would then be able to measure how differences in security performance between the control and treatment group are affected by problem difficulty.

The idea that the security impacts of Copilot could be more significant at higher levels of difficulty has implications for future research on and testing of Copilot and similar CGTs. Mainly, it implies that testing CGTs on trivial problems could yield misleading results. Researchers and developers of CGTs may want to take steps to ensure that the problems upon which their tools are tested and evaluated are above a certain level of complexity, especially when the tools are being tested in conjunction with human users. For regular users of Copilot, a takeaway from the observations from our sample is the suggestion that Copilot can be especially helpful in writing more secure code when tackling more complex problems.

6 THREATS TO VALIDITY

6.1 Construct Validity

A possible threat to the construct validity of this study is the manual analysis used to evaluate participant solutions. In order to check for the presence of vulnerabilities, we manually analyzed participant solutions. It is possible that this analysis process may have

missed (false negative) or misidentified (false positive) certain vulnerabilities. As a mitigation, we used two evaluators to decrease the chances of false results.

Our participant screening process did not take participants' security experience into account. This was because we wanted to investigate the impact of Copilot use on code security regardless developers' of security experience. However, we believe a study that explores the relationship between the security experience of users of CGTs and the security of code generated with CGTs would be interesting and we leave it as an avenue for future work.

6.2 External Validity

Threats to the external validity of this study are the sample size and sample composition. While we observe some effects of Copilot on the security of solutions for problem T, the tests we perform indicate that our findings are not statistically significant. This indicates that we cannot assume that the observations in our sample generalize to the larger population. Further, the majority of our sample (approximately 90%) comprised students, both at the graduate and undergraduate level. As a result, our observations may also not be generalizable to professional, full-time software developers. We relaxed our selection criteria and designed accessible problems in order to be able to reach a wider audience while retaining the integrity of the study. We also provided compensation for participants who completed the study. However, there were also time constraints that determined when we could no longer accept participants. For future studies, the goal would be to have the study open for a longer time and take additional steps to reach a wider audience outside of the university environment. Still, we believe the findings of this study are still useful. The study highlights potential security benefits of CGTs like Copilot, specifically their being more beneficial (security-wise) when solving difficult problems as well as their not being disproportionately susceptible to the CWEs considered. The study also sets the stage and direction for future studies on the security of CGTs and the ways in which they can be improved.

7 RELATED WORK

There are a number of papers that have conducted evaluations of CGTs [3, 5–7, 13, 18, 20, 22–24]. As mentioned earlier, most evaluations tend to focus less on security. There are however two existing user studies of CGTs with a focus on security that we would like to discuss briefly.

Sandoval et al. [16] conducted a user study that sought to investigate the cybersecurity impact of LLMs on code written by student programmers. They specifically evaluated the Codex language model on a sample size of 58 students. They found a small impact of LLMs on code security and a beneficial impact on functional correctness, indicating their use did not introduce new security risks but helped participants generate more correct solutions.

On the other hand, Perry et al. [15] also performed a large-scale study that also aimed to determine whether users wrote more insecure code with AI assistants. They also performed their evaluation using the Codex model and a sample size of 47 participants. They found that participants who had access to the Codex assistants

	This Study	Sandoval et al. [16]	Perry et al. [15]
Tool Evaluated	Copilot	Codex (code-cushman-001)	Codex (code-davinici-002)
Sample Size	25	58	47
Sample make-up	CS Students and Professionals	CS Students	CS Students and Professionals
Number of Problems	2	1 (subdivided into 12 functions)	6
Time Given	1 hour per problem	2 weeks	2 hours total
Programming Languages	C	C	Python, JavaScript, C
Problem Design	Participants were tasked with solving two problems: one that implemented user sign on a website and the other that implemented transaction fulfillment. In addition to other criteria, the problems were designed to mimic real world functionality, to be solvable within an hour, and to have the potential for insecure solutions.	Participants were asked to implement a shopping list based on a singly linked list data structure. The problem was designed to have the potential for several memory related bugs.	Participants were asked to solve 6 relatively short problems in different languages. The problems were more direct in terms of security risks. Potential security risks were not obscured by higher level functionality requirements such as a shopping list or user sign in. For example participants were directly asked to implement cryptographic encryption, message signing, and displaying a string input in a browser.
Study Approach	**All participants solved both problems** - one problem was solved with Copilot and the other problem without Copilot. This way, **each participant served in the treatment group for one problem and the control group for the other problem**.	Each participant was assigned to either the treatment or the control group.	Each participant was assigned to either the treatment or the control group.
Mode of CGT Use	Participants used the Copilot extension in the Visual Studio Code text editor.	Participants used a custom VS Code extension connected to a codex model.	In addition to a custom UI for writing solutions, participants were provided a separate interface where they could query the codex model and then copy and paste results into their solution.
Main Security Findings	Participants generally submitted more secure solutions when they had access to AI assistance for the harder problem. For the easier problem, no difference was observed. We also observed no significant difference in performance across the different vulnerability types.	In their context, the LLM did not increase the incidence rate of severe vulnerabilities.	Participants with access to AI assistance produced more security vulnerabilities and were more likely to believe that they wrote secure code.
CWEs Tested	CWE-20, CWE-22, CWE-78, CWE-79, CWE-89, CWE-125, CWE-285, CWE-287, CWE-401, CWE-415, CWE-416, CWE-476, CWE-787	CWE-119, CWE-400, CWE-416, CWE-476, CWE-787, CWE-190, CWE-252, CWE-758, CWE-835	Did not make use of the CWE framework.

Table 6: Table summarizing the differences between three user studies, by different authors, on the effects of CGTs on code security.

wrote significantly less secure code than those without access, and were also more likely to believe they wrote more secure code.

An insight from our user-centered evaluation that is not present in the other studies is the idea that Copilot could be more beneficial (security-wise) for more difficult problems. Beyond that, we observe that our findings about CGT security performance align slightly with those of Sandoval et al. [16] in the sense that they both report either neutral or positive impacts of CGTs on security. These studies have other things in common that could explain this similarity, specifically the focus on a single language (C) and the use of more in depth problems. On the other hand, we note that the findings

by Perry et al. [15] tell a different story - indicating that CGTs negatively impact the security performance of users. The simplest reason for this contradictory finding is the several differences in approach/methodology outlined in table 6, chief among them being the fact that each study evaluates a different tool. The difference in results across the studies suggests that we may not want to generalize the performance of one CGT to all other CGTs.

8 CONCLUSION

In this user-centered evaluation of Copilot, we aimed to determine whether using Copilot correlated with participants writing more secure code (RQ1) and whether there were vulnerability types that Copilot was more susceptible to or more resilient against (RQ2). For RQ1, while there were no major differences in security performance between the two groups (with and without Copilot access) for problem S, we observed that the group with Copilot access for problem T (the relatively harder problem) tended to have higher security scores compared to the group without Copilot access for the same problem. We believe this may be due to the fact that when presented with a seemingly harder problem, participants became more focused on finding *a solution* than finding *a secure solution.* Under these circumstances, those who had access to Copilot may have benefited from a source of code (other than themselves) that placed no less (or more) a premium on secure code. While beyond the scope of this study, we discussed ways of further testing this explanation. For RQ2, we observed a fairly uniform security performance across the different vulnerability types indicating that there was no disproportionate impact of Copilot access on the presence of any one vulnerability type.

9 ACKNOWLEDGEMENTS

REFERENCES

[1] 2023. 2023 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

[2] 2023. The MITRE Corporation. https://mitre.org

[3] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2022. Grounded Copilot: How Programmers Interact with Code-Generating Models. http://arxiv.org/abs/2206.15000 arXiv:2206.15000 [cs].

[4] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. 2000. A Neural Probabilistic Language Model. In *Advances in Neural Information Processing Systems*, Vol. 13. MIT Press. https://proceedings.neurips.cc/paper/2000/hash/728f206c2a01bf572b5940d7d9a8fa4c-Abstract.html

[5] 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. http://arxiv.org/abs/2005.14165 arXiv:2005.14165 [cs].

[6] 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. *arXiv:2107.03374 [cs]* (July 2021). http://arxiv.org/abs/2107.03374 arXiv: 2107.03374.

[7] Matteo Ciniselli, Luca Pascarella, and Gabriele Bavota. 2022. To What Extent do Deep Learning-based Code Recommenders Generate Predictions by Cloning Code from the Training Set? http://arxiv.org/abs/2204.06894 arXiv:2204.06894 [cs].

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805 [cs]* (May 2019). http://arxiv.org/abs/1810.04805 arXiv: 1810.04805.

[9] Thomas Dohmke. 2022. GitHub Copilot is generally available to all developers. https://github.blog/2022-06-21-github-copilot-is-generally-available-to-all-developers/

[10] GitHub Inc. 2019. CodeQL. https://codeql.github.com/

[11] GitHub Inc. 2019. CodeQL Repository. https://github.com/github/codeql

[12] GitHub Inc. 2021. GitHub Copilot · Your AI pair programmer. https://github.com/features/copilot

[13] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. https://doi.org/10.48550/ARXIV.2203.07814

[14] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. 754–768. https://doi.org/10.1109/SP46214.2022.9833571 ISSN: 2375-1207.

[15] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2022. Do Users Write More Insecure Code with AI Assistants? *arXiv preprint arXiv:2211.03622* (2022). https://arxiv.org/abs/2211.03622

[16] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. 2023. Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants. https://www.usenix.org/system/files/sec23fall-prepub-353-sandoval.pdf

[17] Mohammed Latif Siddiq and Joanna C. S. Santos. 2022. SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (MSR4P&S 2022)*. Association for Computing Machinery, New York, NY, USA, 29–33. https://doi.org/10.1145/3549035.3561184 event-place: Singapore, Singapore.

[18] Dominik Sobania, Martin Briesch, and Franz Rothlauf. 2022. Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, Boston Massachusetts, 1019–1027. https://doi.org/10.1145/3512290.3528700

[19] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode compose: code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Virtual Event USA, 1433–1443. https://doi.org/10.1145/3368089.3417058

[20] 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 *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. ACM, New Orleans LA USA, 1–7. https://doi.org/10.1145/3491101.3519665

[21] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, \Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010. event-place: Long Beach, California, USA.

[22] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. ACM, San Diego CA USA, 1–10. https://doi.org/10.1145/3520312.3534862

[23] Weixiang Yan and Yuanchun Li. 2022. WhyGen: Explaining ML-powered Code Generation by Referring to Training Examples. http://arxiv.org/abs/2204.07940 arXiv:2204.07940 [cs].

[24] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. ACM, San Diego CA USA, 21–29. https://doi.org/10.1145/3520312.3534864