A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges

Jenny T. Liang
Carnegie Mellon University
Pittsburgh, PA, USA
jtliang@cs.cmu.edu

Chenyang Yang
Carnegie Mellon University
Pittsburgh, PA, USA
cyang3@cs.cmu.edu

Brad A. Myers
Carnegie Mellon University
Pittsburgh, PA, USA
bam@cs.cmu.edu

ABSTRACT

The software engineering community recently has witnessed widespread deployment of AI programming assistants, such as GitHub Copilot. However, in practice, developers do not accept AI programming assistants' initial suggestions at a high frequency. This leaves a number of open questions related to the usability of these tools. To understand developers' practices while using these tools and the important usability challenges they face, we administered a survey to a large population of developers and received responses from a diverse set of 410 developers. Through a mix of qualitative and quantitative analyses, we found that developers are most motivated to use AI programming assistants because they help developers reduce key-strokes, finish programming tasks quickly, and recall syntax, but resonate less with using them to help brainstorm potential solutions. We also found the most important reasons why developers do *not* use these tools are because these tools do not output code that addresses certain functional or non-functional requirements and because developers have trouble controlling the tool to generate the desired output. Our findings have implications for both creators and users of AI programming assistants, such as designing minimal cognitive effort interactions with these tools to reduce distractions for users while they are programming.

CCS CONCEPTS

• **Software and its engineering** → **Software notations and tools**; • **Human-centered computing** → **Empirical studies in HCI**; • **Computing methodologies** → *Natural language processing*.

KEYWORDS

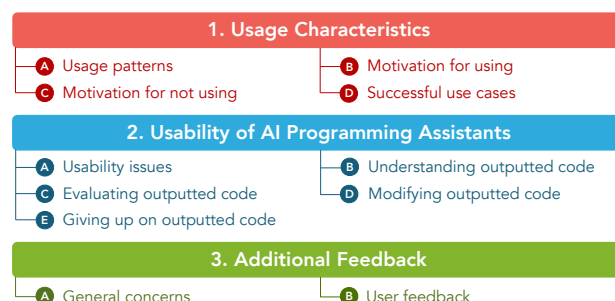AI programming assistants, usability study

Figure 1: An overview of the topics covered in our usability study of AI programming assistants.

1 INTRODUCTION

The recent widespread deployment of AI programming assistants, such as GitHub Copilot [6] and ChatGPT [1], has introduced a new paradigm to building software that has taken the software engineering community by storm. Some current publications report that AI programming assistants are powerful enough to produce high-quality code suggestions for developers [59, 61]. While some recent studies do not find any significant difference in using AI programming assistants in terms of task completion [56, 60] and code quality [29], other studies find these tools are positively associated with developers' self-perceived productivity [62].

However, in practice, prior literature indicates that developers do not accept AI programming assistants' initial suggestions at a high frequency. Ziegler et al. [62] found that developers accepted 23.3%, 27.9%, and 28.8% of GitHub Copilot's suggestions for TypeScript, JavaScript, and Python respectively. There are many potential reasons for the lack of adoption of AI programming assistants' suggestions. One study shows that developers feel concerned that the generated code may contain defects, may not adhere to the project's coding style, or may be difficult to understand [56]. Other studies report that software developers face barriers in comprehending and debugging generated code to fit their use cases, because they need to have prior knowledge of the underlying programming principles, frameworks, or APIs [12, 60].

While prior work has surfaced initial results about the usability of state-of-the-art AI programming assistants, to our knowledge, they have not systematically investigated the prevalence of usability factors related to these tools. Quantifying the usability of AI programming assistants could help tool creators understand which usability aspects are currently successful in practice. Further, it could help tool creators prioritize features and improvements to the modeling and user interface of these tools in the future, potentially

increasing the adoption of these tools and improving the productivity of developers. Usability is an important factor to study in AI programming assistants, since modeling improvements may not necessarily address the needs of developers, rendering these tools hard-to-use or even useless [45].

We performed an exploratory qualitative study in January 2023 to understand developers' practices when using AI programming assistants and the importance of the usability challenges that they face. We used a survey as a research instrument to collect large-scale data on these phenomena to understand their importance to the usability of AI programming assistants (see Figure 1).

In the end, we collected and analyzed responses from 410 developers who were recruited from GitHub repositories related to AI programming assistants, such as GitHub Copilot and Tabnine [2]. In summary, we find that:

Usage characteristics of AI programming assistants (Section 4)

(1) Developers who use GitHub Copilot report a median of 30.5% of their code being written with help from the tool.

(2) Developers report the most important reasons why they use AI programming assistants are because of the tools' ability to help developers reduce key-strokes, finish programming tasks quickly, and recall syntax.

(3) The most important reasons why developers do *not* use these tools at all are that the tools generate code that do not meet certain functional or non-functional requirements and that it is difficult to control these tools to generate the desired output.

Usability of AI programming assistants (Section 5)

(4) Developers report the most prominent usability issues are that they have trouble understanding what inputs cause the tool's generated code, giving up on incorporating the outputted code, and controlling the tool to generate helpful code suggestions.

(5) The most frequent reasons why users of these tools give up on using outputted code are that the code does not perform the correct action or it does not meet functional or non-functional requirements.

Additional feedback about AI programming assistants from users (Section 6)

(6) Developers would like to improve their experience with AI programming assistants by providing feedback to the tool to correct or personalize the model as well as by having these tools to learn a better understanding of code context, APIs, and programming languages.

In this paper, we refer to *tool creators* as the individuals who build and develop software related to AI programming assistants. *Tool users* are the people who use these tools while building software. We use this term interchangeably with *developers*. Finally, we use the term *inputs* to refer to the code and natural language context AI programming assistants use to produce *outputted code*, which we also call *generations*.

2 RELATED WORK

We discuss work related to the usability of AI programming assistants. Since this field is rapidly developing, the papers discussed are a snapshot of the current progress in the field as of March 2023.

Prior work includes a few usability studies on various AI programming assistants using programming by demonstration approaches [14, 20] and recurrent neural networks-based approaches [39]. Lin et al. [39] reported that developers have difficulty in correcting generated code, while Ferdowsifard et al. [20] showed that a mismatch in the perceived versus actual capabilities of program synthesizers may prevent the user from using them effectively. Meanwhile, Jayagopal et al. [30] also conducted usability studies to understand the learnability of five of these tools with novices. Finally, McNutt et al. [43] enumerated a design space of interactions with code assistants, including how users can disambiguate programs or refine generated code. Our study diverges from these works by evaluating AI programming assistants that are widely used in practice by developers rather than evaluating these tools in laboratory settings. In particular, we examine tools based on the transformer neural network architecture [58], such as GitHub Copilot and Tabnine. Transformer-based tools have shown strong performance in working with both natural language and code inputs [59] compared to other types of these tools.

Researchers have performed user studies on transformer-based AI programming assistants [e.g., 31, 60]. Both studies found users may have trouble expressing the intent in their queries. In particular, Xu et al. [60] revealed a challenge their users faced was that the tool assumed background knowledge in underlying modules or frameworks.

Also related to our study are usability studies on how users are using GitHub Copilot in practice. Vaithilingam et al. [56] performed a user study of GitHub Copilot with 24 participants, where they found users struggled with understanding and debugging the generated code. In a user study with 20 participants, Barke et al. [12] found that developers used GitHub Copilot in two different modes–when they do not know what to do and explore different options (i.e., *exploration mode*), or when they do know what to do but use GitHub Copilot to complete the task faster (i.e., *acceleration mode*)–and that users are less willing to modify suggestions. Meanwhile, Mozannar et al. [44] identified 12 core activities associated with using GitHub Copilot, such as verifying suggestions, looking up documentation, and debugging code, which was then validated on a user study with 21 developers. Finally, Ziegler et al. [62] performed a large-scale user study of GitHub Copilot. They analyzed telemetry data from the model and 2,631 survey responses on developers' perceived productivity with the tool. They reported that 23.3%, 27.9%, and 28.8% of GitHub Copilot's suggestions were accepted for TypeScript, JavaScript, and Python respectively, and 22.2% for all other languages. We extend their user study by performing a large scale study with a focus on the usability challenges of many AI programming assistants, including GitHub Copilot, which provides possible explanations for their findings.

Other works have studied various design aspects of AI programming assistants. For instance, Vaithilingam et al. [55] suggested six design principles of inline code suggestions from AI programming assistants, such as having glanceable suggestions. With the recent

popularity of transformer-based chatbots, such as ChatGPT [1], recent work [e.g., 48, 49] has investigated developers' interactions with conversational chatbots. For example, Ross et al. [49] find that developers are initially skeptical of chatbot programming assistants, but are hopeful about their ability to improve their productivity after using them.

Many of the user studies enumerate potential usability challenges of using AI programming assistants. However, it is unclear to what extent the enumerated challenges are important to developers in practice. Therefore, our study validates and extends these works by quantifying to what extent these usability challenges are encountered by developers in practice. Compared to prior work, we also investigate a larger number of these tools and have a broader focus on usability of both the tools and the tool's outputted code.

3 METHODOLOGY

3.1 Participants

We recruited a large number of participants in order to elicit a diverse range of programming experiences.

Sampling strategy. We recruited participants by selecting contributors from GitHub repositories, following a sampling strategy similar to prior work [28, 38]. To recruit developers who are interested in AI programming assistants, we identified the three projects related to these tools. Two were from GitHub's official GitHub account (i.e., *github/copilot-docs* [4] and *github/copilot.vim* [5]), while one was the official project repository for Tabnine [2], a popular AI programming assistant (i.e., *codota/Tabnine* [3]). To sample participants from the repositories, we used GitHub's GraphQL API [8] to retrieve users who had forked or starred the repositories. 2,329 GitHub users forked, 21,302 GitHub users starred, and 396 GitHub users watched *github/copilot-docs*. 379 GitHub users forked, 6,299 GitHub users starred, and 87 GitHub users watched *github/copilot.vim*. 420 GitHub users forked, 9,594 GitHub users starred, and 133 GitHub users watched *codota/Tabnine*. We then took the set union of the 9 sets of participants, removing all duplicates. This resulted in 33,983 unique GitHub users who had activities associated with the three repositories.

Finally, we filtered the GitHub users by whether they had a publicly available email address, yielding 10,530 unique users who we invited to take the survey. A random sample of 500 users was first sent the survey to verify the quality of the data. Email invitations were sent to the remaining 10,030 users.

Demographics. The Qualtrics survey was sent to all 10,530 GitHub users and received 410 responses, resulting in a response rate of around 4%. This response rate is comparable to other research surveys in software engineering [e.g., 38, 52].

We summarize the attributes of our participants. Questions on their background were optional and thus may not sum up to 410. Overall, participants represented 57 unique countries. They were from Africa ($n = 9$), Asia ($n = 116$), Europe ($n = 77$), North America ($n = 77$), Oceania ($n = 4$), and South America ($n = 13$). They also represented multiple genders, such as man ($n = 280$), woman ($n = 8$), and non-binary ($n = 7$). Participants programmed under a variety of contexts, including for their profession as a software engineer ($n = 203$) or an end-user developer ($n = 82$), an open-source project

> SURVEY QUESTIONS
> - For this software project, estimate what percent of your code is written with the help of the following code generation tools.
> - For each of the following reasons why you use code generation tools in this software project, rank its importance.
> - For each of the following reasons why you do not use code generation tools, rank its importance.
> - For your software project, estimate how often you experience the following scenarios when using code generation tools.
> - For your software project, estimate how often the following reasons are why you find yourself giving up on code created by code generation tools.
> - ★ What types of feedback would you like to give to code generation tools to make its suggestions better? Why?

Figure 2: A subset of the actual survey questions about the usability of AI programming assistants. An open-ended question is indicated with a star (★). The complete survey instrument is in the supplemental materials [37].

($n = 131$), hobby ($n = 155$), and/or school ($n = 172$). Additionally, they had a wide range of programming experience, ranging from 1 to 41 years, with a median of 6 years. Survey participants reported using a variety of programming languages, such as Python ($n = 199$), JavaScript ($n = 175$), HTML/CSS ($n = 157$), TypeScript ($n = 123$), Bash/Shell ($n = 134$), and/or Java ($n = 84$). They also used AI programming assistants (see Table 1), such as GitHub Copilot, Tabnine, Amazon CodeWhisperer, ChatGPT, and AI programming assistants specific to an organization that was trained on proprietary code.

3.2 Survey

We designed a 15-minute Qualtrics survey to gather data for our research questions and distributed it to participants using the sampling strategy described in Section 3.1. After completing the survey, participants could join a sweepstakes to win one of four $100 electronic gift certificates. All questions in the survey were optional. The study was approved by our institution's institutional review board.

The survey first asked participants how often they used AI programming assistants and whether they had any concerns about using these tools. If the participant used AI programming assistants, they were asked to consider a specific project where they used AI programming assistants and were asked a set of questions regarding their experience with these tools. Survey topics included: why participants use AI programming assistants, how often these tools are used, strategies participants use to make AI programming assistants work better, and why participants give up using generated code. If the participant did not use AI programming assistants, they answered questions regarding why they did not.

The survey also collected information on the participants' programming backgrounds and demographics. Following best practices, we used the HCI Guidelines for Gender Equity and Inclusivity to collect gender-related information [51]. We allowed participants to select multiple responses for questions on gender. A subset of the survey questions is included in Figure 2; the full survey instrument is included in the supplemental materials [37]. While developing the survey, an external researcher reviewed and provided feedback on the survey for clarity and topic coverage.

We conducted pilots of the survey to identify and reduce confounding factors, following the best practices for experiments with human subjects in software engineering research [33]. We piloted drafts of the survey with 11 developers, who were recruited through snowball sampling. These pilots helped clarify wording, ensure data quality, and identify usability factors prior literature may have missed. The survey was updated between each round of feedback. The results from the pilots were not included in the data used in this study.

3.3 Analysis

To analyze the data, we used both quantitative and qualitative techniques. This is because survey questions were largely closed-ended but participants could also select an "other" option, and many questions also provided space to enter open-ended responses. The choices are based on survey piloting and results from prior literature on human evaluations of AI programming assistants (i.e., [12, 15, 17, 18, 29–31, 46, 56, 60, 62]). The first author reviewed these papers and extracted mentions of usability-related issues with AI programming assistants, resulting in a set of usability issues with these tools. This set of usability issues was then de-duplicated and used as choices for closed-ended questions in the survey. Below, we describe our methods in further detail.

Quantitative analysis. To perform quantitative analysis on the closed-ended questions, we followed best practices for statistical analysis techniques described by Kitchenham and Pfleeger on how to analyze survey data [32]. In particular, we report the frequencies of how often an item was selected. We also report how frequently participants rated statements as being important or very important, situations as occurring often or always, and feeling concerned or very concerned about a situation. Following best practices [45], we report measurements on perceived frequency to understand the importance of a situation rather than an accurate measurement on how frequently a situation occurs.

Qualitative analysis. For qualitative analysis, the first two authors performed multiple rounds of open coding on each set of responses to the open-ended questions. We used general best practices [26, 50], such as interpreting generated codes as itemized claims about the data to be investigated in other work and shuffling responses to reduce any ordering effects that could emerge during coding.

In the first round of coding, the authors open-coded the same initial set of 100 responses. Each response was labeled with zero or more codes. Each code was given a unique identifier and brief description. Then, the authors convened to discuss the resulting set of codes and their scopes. To merge the codes, the authors identified codes with similar themes and merged them into a single code in the shared codebook. The remaining codes were then added to or removed from the codebook by a unanimous vote between the two authors. Coding disagreements most frequently occurred due to different scopes of codes rather than the meaning of participants' statements. The authors then jointly performed a second round of coding on the original data by applying codes from the shared codebook onto each instance based on a unanimous vote. We do not report IRR because following best practices from Hammer and Berland [26], each instance's codes were unanimously agreed upon and because the codes were the process, not the product [42].

4 USAGE CHARACTERISTICS

We present our findings on how developers use AI programming assistants. We first present quantitative results on how developers use these tools (Section 4.1) and developers' motivations for using them (Section 4.2). To elucidate the quantitative results, we describe qualitative results on successful use cases (Section 4.3) and users' strategies to generate helpful output (Section 4.4).

4.1 Usage patterns

In the survey, we asked participants to describe how often they used AI programming assistants and how much of their code was written with the help of these tools (see Table 1). We report the median percentage of code written by each tool's users. Unsurprisingly, GitHub Copilot was the most popular AI programming assistant by the number of users (306), with 46% of its users reporting using the tool frequently. GitHub Copilot's users reported writing 30.5% of their code with the help of the tool. However, organization-specific AI programming assistants helped write the largest percentage of code for survey participants (37%). Interestingly, we found that chatbot-based programming assistants (i.e., ChatGPT) were self-reported by 25 participants. Even though ChatGPT had the highest proportion of frequent users (59%), it was the penultimate tool in terms of the amount of code it helped write for survey participants (20%).

4.2 Motivation

Motivation for using. Participants who reported using an AI programming assistant on at least a monthly basis reported their motivations for using these tools (see Table 2-A). Participants largely used these tools for convenience in programming–86%, 76%, and 68% of participants cited an important motivation for using these tools was autocompletion (*M1*), finishing tasks faster (*M2*), and skipping going online to recall syntax respectively (*M3*). On the other hand, 50% and 36% of participants said an important reason for using these tools was finding potential code solutions (*M4*) or edge cases respectively (*M5*).
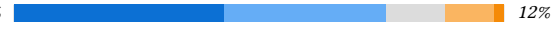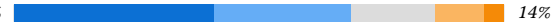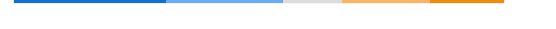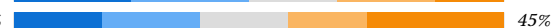
Motivation for not using. Participants who reported not using any AI programming assistant on at least a monthly basis reported their motivations for *not* using these tools (see Table 2-B). Participants seemed to not use these tools because the tools did not provide useful or relevant output. Two important motivations were that the models did not write code that met certain functional or non-functional requirements (*M6*, 54%) and users had difficulty controlling the model (*M7*, 48%). 34% of participants cited these tools not providing helpful suggestions as an important reason for

Table 1: Participants' self-reported usage of popular AI programming assistants. An asterisk (*) denotes a write-in suggestion, which has limited information on its usage distribution. Percentages in italics on the chart ($N\%$) represent the percent of the distribution that reported "Always"/"Often" (left) and "Rarely"/"Tried but gave up" (right).

Tool	# users	Med. % code written	Usage distribution	
Amazon CodeWhisperer	50	5%	*24%*	*61%*
ChatGPT*	25	20%	*59%*	*14%*
GitHub Copilot	306	30.5%	*46%*	*30%*
TabNine	118	20%	*27%*	*66%*
Organization-specific code generation tool trained on proprietary code	54	37%	*29%*	*56%*

■ Always (1+ times daily) ■ Often (once daily) ■ Sometimes (weekly) ■ Rarely (monthly) ■ Tried but gave up

Table 2: Participants' motivations for using and not using AI programming assistants.

|	Motivation	Distribution	
	A. For using		
M1	To have an autocomplete or reduce the amount of keystrokes I make.	*86%*	*6.2%*
M2	To finish my programming tasks faster.	*76%*	*12%*
M3	To skip needing to go online to find specific code snippets, programming syntax, or API calls I'm aware of, but can't remember.	*68%*	*14%*
M4	To discover potential ways or starting points to write a solution to a problem I'm facing.	*50%*	*24%*
M5	To find an edge case for my code I haven't considered.	*36%*	*44%*
	B. For not using		
M6	Code generation tools write code that doesn't meet functional or non-functional (e.g., security, performance) requirements that I need.	*54%*	*34%*
M7	It's hard to control code generation tools to get code that I want.	*48%*	*36%*
M8	I spend too much time debugging or modifying code written by code generation tools.	*38%*	*45%*
M9	I don't think code generation tools provide helpful suggestions.	*34%*	*46%*
M10	I don't want to use a tool that has access to my code.	*30%*	*51%*
M11	I write and use proprietary code that code generation tools haven't seen before and don't generate.	*28%*	*59%*
M12	To prevent potential intellectual property infringement.	*26%*	*66%*
M13	I find the tool's suggestions too distracting.	*26%*	*51%*
M14	I don't understand the code written by code generation tools.	*16%*	*76%*
M15	I don't want to use open-source code.	*10%*	*89%*

■ Very important ■ Important ■ Moderately important ■ Slightly important ■ Not important at all

not using them (**M9**). By having code that was not useful, users engaged in the time-consuming process of modifying or debugging code (**M8**). This was also a salient motivation, as 38% of participants rated it as an important reason for not using these tools. Participants resonated the least with not understanding generated code (**M14**) and not wanting to use open-source code (**M15**), as 76% and 89% of participants rated them as not important.

4.3 Successful use cases

Survey participants described situations where they were most successful in using AI programming assistants. We found 10 types of situations, which we describe below. We report the frequencies of the codes using the multiplication symbol (×).

Repetitive code *(78×).* Participants were successful in using the AI programming assistants to generate repetitive code, such as *"boilerplate [code]"* (P165), *"repetitive endpoints for crud"* (P164), and *"college assignments"* (P265) that had repeated functionality or were common programming tasks. This was the most frequent code in our data.

❝ *Complete code that is highly repetitive but cannot be copied and pasted directly."* (P195)

Code with simple logic *(68×).* Consistent with prior work [56], participants reported using AI programming assistants to successfully generate code with simple logic. This was the second most mentioned code in the dataset. Examples include *"small independent utils functions"* (P155), *"sorting algorithms"* (P177), and *"small functions like storing the training model into local file systems"* (P255).

Participants said that having the tool write more complex logic often resulted in it not working:

> 66 *It however, fails assisting me when I'm writing a more complex algorithm (if not well known)."* (P28)

Autocomplete *(28×).* We found participants also utilized AI programming assistants to do short autocompletions of code, which is associated most with *acceleration mode* usages of these tools [12]. This code was the third most mentioned code in the dataset.

> 66 *I wrote* `s_1, a_1 = draw('file_1')`, *then I want to complete* `s_2, a_2 = draw('file_2')`. *After I type* `s_2`, *copilot helps me [with] this line."* (P240)

Quality assurance *(21×).* Participants reported using AI programming assistants for quality assurance, such as *"[generating] useful log messages"* (P212) and *"[producing] a lot of test cases quickly"* (P356). As found in prior work [12], participants used these tools to consider edge cases:

> 66 *This tool can almost instantly generate the code with good edge case coverage."* (P160)

Proof-of-concepts *(20×).* Similar to prior work [12, 56, 60], participants mentioned that using AI programming assistants helped with brainstorming or building proof-of-concepts by helping generate multiple implementations for a given problem. Participants relied on this when they *"need[ed] another solution"* (P193) or *"only [had] a fuzzy idea about how to approach it"* (P163), so these tools also helped with provide a starting implementation to work off of:

> 66 *We most use these tools at the beginning as a start point or when we get stuck."* (P21)

Learning *(19×).* Study participants also utilized these tools when *"learning new programming languages"* (P197) or *"new libraries"* (P140) they had limited to no experience with, rather than using online documentation [47] or video tutorials [40]. Participants reported that it was especially useful when a project used multiple programming languages:

> 66 *Since [the codebase] is a polyglot project with golang, java, and cpp implementations, I benefit a lot from...polyglot support."* (P40)

Recalling *(19×).* As found in prior work [60], participants leveraged AI programming assistants to find syntax of programming languages or API methods that they were familiar with, but could not recall. This replaced the traditional methods of using web search [47] to find online resources like StackOverflow [27, 41] to recall code snippets or syntax:

> 66 *To skip needing to go online to find...code snippets."* (P179)

Efficiency *(18×).* Study participants also echoed prior work [62] by describing an AI programming assistant's ability to *"speed up...work"* (P246). Participants reported that it helped them to "stay in the flow", an important aspect of developer productivity [23]:

> 66 *Code generation will help the process go smoother and does not introduce unwanted interruptions."* (P166)

Documentation *(6×).* A few participants used AI programming assistants to generate documentation. One participant noted generating documentation helped with collaboration:

> 66 *I mainly use it to...annotate my code for my colleagues."* (P258)

Code consistency *(4×).* A few participants used these tools to improve style consistency in a codebase, which is a factor developers consider while making implementation decisions [36]. Participants applied these tools to *"[follow]...standard clean code style"* (P156), such as *"proper indentation in different [programming] languages"* (P50). It also helped with consistency within a project:

> 66 *To ensure consistency of code by quickly referencing sources created within the project."* (P36)

4.4 User input strategies

Finally, we asked participants to enumerate strategies they used to get AI programming assistants to output the best answers. We found 7 strategies, which we describe below.

Clear explanations *(99×).* The most popular strategy participants reported was providing very clear and explicit explanations of what the code should do in comments, which is a major activity while using AI programming assistants [44]. Participants wrote *"a docstring which tells the function of the function"* (P22) or *"outlining preconditions and postconditions and [writing a]...test case* (P356). Others opted to *"use words (tags) rather than sentences"* (P206).

> 66 *Be incredibly specific with the instructions and write them as precisely as I would for a stupid collaborator."* (P170)

No strategy *(44×).* Many participants reported not employing any strategy, as they found AI programming assistants to provide helpful suggestions without needing to perform specific actions.

> 66 *Nothing, I just review the suggestions as they come up."* (P268)

Adding code *(36×).* Participants often reported consciously writing additional code as context for the AI programming assistant to later complete. Participants did this to *"make some context"* (P117) and provide a *"hint to [improve] the code generation"* (P93).

> 66 *Write a partial fragment of the code I think is...correct."* (P166)

Following conventions *(24×).* Many participants also resorted to following common conventions, such as *"communities' rules and design patterns"* (P157), *"well-named variables"* (P366), or *"[giving] the function a very precise name"* (P254). Participants even viewed the generated code as a source of code with proper conventions:

> 66 *Proper naming conventions also helps... Since these tools learn from excellent code, I should also write code that follows conventions, this can make tools easily find the right result."* (P224)

Breaking down instructions *(18×).* Participants also reported breaking down the code logic or prompts into shorter, more concise statements by explaining the functionality step-by-step. Examples include *"break[ing] the problem into smaller parts"* (P166) and *"split[ting] the sentence to be shorter"* (P167).

> 66 *You have to break down what you're trying to do and write it in steps, it can't do too much at once."* (P126)

Existing code context *(18×).* Participants developed mental models of these tools [15], as they reported leveraging existing code as additional data for the AI programming assistant to use, such as by *"opening files for context"* (P274). Participants reported specifically using AI programming assistants only when there was sufficient existing code context:

> 66 *I try to use it at advanced stages of my project, where it can give better suggestions based on my project's history."* (P111)

Prompt engineering (13×). Some participants iteratively changed their inputs to query the tool. such as *"changing the prompt/comment to simpler sentences"* (P82) or *"tweak[ing] the comments...to [be more] interactive...for the specific task"* (P80).

> 66 *If the code generated does not satisfy me, I will edit the comments."* (P150)

❯ **Key findings:** Participants who were GitHub Copilot users reported a median of 30.5% of their code being written with its help (#1). The most important reasons for using AI programming assistants were for autocomplete, completing programming tasks faster, or skipping going online to recall syntax (#2). Participants successfully used these tools to generate code that was repetitive or had simple logic. Participants reported the most important reasons for not using AI programming assistants were because the code that the tools generated did not meet functional or non-functional requirements and because it was difficult to control the tool (#3).

---

5 USABILITY OF AI PROGRAMMING ASSISTANTS

In this section, we present our findings on what challenges developers encounter while interacting with AI programming assistants. We first report the frequency of usability issues (Section 5.1). To better understand these challenges, we explore the practices of users in understanding (Section 5.2), evaluating (Section 5.3), modifying (Section 5.4), and giving up (Section 5.5) on outputted code.

5.1 Usability issues

We asked participants to rate how frequently certain usability issues occurred while they used AI programming assistants (see Table 3-A). The biggest challenges participants reported facing were not knowing what part of the input influenced the output (*S1*), giving up on using outputted code (*S2*), and having trouble controlling the model (*S3*), as 30%, 28%, and 26% of participants encountered these situations often. Meanwhile, participants had the least trouble with understanding the code generated by the tool (*S9*)–only 5.6% of participants frequently encountered this issue, despite it being discussed in prior literature [56].

5.2 Understanding outputted code

We asked participants who reported having trouble understanding the outputted code to rate the reasons why (see Table 3-B). 25% of participants said it was often because the outputted code used unfamiliar APIs (*S10*). Meanwhile, 23% and 19% of participants stated it was often due to the code being too long to read quickly (*S11*) and the code having too many control structures (*S12*) respectively.

5.3 Evaluating outputted code

We asked participants how they evaluated generated code (see Table 3-C). The order of the evaluation methods by frequency closely related to how time-consuming each method was reported to be. Participants often reported using quick visual inspections of the code (*S13*, 74%), static analysis tools like syntax checkers (*S14*, 71%), executing the code (*S15*, 69%), and examining the details of

the outputted code's logic in depth (*S16*, 64%). However, participants reported frequently consulting API documentation at a lower rate (*S17*, 38%).

5.4 Modifying outputted code

We asked participants how they modified the generated code (see Table 3-D). Participants overall reported regularly having success with modifying the outputted code (*S18*, 63%), most often by changing the generated code itself (*S19*, 62%) rather than by changing the input context (*S20*, 40%). Additionally, a smaller proportion of participants (*S21*, 44%) often used the generated code as-is.

5.5 Giving up on outputted code

We asked participants who reported giving up on outputted code to rate the reasons why (see Table 3-E). The two major reasons were that the generated code did not perform the intended action (*S22*) and because the code did not meet functional or non-functional requirements (*S23*)–43% and 34% of participants frequently encountered these situations respectively. The least salient reasons why participants gave up on using generated code was that they did not understand the outputted code (*S27*), that they found the output too complicated (*S28*), and that the outputted code used unfamiliar APIs (*S29*). This was regularly encountered by 12%, 10%, and 10% of participants respectively.

❯ **Key findings:** The most frequent usability challenges participants reported encountering were understanding what part of the input caused the outputted code, giving up on using the outputted code, and controlling the tool's generations (#4). Participants most often gave up on outputted code because the code did not perform the intended action or did not account for certain functional and non-functional requirements (#5).

---

6 ADDITIONAL FEEDBACK

We present our results on what additional feedback developers have to improve their experiences with AI programming assistants. We discuss general concerns that participants had about these tools (Section 6.1) and participants' responses on how they would improve them (Section 6.2).

6.1 General concerns

We asked all participants to rate their level of concern on issues related to AI programming assistants (see Table 4), which were derived from Cheng et al. [15] and our survey pilots. Participants overall seemed most concerned about their own and others' intellectual property–they most frequently described feeling concerned over AI programming assistants producing code that infringed on intellectual property (*C1*, 46%) and the tools having access to their code (*C2*, 41%). In contrast, participants seemed less worried about concerns more specific to working in commercial contexts; 29% of participants reported feeling concerned about AI programming assistants not generating proprietary APIs (*C3*) as well as generating outputted code that contained open-source code (*C4*).

Table 3: How frequently participants report usability issues occurring while using AI programming assistants.

	Situation	Distribution	
	A. Usability issues		
S1	I don't know what part of my code or comments the code generation tool is using to make suggestions.	30%	48%
S2	I give up on incorporating the code created by a code generation tool and write the code myself.	28%	35%
S3	I have trouble controlling the tool to generate code that I find useful.	26%	48%
S4	I find the code generation tool's suggestions too distracting.	23%	44%
S5	I have trouble evaluating the correctness of the generated code.	23%	52%
S6	I have difficulty expressing my intent or requirements through natural language to the tool.	22%	36%
S7	I find it hard to debug or fix errors in the code from code generation tools.	17%	61%
S8	I rely on code generation tools too much to write code for me.	15%	67%
S9	I have trouble understanding the code created by a code generation tool.	5.6%	45%
	B. Reasons for not understanding code output		
S10	The generated code uses APIs or methods I don't know.	25%	43%
S11	The generated code is too long to read quickly.	23%	45%
S12	The generated code contains too many control structures (e.g., loops, if-else statements).	19%	44%
	C. Methods of evaluating code output		
S13	Quickly checking the generated code for specific keywords or logic structures	74%	10%
S14	Compilers, type checkers, in-IDE syntax checkers, or linters	71%	14%
S15	Executing the generated code	69%	14%
S16	Examining details of the generated code's logic in depth	64%	15%
S17	Consulting API documentation	38%	28%
	D. Methods of modifying code output		
S18	When a code generation tool outputs something I don't want, I'm able to modify it to something I want.	63%	12%
S19	I successfully incorporate the code created by a code generation tool by changing the generated code.	62%	9.1%
S20	I use the code created by a code generation tool as-is.	44%	24%
S21	I successfully incorporate the code created by a code generation tool by changing the code or comments around it and regenerating a new suggestion.	40%	30%
	E. Reasons for giving up on code output		
S22	The generated code doesn't perform the action I want it to do.	43%	22%
S23	The generated code doesn't meet functional or non-functional (e.g., security, performance) requirements that I need.	34%	28%
S24	The generated code's style doesn't match my project's.	22%	48%
S25	The generated code contains too many defects.	21%	42%
S26	The generated code uses an API I know, but don't want to use.	17%	55%
S27	I don't understand the generated code well enough to use it.	12%	69%
S28	The generated code is too complicated.	10%	68%
S29	The generated code uses an API I don't know.	10%	59%

■ Always ■ Often ▢ Sometimes ■ Rarely ■ Never

6.2 Improving AI programming assistants

We asked participants to describe feedback they would provide to AI programming assistants to make their output better. We identified 8 types of feedback, which we elaborate on below.

User feedback *(52×).* Most frequently, participants wanted to provide feedback to the AI programming assistant for it to learn from. Some wanted to correct the outputted code as feedback, while others wanted to teach the model their personal coding style. While some participants wanted to directly provide feedback in natural language, others preferred code: *"Maybe...code [of] my correct answer. I don't...want to explain in natural language."* (P201). Meanwhile, others suggested rating the output with *"like/dislike buttons...to not get distracted from actual work"* (P52).

Table 4: Participants' level of concern on issues related to AI programming assistants.

	Concern	Distribution	
C1	Code generation tools produce code that infringe on intellectual property.	46%	32%
C2	Code generation tools have access to my code.	41%	38%
C3	Code generation tools do not generate proprietary APIs or code.	29%	46%
C4	Code generation tools may produce open-source code.	29%	53%

■ Very concerned ■ Concerned ■ Moderately concerned ■ Slightly concerned ■ Not concerned at all

❝ *Automatic feedback based on code correction made by the developer.*" (P57)

❝ *Maybe more personaliz[ation]...I have my own code style, so I will need...time to modify the code into my style.*" (P102)

Better understanding of code context *(20×).* Participants also reported wanting AI programming assistants to have additional understanding of code context, such as learning from *"context from other files on the same workspace"* (P12). Others wanted these tools to have a deeper understanding of certain nuances behind APIs and programming languages, such as when *"the code is using [a] deprecated API"* (P88).

❝ *To be able to better describe the contexts of our projects during creation. For a better understanding of our code generator.*" (P208)

Tool configuration *(17×).* A few participants wanted to change the tool's settings. This included *"distinguish[ing when to do] long code generation and short code [generation]"* (P240), having *"adjustable parameters"* (P177), or reducing the frequency of suggestions. This could assist the model in adapting to whether the developer was in *acceleration mode*–associated with short completions– or *exploration mode*–associated with long completions [12].

❝ *I'd like to be able to ask it to calm down sometimes instead of constantly trying to suggest random stuff.*" (P122)

Natural language interactions *(16×).* Some participants wanted opportunities for interaction via natural language. Inspired by Chat-GPT [1], several participants mentioned chat-based interactions: *"would be nice if we could give feedback to it like how we chat with chatGPT"* (P39).

❝ *To comment on the resulting code the tool generates, and let the tool reiterate from such previously generated result, but with my comments.*" (P166)

Code analysis *(13×).* As discussed in prior work [12], some participants also wanted further analysis on the generated code for functional and syntactic correctness, as *"[making] any basic grammatical mistakes or spelling mistakes...would be considered unreliable"* (P105).

❝ *Add extra checks to outputted code to ensure it resembles the input given and that the outputted code is complete and can be run. Often the outputted code that I am given is incomplete, lacks the ability to run or [be] tested immediately.*" (P158)

Explanations *(11×).* Some participants wanted explanations for additional context of the generated code, such as *"sourcing...the suggestions"* (P58) or *"link[ing] direct[ly] to documentation"* (156).

❝ *These tools must show where the code snippet comes from and include the code link of snippet, license, author name if available for better references for that specific code.*" (P281)

More suggestions *(9×).* Consistent with prior work [12], a few participants wanted to have the model regenerate or provide more than one suggestion, such as by having the *"possibility to shuffle between code snippets"* (P177).

❝ *Maybe multiple suggestions and then I pick the best.*" (P149)

Accounting for non-functional requirements *(8×).* Some participants requested AI programming assistants to generate code that addressed non-functional requirements, such as *"time complexity"* (P191). Other participants wanted more readable code:

❝ *Sometimes AI suggest code [with] one lines or short hand logic, which is difficult to read and understand.*" (P98)

> ❯ **Key findings:** Participants were most concerned about potentially infringing on intellectual property and having a tool have access to their code. Participants reported wanting to improve AI programming assistants' output by having users directly provide feedback to correct or personalize the tool or by teaching the underlying model to have a better understanding of code context (#6). They also wanted more opportunities for natural language interaction with these tools.

7 THREATS TO VALIDITY

Internal validity. Memory bias may influence the internal validity of the study, as the survey questions required participants to recall their experiences with AI programming assistants. We addressed this threat by asking participants to consider their experiences with these tools with respect to a specific project in order to ground participant responses with a concrete experience.

Study participants may also misunderstand the wording of some of the survey questions. To reduce this threat, we piloted the survey 11 times with developers with a focus on the clarity of the survey questions and updated the survey based on their feedback.

External validity. Any empirical study may have difficulties in generalizing [21]. To address this, we sample from a set of participants who are diverse in terms of geographic location and software engineering experience. However, our study may still struggle with sampling bias. This is because we sampled from GitHub projects that were related to AI programming assistants, such as GitHub Copilot and Tabnine. Thus, our sample largely represents people who are enthusiastic about these tools. Further, our sample does

not specifically sample individuals who are not interested in AI programming assistants, so this population may be underrepresented within our study. Therefore, our sample may not be representative of all users of AI programming assistants.

Because the survey was deployed in January 2023, participants provided responses based on their experiences with AI programming assistants at the time. Thus, some aspects may not be relevant to future versions of these tools that perform differently.

Construct validity. Many survey questions asked participants to provide subjective estimates of the frequency of encountering certain situations or using specific tools. Thus, these estimates may not be accurate. Collecting *in-situ* data in future studies, such as in [44] and [62], would be more appropriate to evaluate the frequency of these events. We report measurements on perceived frequency as a proxy for the importance of each usability challenge–following best practices in human factors in software engineering research [45]–rather than the ground truth on the usability challenge's frequency.

Ethical Considerations. An important component of this research study was gathering a sufficiently large number of responses to our survey. Our goal was to receive 385 survey responses, so that we could achieve a 95% confidence level with a 5% margin of error with our sample.

Given our recruitment method needed to result in a large number of responses from programmers, traditional methods of recruitment used in smaller-scale user studies were not practical for our study. Snowball sampling was unlikely to yield the scale of responses that were necessary, while recruiting student programmers from our institution or using traditional crowd-sourcing platforms (e.g., Amazon Mechanical Turk) would not target a representative population of developers. Therefore, we followed prior research in the past 10 years published in top software engineering conferences ([e.g., 24, 25, 28, 38]) that utilized large-scale participant recruitment from populations on GitHub that achieved a sufficient number of survey responses. However, community standards following this recruitment method have recently shifted. Recent work from Tahaei and Vaniea [54] has noted limitations in this method, as mining emails from GitHub is not encouraged by the platform. We advise future work to not use our recruitment strategy and instead follow Tahaei and Vaniea [54]'s recommendation in using the crowdsourcing platform, Prolific [7], as it is a more sustainable way of gathering survey responses from developers at scale.

8 DISCUSSION & FUTURE WORK

The findings from our study overlap with prior usability studies of AI programming assistants [e.g., 12, 13, 56, 62]. In this section, we discuss these works in relation to our results. This produces several implications for future work, which we elaborate on further.

8.1 Implications

Acceleration mode versus exploration mode. Barke et al. [12] found that users of AI programming assistants, such as GitHub Copilot, use the tools in two main modes: *acceleration mode*, where the developer knows what code they would like to write and uses the tool to complete the code more quickly, or *exploration mode*, where

the developer is unsure of what to write and would like to visit potential options. Our results support this theory of AI programming assistant usage, as both *acceleration mode* and *exploration mode* emerge as themes in our results. In particular, these modes appear when developers use AI programming assistants (e.g., **repetitive code**, **code with simple logic**, **autocomplete**, **recalling** versus **proof-of-concepts**), why developers use these tools (e.g., autocompleting (**M1**), finishing programming tasks faster (**M2**), not needing to go online to find code snippets (**M3**) versus discovering potential ways to write a solution (**M4**), finding an edge case (**M5**)), and how developers interacted with the tool to produce better suggestions (e.g., **no strategy**, **following conventions**, **adding code** versus **clear explanations**).

We further augment Barke et al. [12]'s theory by finding that aspects related to *acceleration mode* are represented within our data more than aspects related to *exploration mode*. For example, **repetitive code** (78×), **code with simple logic** (68×), and **autocomplete** (28×), all occur more frequently than **proof-of-concepts** (20×) as situations when participants successfully used AI programming assistants. Additionally, participants rated **M1** (86%), **M2** (76%), and **M3** (68%) to be important reasons for using AI programming assistants at higher rates than **M4** (50%) and **M5** (36%). This suggests that developers may value *acceleration mode* over *exploration mode*.

Chatbots as AI programming assistants. Our results also indicate a potential for AI programming assistant users to rely more on chat-based interactions, following the recent rise of powerful chatbots such as ChatGPT [1]. 6% of our participants explicitly wrote that they used ChatGPT as an AI programming assistant, and a popular feedback was to provide more opportunities for **natural language interactions**. While recent work shows promise in this method of interaction with AI programming assistants [48, 49], it also raises additional questions of when these interaction methods should be applied. Understanding *when* developers should rely on these interactions is fundamentally a usability question that cannot be addressed through technological advances alone, as it is unclear how to balance this interaction mode with users' cognitive load. While participants seemed to prefer *acceleration mode* over *exploration mode*, our results also indicate that some users may be amenable to using chat; this is because providing **clear explanations**, often in natural language, was the most cited strategy to having AI programming assistants produce the best output.

Developers using AI programming assistants to learn APIs and programming languages. The findings from our study indicate the potential for developers using AI programming assistants to learn APIs and programming languages. Learning is a fundamental action in software engineering [22] and is independent of any technological innovation. Further, it is an important skill for developers [11, 34, 35, 38]. While developers previously used online resources, such as documentation [47], StackOverflow [27, 41], or blogs [53] to learn how to use new technologies, our study participants often favored AI programming assistants over these resources for both **recalling** and **learning** syntax of APIs and programming languages.

Aligning AI programming assistants to developers. Our results indicate that there are several opportunities in aligning AI programming

assistants to the needs of developers. Giving up on incorporating code (**S2**) was the most common usability issue encountered and it often occurred because the code did not perform the correct action (**S22**). Future work could mitigate this issue by designing new metrics (e.g., [19]) to increase developer-tool alignment.

Further, one emergent theme to align these tools with developers is by giving developers more control over the tools' outputs. In our study, the most frequent usability issues encountered were not knowing why code was outputted (**S1**, 30%) and having trouble controlling the tool (**S3**, 26%). Participants also often reported not using these tools due to difficulties controlling the tool (**M7**, 48%). Additionally, the most frequent feedback provided was accepting **user feedback** to correct the tool. Thus, future work should investigate techniques to allow users to better control AI programming assistants, such as through interactive machine learning approaches [10].

Another theme that emerged was the need for AI programming assistants to account for non-functional requirements in the generation. It was mentioned within the feedback that study participants had for the tools (**accounting for non-functional requirements**) and was a reason why participants did not use them (**M6**, 54%) or gave up on generated code (**S23**, 34%). Therefore, future work should investigate avenues for incorporating non-functional requirements–such as readability and performance–into the generation, which could help increase developers' adoption of these tools. One such example is GitHub's recent project, Code Brushes [9].

8.2 Takeaways

These implications affect both software engineering researchers and practitioners. Below we describe how our findings apply to these populations and discuss opportunities for future work.

For practitioners & tool users. Our findings point to strategies for practitioners to use AI programming assistants more efficiently, which could potentially boost productivity. For instance, software practitioners could make additional efforts to provide **clear explanations** to prompt the AI programming assistant effectively. Practitioners could consider combining this with **adding code** or **following conventions** (e.g., programming conventions) to get the highest quality output possible.

Additionally, our results reveal new use cases of AI programming assistants for practitioners. Rather than using these tools for only autocompletion, software practitioners could consider using them for **quality assurance** (e.g., generating test cases) as well as learning new APIs or programming languages.

For researchers & tool creators. The results from our study reveal several interesting directions for future research, which could be incorporated into AI programming assistants. For example, given participants' reliance on ChatGPT and **natural language interactions**, future work could investigate methods for supporting chat-based interactions without impacting developers' efficiency and flow while programming [23]. Additionally, future work could investigate how developers learn new technologies with AI programming assistants and design experiences that help support developer **learning**.

Another line of research is to study how to improve AI programming assistants' alignment with developers. This is unlikely to be resolved entirely through modeling improvements, as human developers must be able to articulate requirements and evaluate solutions for any given problem. However, this is challenging, as software design and implementation are notoriously complex. Software solutions and problems can co-evolve with one another [57], and software design knowledge can be implicit [36]. Thus, facilitating ways for developers to explicitly describe their software design knowledge to these tools is a challenge to address.

Finally, future work should also investigate new interaction techniques to support *acceleration mode* specifically, given participants' emphasis on this type of usage of AI programming assistants. Following design recommendations for generative AI in creative writing contexts [16], these interaction techniques should require minimal cognitive effort for developers to prevent distracting them from their tasks. Study participants described favoring implicit interactions with AI programming assistants over explicit ones:

> 💬 *Automatic feedback. The tool knows whether I choose...to apply its suggestions. Because it won't distract me.*" (P246)

> 💬 *Feedback...is important, but I'm not sure I want to invest time in "teaching" the tool.*" (P111)

9 CONCLUSION

In this study, we investigated the usability of AI programming assistants, such as GitHub Copilot. We performed an exploratory qualitative study by surveying 410 developers on their usage of AI programming assistants to better understand their usage practices and uncover important usability challenges they encountered.

We find that developers are most motivated to use AI programming assistants because of the tools' ability to autocomplete, help finish programming tasks quickly, and recall syntax, rather than helping developers brainstorm potential solutions for problems they are facing. We also find that while state-of-the-art AI programming assistants are highly performant, there is a gap between developers' needs and the tools' output, such as accounting for non-functional requirements in the generation.

Our findings indicate several potential directions for AI programming assistants, such as designing interaction techniques that provide developers with more control over the tool's output. To facilitate replication of this study, the survey instrument and codebooks are included in the supplemental materials for this work [37].

REFERENCES

[1] 2023. ChatGPT | OpenAI. Retrieved March 11, 2023 from https://chat.openai.com/.
[2] 2023. Code faster with AI code completions | Tabnine. Retrieved March 11, 2023 from https://www.tabnine.com/.
[3] 2023. codota/TabNine - AI code completions. Retrieved March 11, 2023 from https://github.com/codota/TabNine/.
[4] 2023. github/copilot-docs - Documentation for GitHub Copilot. Retrieved March 11, 2023 from https://github.com/github/copilot-docs/.
[5] 2023. github/copilot.vim - Neovim plugin for GitHub Copilot. Retrieved March 11, 2023 from https://github.com/github/copilot.vim/.
[6] 2023. GitHub Copilot - Your AI pair programmer. Retrieved March 13, 2023 from https://copilot.github.com/.
[7] 2023. Prolific • Quickly find research participants you can trust. Retrieved September 2, 2023 from https://prolific.co/.
[8] 2023. GitHub GraphQL API - GitHub Docs. Retrieved March 11, 2023 from https://docs.github.com/en/graphql/.
[9] 2023. GitHub Next | Code Brushes. Retrieved March 11, 2023 from https://githubnext.com/projects/code-brushes/.
[10] Saleema Amershi, Maya Cakmak, William Bradley Knox, and Todd Kulesza. 2014. Power to the people: The role of humans in interactive machine learning. AI Magazine 35, 4 (2014), 105–120. https://doi.org/10.1609/aimag.v35i4.2513
[11] Sebastian Baltes and Stephan Diehl. 2018. Towards a theory of software development expertise. In ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). 187–200. https://doi.org/10.1145/3236024.3236061
[12] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2022. Grounded Copilot: How Programmers Interact with Code-Generating Models. arXiv preprint arXiv:2206.15000 (2022).
[13] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. 2022. Taking flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools. Queue 20, 6 (2022), 35–57. https://doi.org/10.1145/3582083
[14] Sarah E Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping distributed hierarchical web data. In ACM Symposium on User Interface Software and Technology (UIST). 963–975. https://doi.org/10.1145/3242587.3242661
[15] Ruijia Cheng, Ruotong Wang, Thomas Zimmermann, and Denae Ford. 2022. "It would work for me too": How online communities shape software developers' trust in AI-powered code generation tools. arXiv preprint arXiv:2212.03491 (2022).
[16] Elizabeth Clark, Anne Spencer Ross, Chenhao Tan, Yangfeng Ji, and Noah A Smith. 2018. Creative writing with a machine in the loop: Case studies on slogans and stories. In International Conference on Intelligent User Interfaces (IUI). 329–340.
[17] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, Zhen Ming, et al. 2022. GitHub Copilot AI pair programmer: Asset or Liability? arXiv preprint arXiv:2206.15331 (2022).
[18] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with Copilot: Exploring prompt engineering for solving CS1 problems using natural language. In ACM Technical Symposium on Computer Science Education (SIGCSE). 1136–1142. https://doi.org/10.1145/3545945.3569823
[19] Victor Dibia, Adam Fourney, Gagan Bansal, Forough Poursabzi-Sangdeh, Han Liu, and Saleema Amershi. 2022. Aligning offline metrics and human judgments of value of AI-pair programmers. arXiv preprint arXiv:2210.16494 (2022).
[20] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-step live programming by example. In ACM Symposium on User Interface Software and Technology (UIST). 614–626. https://doi.org/10.1145/3379337.3415869
[21] Bent Flyvbjerg. 2006. Five misunderstandings about case-study research. Qualitative Inquiry 12, 2 (2006), 219–245. https://doi.org/10.1177/1077800405284363
[22] Denae Ford, Tom Zimmermann, Christian Bird, and Nachiappan Nagappan. 2017. Characterizing software engineering work with personas based on knowledge worker actions. In ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). 394–403. https://doi.org/10.1109/ESEM.2017.54
[23] Nicole Forsgren, Margaret-Anne Storey, Chandra Maddila, Thomas Zimmermann, Brian Houck, and Jenna Butler. 2021. The SPACE of developer productivity: There's more to it than you think. Queue 19, 1 (2021), 20–48. https://doi.org/10.1145/3454122.3454124
[24] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. 2016. Work practices and challenges in pull-based development: The contributor's perspective. In ACM/IEEE International Conference on Software Engineering (ICSE). 285–296. https://doi.org/10.1145/2884781.2884826
[25] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. 2015. Work practices and challenges in pull-based development: The integrator's perspective. In IEEE/ACM International Conference on Software Engineering (ICSE), Vol. 1. 358–368. https://doi.org/10.1109/ICSE.2015.55
[26] David Hammer and Leema K Berland. 2014. Confusing claims for data: A critique of common practices for presenting qualitative research on learning. Journal of the Learning Sciences 23, 1 (2014), 37–46. https://doi.org/10.1080/10508406.2013.802652

[27] James D Herbsleb and Deependra Moitra. 2001. Global software development. IEEE Software 18, 2 (2001), 16–20. https://doi.org/10.1109/52.914732
[28] Yu Huang, Denae Ford, and Thomas Zimmermann. 2021. Leaving my fingerprints: Motivations and challenges of contributing to OSS for social good. In IEEE/ACM International Conference on Software Engineering (ICSE). 1020–1032. https://doi.org/10.1109/ICSE43902.2021.00096
[29] Saki Imai. 2022. Is GitHub copilot a substitute for human pair-programming? An empirical study. In ACM/IEEE International Conference on Software Engineering (ICSE): Companion Proceedings. 319–321. https://doi.org/10.1145/3510454.3522684
[30] Dhanya Jayagopal, Justin Lubin, and Sarah E Chasins. 2022. Exploring the learnability of program synthesizers by novice programmers. In ACM Symposium on User Interface Software and Technology (UIST). 1–15. https://doi.org/10.1145/3526113.3545659
[31] Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. 2022. Discovering the syntax and strategies of natural language programming with generative language models. In ACM CHI Conference on Human Factors in Computing Systems. 1–19. https://doi.org/10.1145/3491102.3501870
[32] Barbara A. Kitchenham and Shari Lawrence Pfleeger. 2008. Personal opinion surveys. In Guide to advanced empirical software engineering, Forrest Shull, Janice Singer, and Dag I. K. Sjøberg (Eds.). Springer, 63–92. https://doi.org/10.1007/978-1-84800-044-5_3
[33] Amy J Ko, Thomas D LaToza, and Margaret M Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. Empirical Software Engineering 20, 1 (2015), 110–141. https://doi.org/10.1007/s10664-013-9279-3
[34] Paul Luo Li, Amy J Ko, and Andrew Begel. 2020. What distinguishes great software engineers? Empirical Software Engineering 25 (2020), 322–352. https://doi.org/10.1007/s10664-019-09773-y
[35] Paul Luo Li, Amy J Ko, and Jiamin Zhu. 2015. What makes a great software engineer? In IEEE/ACM International Conference on Software Engineering (ICSE), Vol. 1. 700–710. https://doi.org/10.1109/ICSE.2015.335
[36] Jenny T Liang, Maryam Arab, Minhyuk Ko, Amy J Ko, and Thomas D LaToza. 2023. A Qualitative Study on the Implementation Design Decisions of Developers. In IEEE/ACM International Conference on Software Engineering (ICSE). 435–447. https://doi.org/10.1109/ICSE48619.2023.00047
[37] Jenny T Liang, Chenyang Yang, and Brad A Myers. 2023. Supplemental Materials to "A Large-Scale Study on the Usability of AI Programming Assistants: Successes and Challenges". https://doi.org/10.6084/m9.figshare.22355017
[38] Jenny T Liang, Thomas Zimmermann, and Denae Ford. 2022. Understanding skills for OSS communities on GitHub. In ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). 170–182. https://doi.org/10.1145/3540250.3549082
[39] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, Luke Zettlemoyer, , and Michael D Ernst. 2017. Program synthesis from natural language using recurrent neural networks. University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03-01 (2017).
[40] Laura MacLeod, Margaret-Anne Storey, and Andreas Bergen. 2015. Code, camera, action: How software developers document and share program knowledge using YouYube. In IEEE International Conference on Program Comprehension (ICPC). 104–114. https://doi.org/10.1109/ICPC.2015.19
[41] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. 2011. Design lessons from the fastest q&a site in the west. In ACM CHI Conference on Human Factors in Computing Systems (CHI). 2857–2866. https://doi.org/10.1145/1978942.1979366
[42] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. 2019. Reliability and inter-rater reliability in qualitative research: Norms and guidelines for CSCW and HCI practice. Proceedings of the ACM on human-computer interaction 3, CSCW (2019), 1–23. https://doi.org/10.1145/3359174
[43] Andrew M McNutt, Chenglong Wang, Robert A DeLine, and Steven M Drucker. 2023. On the design of AI-powered code assistants for notebooks. In ACM CHI Conference on Human Factors in Computing Systems (CHI). 1–16. https://doi.org/10.1145/3544548.3580940
[44] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2022. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. arXiv preprint arXiv:2210.14306 (2022).
[45] Brad A Myers, Amy J Ko, Thomas D LaToza, and YoungSeok Yoon. 2016. Programmers are users too: Human-centered methods for improving programming tools. Computer 49, 7 (2016), 44–52. https://doi.org/10.1109/MC.2016.200
[46] Ben Puryear and Gina Sprint. 2022. GitHub Copilot in the classroom: Learning to code with AI assistance. Journal of Computing Sciences in Colleges 38, 1 (2022), 37–47.
[47] Nikitha Rao, Chetan Bansal, Thomas Zimmermann, Ahmed Hassan Awadallah, and Nachiappan Nagappan. 2020. Analyzing web search behavior for software engineering tasks. In IEEE International Conference on Big Data (Big Data). 768–777. https://doi.org/10.1109/BigData50022.2020.9378083

[48] Peter Robe, Sandeep K Kuttal, Jake AuBuchon, and Jacob Hart. 2022. Pair programming conversations with agents vs. developers: challenges and opportunities for SE community. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 319–331. https://doi.org/10.1145/3540250.3549127

[49] Steven I Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D Weisz. 2023. The programmer's assistant: Conversational interaction with a large language model for software development. In *ACM Conference on Intelligent User Interfaces (IUI)*. 491–514. https://doi.org/10.1145/3581641.3584037

[50] Johnny Saldaña. 2009. *The Coding Manual for Qualitative Researchers*. SAGE Publications.

[51] Morgan Klaus Scheuerman, Katta Spiel, Oliver L Haimson, Foad Hamidi, and Stacy M Branham. 2020. HCI guidelines for gender equity and inclusivity. In *UMBC Faculty Collection*.

[52] Edward Smith, Robert Loftin, Emerson Murphy-Hill, Christian Bird, and Thomas Zimmermann. 2013. Improving developer participation rates in surveys. In *International workshop on cooperative and human aspects of software engineering (CHASE)*. 89–92. https://doi.org/10.1109/CHASE.2013.6614738

[53] Margaret-Anne Storey, Leif Singer, Brendan Cleary, Fernando Figueira Filho, and Alexey Zagalsky. 2014. The (r)evolution of social media in software engineering. *Future of Software Engineering* (2014), 100–116. https://doi.org/10.1145/2593882.2593887

[54] Mohammad Tahaei and Kami Vaniea. 2022. Lessons Learned From Recruiting Participants With Programming Skills for Empirical Privacy and Security Studies. In *International Workshop on Recruiting Participants for Empirical Software Engineering (RoPES)*.

[55] Priyan Vaithilingam, Elena L Glassman, Peter Groenwegen, Sumit Gulwani, Austin Z Henley, Rohan Malpani, David Pugh, Arjun Radhakrishna, Gustavo Soares, Joey Wang, and Aaron Yim. 2023. Towards more effective AI-assisted programming: A systematic design exploration to improve Visual Studio IntelliCode's user experience. In *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

[56] 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 *ACM CHI Conference on Human Factors in Computing Systems (CHI)*. 1–7. https://doi.org/10.1145/3491101.3519665

[57] Hans Van Vliet and Antony Tang. 2016. Decision making in software architecture. *Journal of Systems and Software* 117 (2016), 638–644. https://doi.org/10.1016/j.jss.2016.01.017

[58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[59] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *ACM SIGPLAN International Symposium on Machine Programming (MAPS)*. 1–10. https://doi.org/10.1145/3520312.3534862

[60] Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-IDE code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–47. https://doi.org/10.1145/3487569

[61] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the quality of Github Copilot's code generation. In *International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*. 62–71. https://doi.org/10.1145/3558489.3559072

[62] 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 *ACM SIGPLAN International Symposium on Machine Programming (MAPS)*. 21–29. https://doi.org/10.1145/3520312.3534864