“It’s Weird That it Knows What I Want”: Usability and Interactions with Copilot for Novice Programmers

JAMES PRATHER and BRENT N. REEVES, Abilene Christian University, USA
PAUL DENNY, The University of Auckland, New Zealand
BRETT A. BECKER, University College Dublin, Ireland
JUHO LEINONEN and ANDREW LUXTON-REILLY, The University of Auckland, New Zealand
GARRETT POWELL, Abilene Christian University, USA
JAMES FINNIE-ANSLEY, The University of Auckland, New Zealand
EDDIE ANTONIO SANTOS, University College Dublin, Ireland

Recent developments in deep learning have resulted in code-generation models that produce source code from natural language and code-based prompts with high accuracy. This is likely to have profound effects in the classroom, where novices learning to code can now use free tools to automatically suggest solutions to programming exercises and assignments. However, little is currently known about how novices interact with these tools in practice. We present the first study that observes students at the introductory level using one such code auto-generating tool, Github Copilot, on a typical introductory programming (CS1) assignment. Through observations and interviews we explore student perceptions of the benefits and pitfalls of this technology for learning, present new observed interaction patterns, and discuss cognitive and metacognitive difficulties faced by students. We consider design implications of these findings, specifically in terms of how tools like Copilot can better support and scaffold the novice programming experience.

CCS Concepts: • **Human-centered computing** → **Human computer interaction (HCI)**; **Empirical studies in HCI**; **User studies**; **Natural language interfaces**; **User interface programming**; • **Computing methodologies** → **Artificial intelligence**; • **Social and professional topics** → **Computing education**; **Computer science education**; **CS1**; • **Applied computing** → **Education;**

Additional Key Words and Phrases: AI, Artificial Intelligence, automatic code generation, Codex, Copilot, CS1, GitHub, GPT-3, HCI, introductory programming, large language models, LLM, novice programming, OpenAI

ACM Reference format:
James Prather, Brent N. Reeves, Paul Denny, Brett A. Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. “It’s Weird That it Knows What I Want”: Usability and Interactions with Copilot for Novice Programmers. *ACM Trans. Comput.-Hum. Interact.* 31, 1, Article 4 (November 2023), 31 pages.
https://doi.org/10.1145/3617367

Authors’ addresses: J. Prather, B. N. Reeves, and G. Powell, Abilene Christian University, USA; P. Denny, J. Leinonen, A. Luxton-Reilly, and J. Finnie-Ansley, The University of Auckland, New Zealand; B. A. Becker and E. Antonio Santos, University College Dublin, Ireland.

1073-0516/2023/11-ART4 $15.00
https://doi.org/10.1145/3617367

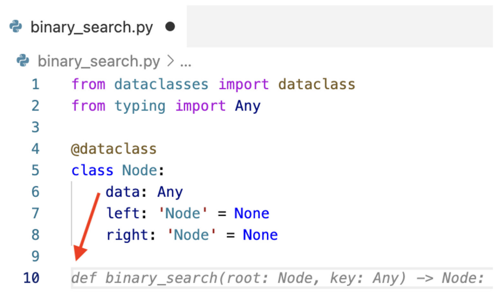
1 INTRODUCTION

Introductory programming courses typically require students to write many small programs [4]. Teachers design programming exercises to facilitate and improve student learning; however, students are not always appropriately oriented to their learning, often focusing on completing tasks as quickly as possible. Therefore, in the context of these programming exercises, teachers and students often have competing user needs. These differing needs converge when students find themselves stuck and unable to complete the tasks required. There is plenty of evidence that students struggle to develop effective plans [97] and to implement plans that are developed [32, 52]. This is a frustrating experience for students [12] that can limit their learning progress, and may result in undesirable behaviors such as copying [48]. To maintain progress and positive learning experiences, an outcome desirable for both teachers and students, there is a need to support students who are stuck [73].

Unfortunately, teachers are not always available to provide this support. Static learning resources such as tutorials, guides, textbooks, and tools such as IDEs do not provide contextualized interactive support. There have been numerous attempts to provide more contextualized help to students learning to program, although effective programming support remains a challenge [53]. Intelligent tutoring systems [29] provide adaptive feedback to students depending on their performance, but such systems typically guide students through pre-constructed tasks and do not support student learning in more general (in-the-wild) environments. More recently, automated hint generation systems have been employed to generate hints for any programming exercises [71]. Although such systems do not require the feedback on tasks to be manually designed, they do need to be deployed in environments that have access to historical student performance data to determine which approaches are more successful for a given exercise [68], which limits their utility [53].

In this article, we explore how students used an LLM tool when engaged in a programming task, from a teaching and learning perspective. We study the use of GitHub Copilot—an IDE plugin powered by the LLM Codex. Copilot is easily accessible to novices, is free for students, and operates as a plug-in directly in popular development environments. Existing work has not explored how students interact with LLM-based tools to support their progress in programming education. This article extends knowledge in this field by presenting the first study that observes students at the first-year university level using Copilot on a typical assignment in an introductory programming course (often generically called "CS1" [13]). In particular, we were interested in capturing the novel experience of interacting with a new tool for the first time. We triangulate observations of novice programmers with interviews that explore their perceptions of the benefits and dangers of this technology for learning.

We find that most students perceived that Copilot would help them write code faster, while also expressing concerns about not understanding the auto-generated code and becoming reliant on the tools—concerns also held by educators [25, 42]. Prior work has demonstrated that student learning is impacted by both cognitive [37], and meta-cognitive processes (such as explicitly monitoring their own progress and reflecting on the effectiveness of their problem solving strategies) [65]. We observed two new interaction patterns. The first was when students guided Copilot by utilizing its auto-generated code prompts, *shepherding* it toward a solution instead of focusing on writing code from scratch and integrating Copilot's suggestions. The second was when some students were moved along by some of Copilot's incorrect suggestions, *drifting* from one to the next and therefore becoming lost. We also observed that students struggled with both cognitive and metacognitive difficulties when using the tool. Finally, we present ethical considerations and design guidelines based on these insights.

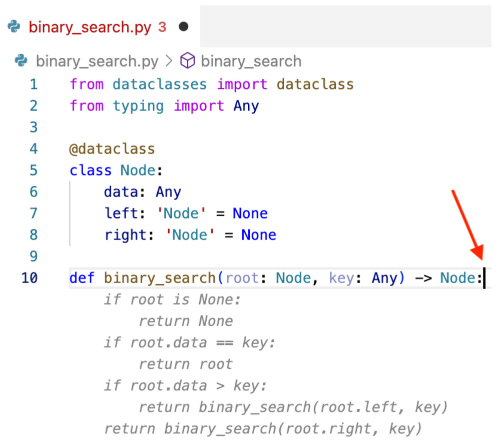


```
from dataclasses import dataclass
from typing import Any

@dataclass
class Node:
    data: Any
    left: 'Node' = None
    right: 'Node' = None

def binary_search(root: Node, key: Any) -> Node:
```

(a) A programmer has entered code up to the cursor (red arrow) at the beginning of line 10 with minimal help from Copilot before this point. The gray italicized code is a Copilot suggestion.



```
from dataclasses import dataclass
from typing import Any

@dataclass
class Node:
    data: Any
    left: 'Node' = None
    right: 'Node' = None

def binary_search(root: Node, key: Any) -> Node:
    if root is None:
        return None
    if root.data == key:
        return root
    if root.data > key:
        return binary_search(root.left, key)
    return binary_search(root.right, key)
```

(b) The programmer has accepted the suggestion in (a) by pressing tab, advancing the cursor to the end of line 10 (red arrow), just after the code that was suggested. The gray italicized code is the next Copilot suggestion.

Fig. 1. Copilot suggesting code to a programmer. Several videos are available at github.com/features/copilot.

1.1 Background

In 2021, OpenAI released Codex [25], a version of GPT-3 trained on billions of lines of Python code from public GitHub repositories [108]. Codex is conversant in both natural and programming languages. It is most proficient in English and Python but can also work in other natural languages such as Spanish and programming languages including Go, JavaScript, Perl, PHP, Ruby, Shell, Swift, and TypeScript [108].

GitHub Copilot uses Codex to suggest code in real-time based on code that has been entered by the user [45]. Copilot was moved out of technical preview in June 2022 and is now available for free to students as a plug-in for IDEs such as Visual Studio Code and JetBrains. Copilot is billed as "Your AI pair programmer" [45]—an intentional reference to pair programming, a well-known software engineering practice [9] that is also used in programming education [75]. Figure 1 shows Copilot in action suggesting code to the programmer as it is being written.

In this study we focus on Copilot, however, there are several other AI code generators available (further discussed in Section 2.1.1). Our work focuses on analyzing how novice programmers use Copilot and learning about novice programmers' experiences with Copilot through interviews.

1.2 Research Questions & Contributions

Our research questions are:

RQ1: How do novices interact with GitHub Copilot when they first encounter it?
RQ2: How do novices perceive their first-time experience of using GitHub Copilot?

The novel contributions of this work are:

(1) We present the first exploration of Copilot use by novices in an undergraduate introductory programming course on a typical learning assignment. We also contribute the first interviews with novices about their initial experiences using Copilot—having never used it (or tools like it) before — to understand both the benefits they perceive as well as their concerns about such tools. While there is some prior work on Copilot's capabilities to solve programming problems at the novice level [42], there has been no work on the tool's *usability* for

novices, nor their perceptions of it. Furthermore, it is likely that capturing these reactions will not be possible in the future as having an entire class, all of whom have never been exposed to AI code generators, will be unlikely.

(2) We contribute new interaction patterns for using LLM-based code generation tools: *drifting* and *shepherding*. These complement and expand upon existing LLM interaction patterns from the literature, such as *exploration* and *acceleration* identified by Barke et al. [8], the "wrestling" that Bird et al. [16] observed in professional developers using Copilot for the first time, as well as aligning with some of the observations made by Vaithilingam et al. [103].

(3) We discuss four design implications for the novice programmer experience with AI code generators such as Copilot.

2 RELATED WORK

In this section we review recent related work on large language models, their use in computing education, and prior user studies of AI code generators.

2.1 Large Language Models

In the field of natural language processing, great progress has been made recently in **large language models** (**LLMs**). These are typically based on a deep learning transformer architecture and often surpass previous state-of-the-art models in most generation tasks. For example, GPT-3 (a text-to-text model) is able to produce text that can be difficult to distinguish from text written by humans [22], and whose applications include summarizing, translation, answering questions, and a variety of other text-based tasks.

LLMs are typically pre-trained by their developers who then provide access to the model to others. Interaction with an LLM consists of giving it *prompts*, which are natural language snippets that instruct the model to produce a desired output. The internal workings of LLMs are opaque for most users, which has led to multiple approaches for constructing functional prompts, often called "prompt engineering" [63]. For a thorough explanation of prompt engineering, see [63].

2.1.1 AI code generation. In addition to text-to-text and text-to-image models such as Dall·E 2 [77], several models specifically aimed at generating programming source code have been released recently. These include Deepmind AlphaCode [61], Amazon CodeWhisperer [5], CodeBert [41], Code4Me [27], FauxPilot [40], and Tabnine [99]. These models are either trained with source code or are LLMs augmented with additional training data in the form of source code. While most of these are aimed at professionals, Copilot presents few barriers to use by novices as it is free for students to use.

These models have proven to be unexpectedly capable in generating functional code. DeepMind purports that AlphaCode can perform similar to the median competitor in programming competitions [61]. Finnie-Ansley et al. found that Codex could solve introductory programming problems better than the average student, performing in the top quartile of real students when given the same introductory programming (CS1) exam questions [42]. A later study conducted by the same group found that Codex was similarly performant in Data Structures and Algorithms (CS2) exams [43]. Chen et al. found increased performance in generating correct source code based on natural language input when the model is prompted to also generate test cases, which are then used for selecting the best generated source code [24].

In addition to their original purpose of generating source code, such models have been found to be capable of other tasks. For example, Pearce et al. explored using several models for repairing code vulnerabilities. While they found that these models were able to repair 100% of synthetic example vulnerabilities, their performance was not as good with real-world examples [81]. Another study by Pearce et al. studied the applicability of AI code generators for reverse engineering [82]. LLMs trained with source code are also good at solving other problems, such as solving university-level math [36], probability and statistics [100], and machine learning [109].

2.2 AI Code Generators and Computing Education

The literature on AI code generators in computing education is growing rapidly but to date there are relatively few empirical evaluations and even fewer user studies. Given their very recent emergence, the impact they will have on educational practice remains unclear at this time [33]. Nonetheless it is clear that there will be many non-trivial impacts, and researchers are currently exploring opportunities and challenges [10]. In work exploring the opportunities and risks presented by these models, Bommasani et al. explicitly list Copilot as a challenge for educators [17], stating that if students begin to rely on it too heavily, it may negatively impact their learning. They also raise concerns about the difficulty of determining whether a program was produced by a student or a tool like Copilot. Similar concerns around over-reliance on such tools were raised by Chen et al. in the article introducing Codex [25]. They included "explore various ways in which the educational ... progression of programmers ... could be influenced by the availability of powerful code generation technologies" in directions for future work [25]. Just how students will adopt and make use of tools like Copilot is unclear [39], but it seems certain they will play an increasing role inside and outside the classroom.

In terms of empirical work in computing education, AI code generators have been evaluated in terms of their performance on introductory programming problems and their ability to generate learning resources. Early work by Finnie-Ansley et al. explored the performance of Codex on typical introductory programming problems (taken from exams at their institution) and on several common (and unseen) variations of the well-known "rainfall" problem [42]. The model ultimately scored around 80% across two tests and ranked 17 out of 71 when its performance was compared with students who were enrolled in the course. In addition, on the "rainfall" tasks, Codex was capable of generating multiple correct solutions that varied in both algorithmic approach and code length. However, the problems in this study were generally fairly simple, and it is likely that more human interaction with the models would be needed for more complex problems [6]. Savelka et al. evaluated GPT-3 on multiple choice questions, quizzes, and more complex programming projects in introductory and intermediate programming courses and found that it is capable of achieving more than passing scores [95].

More recently, Sarsa et al. explored the natural language generation capabilities of Codex by using it to synthesize novel programming exercises and explanations of code suitable for introductory programming courses [94]. They generated programming exercises by providing a single example exercise as input to the model ("one-shot" learning), and attempted to create new problems that targeted similar concepts but involving prescribed themes. They found that well over 80% of the generated exercises included a sample code solution that was executable, but that this code passed the test cases that were also generated by Codex only 30% of the time. In addition, around 80% of the exercises involved a natural language problem description that used the themes that were prescribed, illustrating the ability of the models to easily customize outputs.

Large language models have also been found to be capable of producing pedagogical code explanations [58, 67, 94]. Sarsa et al. used Codex to generate natural language explanations of code samples typically seen in introductory programming classes [94]. Analysis of the thoroughness of the explanations and the kinds of mistakes that were present revealed that in 90% of the cases all parts of the code were explained, but that only 70% of the individual lines had correct explanations. MacNeil et al. incorporated code explanations created by GPT-3 in an online ebook and found that students found these explanations useful for learning [67]. Leinonen et al. compared code explanations created by GPT-3 and students and found that students rated the explanations created by GPT-3 as being both easier to understand and being more accurate summaries of the code [58].

Recent work has analyzed how well large language models could explain programming error messages [60], which are notoriously hard for novice programmers to comprehend [11]. Leinonen et al. found that Codex could be used to enhanced programming error messages in some cases, although they note that the performance of the model was not good enough to use with students directly [60]. Wermelinger [106] evaluated the capabilities of Copilot in the spirit of Finnie-Ainsley et al. [42] who evaluated Codex and found it to still miss the mark more often than not. Reeves et al. evaluated the ability of GPT-3 to solve Parsons Problems and found that the nature of generative AI tools as next-token-predictors means that they often struggle to only use the lines of code provided to them in the prompt to solve the puzzle [90].

2.3 User Studies of AI Code Generation Tools

Recent work has studied how developers use code generation IDE plugins such as Copilot. Vaithilingam et al. had participants complete three programming tasks in Python within VS Code [103]. For one of the tasks, participants used Copilot; in the other cases, they used VS Code's built-in IntelliSense code completion. Although participants did not save time with Copilot, a majority of participants (19/24) preferred Copilot over IntelliSense, citing "time saving" as a benefit. Positive perceptions of Copilot included the generation of starter code and providing programmers with a starting point – even if the starter code led to a "debugging rabbit hole". On the other hand, some participants found code generated by Copilot to be hard to understand and unreliable. In contrast to the present study, which focuses on novice programmers, Vaithilingam et al. had only one participant with fewer than two years of programming experience. They identified three (unnamed) "interaction patterns" that users exhibit when using Copilot. One of these which could be called "confusion" is more of a result of interaction. The other two were "substitution for internet search", and "over-reliance". They also identified two (also unnamed) "coping strategies" used to deal with incorrect code: "accept-and-repair" and "delete and search" where participants rejected Copilot's suggestion and used internet search to proceed.

Barke et al. developed a grounded theory of interaction with Copilot. They asked 20 participants, nine of whom already had experience with Copilot, to complete tasks and found that interactions can be split into "acceleration mode'—in which programmers use Copilot to complete the code they have already planned on writing—and "exploration mode'—in which programmers prompt Copilot to write code they are not sure how to complete [8]. They observed that over-reliance on Copilot can lead to reduced task completion, and having to select from multiple suggestions can lead to cognitive overload.

Jayagopal et al. observed novice programmers using five program synthesis tools [50]. They found that being able to prompt them by starting to write the desired code in the editor "appeared to be more exciting and less confusing" than tools that require a more formal specification. They also observed that novices would engage in prompt engineering if the generated code did not satisfy their specifications, and would sometimes carefully read suggested code.

Bird et al. studied how professional developers who were first-time users of Copilot engaged with it. They found that participants accepted suggestions for efficiency but in doing so gave up a small amount of autonomy and control over their code, noting that some "wrestled" with this tradeoff [16]. A key observation was that developers using AI-assisted tools regularly spent more time reading and reviewing code than writing—an observation that could have important pedagogical implications.

3 METHODOLOGY

In order to better understand how novice programming students interact with AI code generators, we conducted a study observing participants using GitHub Copilot. We then interviewed students about their experience.

3.1 Participants and Context

Participants were all university students age 18–22 enrolled in an introductory programming (CS1) course at a Midwestern private research university in the USA. The language of instruction was C++. We recruited 19 students (5 identifying as women and 14 identifying as men) to participate in the study. All participants were novice programmers and came into the course with little to no prior programming experience. The study took place in April 2022, during the final week of the Spring semester. None of the participants had prior exposure to Copilot and were briefly trained on what to expect and how to use it before the study began.

We observed students solving a new homework assignment in a similar style to all other assignments that semester. Every assignment that semester appeared in the Canvas LMS through a plugin to our automated assessment tool. The assignment (see Figure 2), modeled after the classic game "Minesweeper" was at a level of a programming assignment that could have been assigned two weeks prior. Solving the problem involved receiving input, nested loops, checking two-dimensional storage for certain conditions, updating the received data when those conditions have been met, and outputting the result. As with all programming assignments that semester, students could view the problem description before coding and subsequently submitting their solution. Students were observed during class time in an adjacent room to the regular lecture room. In this way, the context of the study was similar to other invigilated in-class program writing assignments students had received that semester. The only difference was that each student was observed one at a time.

3.2 Procedure

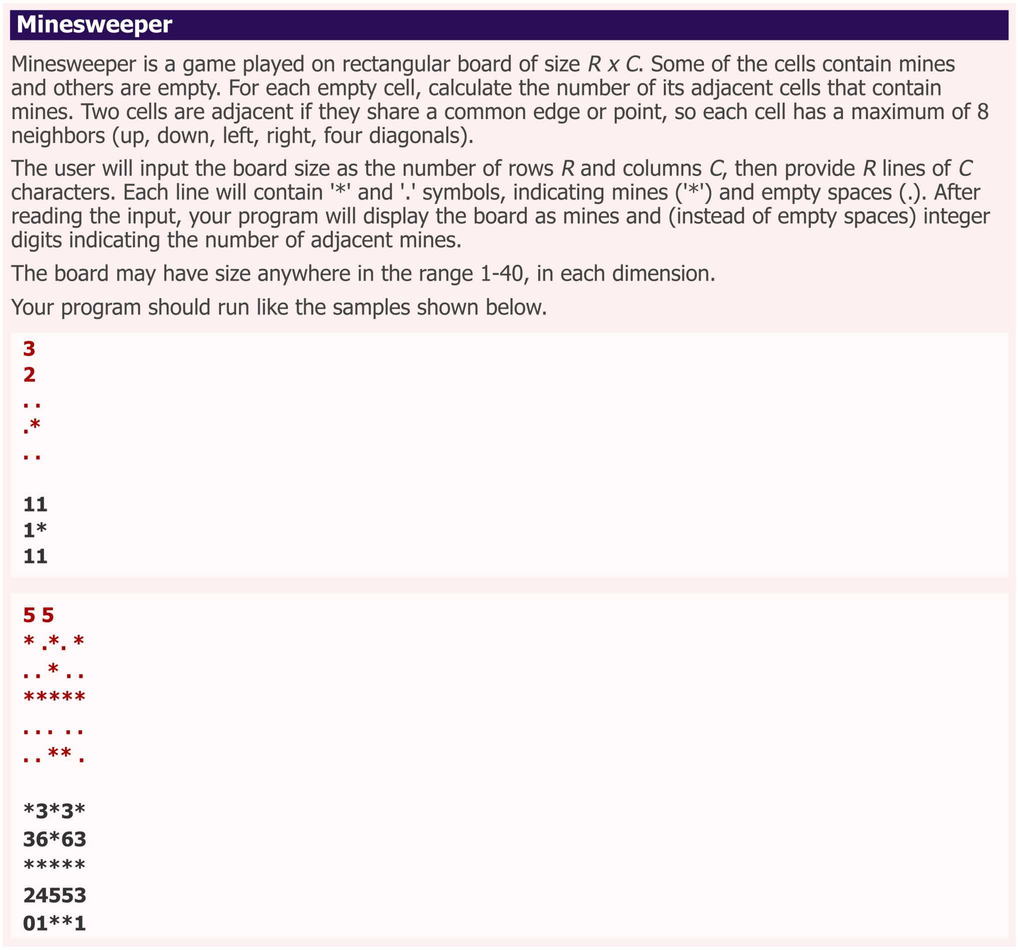
A single researcher sat in the room and observed one student at a time, taking notes about what they did and said, following a think-aloud protocol [38]. Each participant had 30 minutes to complete the program and was allowed to utilize Copilot as well as any resource such as notes or the internet. This is the same time limit as other invigilated in-class code writing activities that semester.

Each participant used **Visual Studio Code** (**VS Code**) with Copilot enabled. Copilot suggests between one and several lines of code in light gray text (Figure 1), which students can either accept by pressing the tab key or reject by pressing escape. Students can also just keep typing when suggestions appear. As students type, Copilot immediately begins suggesting code based on what is present in the text file and its suggestions generally become more accurate, useful and relevant to the task as more code is written.

After completing the program, or after the allotted time had expired, we conducted a short interview which was manually transcribed. Our interview questions were:

(1) Do you think Copilot helped you better understand how to solve this problem? If so, why? If not, why not?
(2) If you had a tool like this yourself, and it was allowed by the instructor, do you think you'd use it for programming assignments? If so, how?
(3) What advantages do you see in a tool like Copilot?
(4) What fears or worries do you have about a tool like Copilot?

Although interview questions are typically much more open-ended, we utilized these targeted questions to focus on our user-centered approach to what student users want from such a system.



Minesweeper

Minesweeper is a game played on rectangular board of size *R x C*. Some of the cells contain mines and others are empty. For each empty cell, calculate the number of its adjacent cells that contain mines. Two cells are adjacent if they share a common edge or point, so each cell has a maximum of 8 neighbors (up, down, left, right, four diagonals).

The user will input the board size as the number of rows *R* and columns *C*, then provide *R* lines of *C* characters. Each line will contain '*' and '.' symbols, indicating mines ('*') and empty spaces (.). After reading the input, your program will display the board as mines and (instead of empty spaces) integer digits indicating the number of adjacent mines.

The board may have size anywhere in the range 1-40, in each dimension.

Your program should run like the samples shown below.

```
3
2
..
.*
..

11
1*
11
```

```
5 5
*.*.*
..*..
*****
.....
..**.

*3*3*
36*63
*****
24553
01**1
```

Fig. 2. The prompt from the automated assessment tool describing the problem that participants were asked to solve.

We tried to frame these questions such that their answers would be of value and interest to instructors.

3.3 Analysis

To analyze the data we used reflexive thematic analysis [18, 20] which aims at understanding pattern and meaning present in qualitative data. This approach requires researchers to engage deeply with the data and develop themes in a process that is flexible, exploratory, and cyclical. Coding is fluid and involves combining and splitting codes in an iterative process as the researchers become more familiar with the data, and throughout the analysis [19]. Since codes develop throughout the process, inter-coder reliability measures are not calculated, but instead reliability of results is achieved through other means. To ensure reliability in our analysis, we held several group meetings where authors compared codes and discussed differences as the themes began to emerge, as is appropriate during reflexive thematic analysis [74]. Our themes are an *output* of the data analysis process, rather than an input as occurs in some other forms of qualitative analysis. We employed six main phases in our analysis, as outlined by Braun and Clark [18]:

(1) Familiarization with the data: Observational notes and responses to interview questions were shared with the research team. Four of the research team focused on the analysis and read through a sample of the observational and interview data to become familiar with it, noting initial ideas and associated codes. The codes that emerged from the data aimed at capturing meaningful expressions and concept.
(2) Generate initial codes: Researchers involved in the analysis met to discuss their initial codes and examples of data that reflected the codes. These codes were combined, split and extended over four meetings as a growing shared understanding of the data was developed.
(3) Searching for themes: These codes were subsequently grouped into potential themes, which were discussed at length. Examples of data that exemplified the potential themes were identified.
(4) Reviewing themes: The themes were reviewed in the context of the entire data set, and refined through discussion and reorganization of codes to better reflect the data.
(5) Defining and naming themes: Names and clear descriptions for the themes were developed to ensure consistency with the data and in the overall context of the study.
(6) Writing: The analysis was completed during the writing process with links formed between the research questions, literature, themes, and data.

We found saturation being the point at which "additional data do not lead to any new emergent themes" [46]. We treated the data from observations and interviews as a single data set since the interviews asked participants to reflect on their experiences in the observation study. The themes that emerged from the analysis of the complete data set therefore form a coherent narrative about the experiences of participants using Copilot.

All methods and data collected from this study were approved by the IRB at the institution where this study was conducted, Abilene Christian University. All participants signed informed consent forms that discussed the study before participating. Data collected from participants was immediately anonymized by creating a key only available to the researcher who collected the data. All other researchers only saw fully deidentified data. All data was stored securely in a Google Drive protected by multifactor authentication.

4 RESULTS

We identified several themes emerging from the data of both the observations and interviews: **Interactions**, **Cognitive**, **Purpose**, and **Speculation**. These themes and the sub-themes that they incorporate can be found in Table 1 along with the number of occurrences in our dataset and the number of unique participants that we observed doing or saying something related to that sub-theme. The sub-theme "No downside" appeared only in interviews, not in the observations. We also include a breakdown at the sub-theme level of answers by interview question in Table 2. For the counts of observation themes by participant, see table Table 3 and for interview themes by participant Table 4. In the remainder of this section, we synthesize results from the observations and interviews, organized by theme, using representative quotes to illustrate them and illuminate participant thinking.

4.1 Theme: Interactions

This theme comprises observations about the interactions and actions taken by participants as they completed the tasks, along with utterances from the observations and reflections from the interviews that reflect those interactions. It was by far the most common theme in the dataset and had four sub-themes. **Coding** is comprised of three kinds of behavior that we observed: coding activities, adapting autogenerated code, and deciphering Copilot's suggestions. We use

Table 1. Themes and Sub-themes Arising from Observations and Interviews, Ordered by Observation Sub-theme Count

	Observations			Interviews		
Theme	Sub-theme	Count	Unique	Sub-theme	Count	Unique
Interactions	Coding	244	19	Coding	11	5
	Accept	93	19	Accept	10	4
	UX	68	17	UX	33	14
	Reject	64	17	Reject	3	3
Cognitive	Confused	65	13	Confused	11	8
	Positive Emotion	30	6	Positive Emotion	8	5
	Metacognitive	24	10	Metacognitive	17	10
	Negative Emotion	6	4	Negative Emotion	14	9
Purpose	Guiding	4	3	Guiding	49	15
	Outsourcing	3	3	Outsourcing	46	17
	Speed	1	1	Speed	37	17
Speculation	Intelligence	6	3	Intelligence	9	4
	Future	2	2	Future	13	8
	No downside	0	0	No downside	10	8

Table 2. Sub-themes Arising from the Interview Study, Listed by Question and Sub-theme Count (For Items with Counts over 4)

Question	Interviews Sub-theme	Count	Unique
Do you think copilot helped you better understand how to solve this problem? If so, why? If not, why not?	Guiding	20	10
	UX	14	7
	Outsourcing	12	8
	Metacognitive	9	8
	Speed	9	8
	Confused	8	6
	Coding	7	4
	Intelligence	6	3
	Positive Emotion	5	3
If you had a tool like this yourself, and it was allowed by the instructor, do you think you'd use it for programming assignments? If so, how?	Speed	13	11
	Outsourcing	10	7
	Guiding	8	5
What advantages do you see in a tool like copilot?	Guiding	20	10
	Speed	15	15
	Outsourcing	7	6
What fears or worries do you have about a tool like copilot?	Outsourcing	17	12
	Future	10	7
	UX	9	4
	No downside	6	6

the term "coding activities" to mean some kind of programming-related task that is not already covered more specifically elsewhere. The sub-theme **User Experience** attempted to bifurcate how Copilot appears to the user (that is, the user interface) and any difficulties using it (that is, the usability).

Table 3. Participant Observation Themes and Sub-themes, Ordered by Total Count

		Participant																			
Theme	Sub-theme	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	Total
Interactions	Coding	27	13	5	12	10	19	23	19	23	18	10	2	6	4	19	18	3	3	10	244
	Accept	6	2	3	10	5	7	3	2	4	6	4	7	1	3	6	7	5	4	8	93
	UX	5	10	1		3	5	2	2	6	10	4	2	1	5	6	3	1	2		68
	Reject	11	8		4	2	9	1	4	5	2	1	4	2	2	2	2	3		2	64
Cognitive	Confused	2		5		1	1			9	12		4	4	6	6	5		4	6	65
	Positive Emotion	2										11		1	2	7			7		30
	Metacognitive	1							2	3	3	6	1	2			4	1		1	24
	Negative Emotion							1					1		2	2					6
Purpose	Guiding									2	1					1					4
	Outsourcing	1													1					1	3
	Speed											1									1
Speculation	Intelligence													3		2			1		6
	Future	1												1							2
	Total	56	33	14	26	21	41	30	29	52	52	37	21	21	25	51	39	13	21	28	

Table 4. Participant Interview Themes and Sub-themes, Ordered by Total Count

		Participant																			
Theme	Sub-theme	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	Total
Purpose	Guiding	4	2	4	1	3	2	1	2	10			1			8	2	4	4	1	49
	Outsourcing	1		3	1	2	4	2	3	2	1	1	2	2	1	4		2	6	9	46
	Speed	3	3		3	2	2	1	2	2	2	2	1	1	2	1	3	4		3	37
Interactions	UX	9	1		1	2	3	1			4			1	3	2	1	1	1	3	33
	Coding	4		1						1		1						4			11
	Accept	5			1											1		3			10
	Reject	1	1								1										3
Cognitive	Metacognitive			1								3	1	2	1	2	2	2	1	2	17
	Negative Emotion	2					1	1			1		1	3	1	2		2			14
	Confused	2				2	1				2		1	1		1				1	11
	Positive Emotion			1		3				1						2		1			8
Speculation	Future	1	1			1		1			3					3		2	1		13
	No downside		2		1	1		1		2					1	1	1				10
	Intelligence					1										3	1			4	9
	Total	32	10	10	8	17	13	8	7	18	14	7	7	10	9	30	10	25	13	23	

Accept here means pressing Tab after Copilot generated a suggestion, which then placed the suggested code into the file. A suggestion is necessarily distracting because it takes up space in the user's view of a file. But when interrupted with a suggestion, if the user just keeps typing, that suggestion disappears. As the typing continues, perhaps another suggestion is made; and so it continues. In the extreme case, users can type out complete files of code and ignore copilot all along the way without having to "interact" with a "UI." The suggestions just keep appearing and disappearing as the user keeps typing. Although it can be very distracting, no extra keystrokes or mouse clicks are required to ignore all these suggestions. Participants accepted both small (one line) and large (multiline) suggestions at an even rate. The ideal accept is one where Copilot offers both useful and necessary code that the user can utilize without modification. For our participants, this most often occurred when Copilot generated code they could easily spot check for correctness, such as a standard input for-loop or a single "cout" statement. Since the presumption is that users are considering each suggestion, we call it **Reject** when participants saw a Copilot suggestion, but continued typing anything but the Tab. Several interesting interaction patterns emerged from data tagged with this theme, which we discuss below.

4.1.1 Interaction Pattern: Shepherding. The first interaction pattern, *shepherding*, is based on three behaviors, the first of which was the phenomenon of the "slow accept." Here participants would type out Copilot's suggestion, often character for character, without outright accepting it (by pressing Tab). We noticed this behavior 14 times in 7 different participants. These experiences were not limited to the first time they encountered Copilot's suggestions. In one case, participant #4 typed out a slow accept of a for-loop, then pressed Tab to accept Copilot's suggestion for what should go inside of it, and then performed another slow accept later for the next loop. Participant #9 performed a slow accept near the end of their session after a slough of regular accepts, rejects, and adaptations. This may indicate that novice programmers are unsure about dropping large amounts of code into their files that they did not write themselves. Participant #1 said:

> P01: "I spent majority of the time decoding the code it gave me...If I saw a prompt I mostly understood, I might use it to help auto fill small parts. I might look through a large chunk of code and see if it's something I could actually use and is the way I want to do it. For someone who is less familiar with the language it could be a hindrance. You might have code that works but you have no idea how it works."

The second interaction behavior in *shepherding* is "backtracking." This occurred when a participant would delete code that they had just accepted without making modifications to it. There were 13 participants who did this at least once and it was the fifth most-frequently occurring behavior we observed (35 times). This indicates that novice programmers may accept auto-generated suggestions without carefully reviewing them first, only to delete them immediately afterward, leading to a distracted workflow. This behavior is similar to one of the two unnamed "coping" strategies identified by Vaithilingam et al. [103] who noted that "In cases where the participant is unable or unwilling to repair the code, they will simply get rid of the entire generated code". Some quotes from participants illuminate this further:

> P06: "If you do not know what you're doing it can confuse you more. It gives you code that you do not know what it does."
> P10: "A downside is having to read suggestions and then delete it."
> P14: "I found it annoying when I hit tab and it wasn't at all what I needed."

The third interaction behavior in *shepherding* is "adapting." Novice programmers often simply accepted code generated for them, but would also adapt it to fit their needs. Sixteen of 19 participants spent at least some of their time adapting code generated for them while several did this for the majority of their time. This caused those in the latter group to write very little code from scratch. This may have contributed to their sense that Copilot was saving them time. These behaviors (slow accept, backtracking, and adapting) contribute to the first novel LLM interaction pattern, *shepherding*, which is the idea that students spend a majority of their time trying to coerce the tool (i.e., Copilot) to generate the code they think they need. This interaction pattern occurred in 7 participants, most frequently moving in the order presented above in 5 participants: slow accept -> backtracking -> adapting. We also observed other combinations of the behaviors, such as participant 12 adapting -> slow accept -> backtracking and participant 19 backtracking -> slow accept -> adapting. It is possible that *shepherding* is a novice analogue to the 'wrestling' that was observed by Bird et al. in their observations of professional developers [16].

4.1.2 Interaction Pattern: Drifting. The second interaction pattern, *drifting*, is based on two behaviors. The first behavior in *drifting* is "deciphering" large blocks of unhelpful code (i.e., auto-generated code that would not lead to a correct solution), observed in 18 participants. These blocks of code often seemed like a nuisance to participants. The constant stream of suggested code was also distracting, since most participants would stop what they were doing and attempt to decipher the suggested code once it appeared. This indicates that novices may have difficulty utilizing Copilot when it is constantly interrupting their problem solving process with dense code suggestions of which they cannot immediately determine the value. Participants struggled with this in various ways, saying:

> P01: "Keeps prompting stuff when you don't need it. It makes it difficult to read what you're typing."
> P06: "Kept prompting things when I didn't need them."
> P10: "Some of the suggestions are too big and confused me on what it was actually suggesting. Wasted time reading instead of thinking."
> P15: "How do you make it only do one line and not the entire thing?"

The second interaction behavior in *drifting* was observed when participants would "accept" and "adapt" these incorrect code suggestions that participants didn't understand. In other words, adapting the malformed auto-generated code only led them further away from a correct solution and instead, down a "debugging rabbit hole" which is a phenomenon also observed in a prior Copilot user study [103]. We observed this behavior in 10 participants. For instance, participant #1 accepted code, attempted to decipher it, adapted it, and then deleted it. They repeated these steps three times before deleting everything and starting over. After accepting the next prompt, they spent time reading it before realizing that the code asks for input into a variable and then never uses that variable. They then undid that before accepting, adapting, and deleting pieces of the next suggestion. At the end of the observation, they had this to say about question #1:

> P13: "It gave good baseline code. It also didn't work well. The code it suggests is not necessarily correct and confused me. The code was relevant to the topic, but not necessarily useful. It declared a variable that was never used, so I spent a majority of the time decoding the code it gave me."

In some cases, participants accepted incorrect code without even trying to adapt it. Participant #13 was observed deciphering multiple times, but never adapting. They accepted many of Copilot's suggestions, even when those code suggestions would not have helped them solve the problem and they didn't seem to be aware of it. Despite that, they had this to say near the end of their coding session:

> P13: "It kind of feels like it's generating what I'm thinking. Doesn't feel right, ya know?"

Finally, after spending much time attempting to understand and adapt the unhelpful code, participants would often delete it and start over with the next suggestion. The sequence of these behaviors leads us to propose a second novel interaction pattern: *drifting*. Copilot is intended to be utilized as a tool to help developers. The interaction behaviors we have observed here are not reflective of how Copilot is utilized by experienced developers. Barke et al. reported that experienced coders will use it to move quickly through their work (acceleration) or discover options (exploration) [8]. Unlike professional developers, novices lack the understanding to know when suggested code will help them solve the problem. *Drifting* then is a cycle consisting of the behaviors of deciphering and accepting/adapting unhelpful code suggestions. The user is drifting from one suggestion to the next, without direction, allowing Copilot to push them along aimlessly. Ten participants engaged in this pattern at least once, but five participants found themselves in a drifting cycle, repeating these same steps multiple times in the same session.

4.2 Theme: Cognitive

Cognitive is the second theme in Table 1 and comprises observations and utterances that reflect participant cognitive state—what they were thinking or feeling—and comprises four sub-themes.

Confused describes when participants did not understand the code Copilot would generate, were confused about how Copilot itself works, and other related elements that were not about Copilot. **Metacognition** is thinking about thinking. In the context of programming, it involves how programmers solve problems and the process they go through to do so [65] and the programming environment likely plays a role in this [51]. Although it is a difficult phenomenon to observe, our participants were seen struggling with the prompt and re-reading it, working out the problem on paper and pencil, and using Copilot to explore possible solutions when stuck. **Positive Sentiment** occurred when participants verbally expressed some kind of positive emotion or exclamation during the observation session, including laughter, wonder, and excitement. Many of these were genuinely surprised and happy at Copilot's capabilities. **Negative Sentiment** similarly occurred when participants were frustrated or annoyed while they were engaged in the programming task. This was often linked to the appearance of large blocks of code suggested by Copilot. However, it also occurred when the suggestions were incorrect or unhelpful.

4.2.1 Finding a Way Forward. Participants were often confused by the output generated by Copilot, an observation also made by Vaithilingam et al. [103] and Barke et al. [8]. We also observed that the auto-generated feedback from Copilot was not always correct, especially early on. For instance, participant #9 saw Copilot generate a suggestion for input that didn't match the problem specification and asked out loud, "Is this correct?" Participant #14 verbally expressed confusion when Copilot generated a comment instead of a line of code, causing them to change their understanding of what Copilot would do and how it could be used. Participant #10 interacted with Copilot for several minutes, accepting multiple suggestions, while still acting and talking like Copilot would also check their program for correctness (much like an automated assessment tool). The cognitive difficulties arising for novices using Copilot can be illustrated with the following quotes:

> P6: "if you do not know what you're doing it can confuse you more. It gives you code that you do not know what it does."
> P13: "It's intrusive. It messes up my thought process. I'm trying to think of a solution, but it keeps filling up the screen with suggestions. And I'd be tempted to follow what it's saying instead of just thinking about it."

However, other participants were able to use Copilot's suggestions when they became stuck as a way of finding a path forward. Using a system like Copilot to discover solutions when stuck is perfectly illustrated in the following quote:

> P15: "It was kind of like rubber ducking without a person."

"Rubber ducking" is a practice some programmers use that involves verbalizing issues to someone or something (classically a rubber ducky). This participant understands the value of rubber ducking and seems to indicate some kind of usual preference for practicing it with people rather than inanimate objects. More interestingly, P15 seems to believe that Copilot can be substituted for this practice, which would mean it can act as a metacognitive scaffold (i.e., facilitating thinking about the problem they are trying to solve and where they are in the problem solving process).

4.2.2 Emotion. The cognitive and metacognitive aspects of using Copilot generated both positive and negative sentiment in participants. For instance, participant #1 laughed when they saw the first multi-line suggestion generated by Copilot. Participant #14 said, "whoa that's super weird" and "that's insane!" at different times during their session. Many of these participants expressed feelings appearing to be related to joy and surprise. In response to code generated by Copilot, participant #18 said, "Oh! That's pretty cool! It, like, read my mind!", "Oh wow. Stop. That's crazy.", and “Where did you find this? Where was this when I was learning programming?” Though only about a third of participants expressed positive emotion, this indicates the kind of emotional reaction possible when a first experience goes as it should and we discuss this further in our design implications. Beyond mere excitement, positive emotion can directly benefit students who may find themselves intimidated or anxious about learning programming:

> P3: “For people like me who don’t know what they’re doing, coming into coding with no prior experience, it’s more encouraging that once you get on the right path you start seeing suggestions that help you. It helps me feel more like I know what I’m doing and feel better about my work and that I can continue a career in computer science.”

On the other hand, negative emotions can reduce student ability to persist and complete tasks required during learning [54]. This can be something as big as failure to move forward in the problem solving process after a lengthy amount of time or as small as receiving feedback in the form of error messages from the compiler. One example of this from our observations was participant #15 who accidentally accepted some code suggested by Copilot and then became visibly agitated by the addition of that code. Some participants appeared to be frustrated by the amount of attention that Copilot demanded. For example, participant #7 reported “It kept auto-filling for things I didn’t want”. Along similar lines, participant #15 was annoyed with the large blocks of code Copilot would suggest and said that “It giving too large of subsections is frustrating.” Although these examples of negative interactions were relatively rare, it remains unknown if such interactions would lessen or be exacerbated with prolonged use of Copilot.

4.3 Theme: Purpose

The third theme in Table 1 is *Purpose*, which captures the reasons that participants give for using Copilot, and potential issues that arise from those motivations, collected from utterances during observation and reflections during interviews. Three separate sub-themes comprise this theme. **Guiding** refers to Copilot assisting participants through the programming problem-solving process, such as helping them learn something new or discover previously unknown edge cases. **Outsourcing** contains two distinct ideas. The first is that participants may be concerned that Copilot could generate working code that they cannot understand and therefore could begin to treat it like a “black box.” The second was the concern that Copilot could become a crutch. Finally, **Speed** refers to any sentiment from participants that Copilot will help them complete their assignments faster than they would otherwise be able to accomplish on their own.

4.3.1 Learning.

Participants commented positively on the guidance that Copilot provided them. Such comments often referred to higher level direction setting and problem solving. For example:

> P05: “if I had a general idea of how to do something it might help me be able to finish it or know what to do next.”
> P06: “might give some useful ideas on how to solve the problem.”
> P15: “I kind of knew what I need to do, just not how to execute it.”
> P19: “It’s guiding me to an answer.”

Three participants specifically mentioned how Copilot’s suggested code taught them something they didn’t know before. While it’s possible to solve the Minesweeper problem with a one-dimensional array, some may find it more intuitive to use a two-dimensional array. However, the course had not yet covered this material. Nevertheless, when Copilot auto-generated code with a two-dimensional array, several students remarked that they had just learned something new. Another example illuminates this further:

P08: “There was a case I hadn’t thought of and it auto-completed it and I was like: Oh, I guess I need to think about that case.”

Several participants also expressed concerns about using Copilot in practice and potential negative effects on learning. Some worried that they wouldn’t learn what was necessary to succeed in class (and, therefore, the field):

P03: “If someone is using it all of the time, then they’re not actually processing what’s going on, just hitting tab, and they don’t know what exactly they’re implementing.”
P06: “I don’t have to know how to code, it would just do it for me.”
P08: “It would make me a worse problem solver because I’m relying on it to help me out.”

Students were aware of the risk of over-reliance on the suggestions produced by Copilot and this idea appeared in one third of the sessions. On introductory level problems like the one in our study, Copilot generates correct solutions most—but not all—of the time [42]. Over-reliance on the tool is thus a particular problem when the suggested code is incorrect, as noted by participant #11: “I could potentially see myself getting a little complacent and at some point not really proofreading the longer bits it suggests and then missing something.” Students also expressed concerns that using Copilot like a crutch would hinder their learning, such as participant #12, who said: “If I was using it, I would become dependent on it. I might zone out in class and not pay attention because I would think ‘Oh I could just do this with Copilot.’ So it would be my crutch.”

4.3.2 Going Faster. Students perceived efficiency gains from not having to type the code themselves as well as from suggested approaches for solving the problem. Similar interactions were noted by Barke et al. [8] who called this class of behavior ‘acceleration’. Participants stated this as:

P01: “If I saw a prompt I mostly understood I might use it to help auto fill small parts.”
P02: “Yes, it made it faster to think of ideas on how to proceed.”
P06: “If I can do the program without it I wouldn’t, if I knew how to solve it I would use it to be faster.”
P11: “It would have taken me forever to type out the loop, but it put it in and all I had to do was proofread.”

One participant noted the efficiency gained by a shift away from time spent typing code and toward time spent thinking about the problem: “So much faster! It got me to testing in less than 20 minutes and most of that time was me reading the problem and thinking about it.” Copilot tends to generate syntactically-correct suggestions, and thus may be particularly helpful in assisting students to avoid syntax errors. A couple of participants illustrated this well during the interview:

P11: “And then the syntax is a huge thing. It just gave me the syntax and all I had to do was proofread.”
P17: “If nothing else, it cuts down on time and going back-and-forth checking how to do things. I liked having Copilot for syntax because that’s been my biggest challenge so far.”

Despite mostly positive statements regarding speed improvements, not all participants viewed it as a benefit to avoid typing code to save time. However, some of the participants in our study recognized the value in typing out code as a kind of practice for learning, noting that accepting Copilot suggestions can interfere with this:

P18: “I think typing out your own code helps you memorize little things and details. When you have it handed to you, you forget little things like semicolons or where parentheses are supposed to be.”

4.4 Theme: Speculation

The last theme in Table 1 is *Speculation* and includes any statements about the potential future use of Copilot and the concerns or issues that might arise. There are three sub-themes that comprise this theme. **Intelligence** here refers to when a participant indicated they thought there was some level of intelligence in Copilot, such as it "knowing" things. **Future** refers to participant speculation about the world as it will be once tools like Copilot are commonplace, such as putting programmers out of their jobs. Finally, we tagged participant reflections in the interviews with **No downside** if they did not say anything negative about the implications of using Copilot.

4.4.1 Agency of Artificial Intelligence. Copilot was a new experience for the participants, and their exposure to the tool was limited to the 30 minute programming activity used in the observation study. During the coding activity, several participants reported that they felt that Copilot was aware, knowledgeable, and had agency. The following quotes illustrate this feeling:

> P13: "Does Copilot know what I'm trying to do? It kind of feels like it's generating what I'm thinking."
> P19: "It's guiding me to an answer."
> P18: "It like read my mind!"
> P15: "I thought it was weird that it knows what I want."

4.4.2 Fears and Concerns. Given the attribution of intelligence to the system, it is unsurprising that some of the responses from students were speculative and expressed concerns about how it might be used in the future. For example, participant #15 expressed concern that "It might take over the world and write its own code." Two students also expressed concern that Copilot may impact the job market, perhaps taking jobs from software developers. Students also expressed concerns that Copilot may raise ethical issues involving privacy, and were uncertain about the implications for plagiarism. We discuss these further in Section 5.

5 DISCUSSION

We now return to our research questions to discuss the implications of our findings on novice programmers using LLM-based AI code generators such as Copilot for the first time. We then discuss ethical considerations arising from the use such systems. Finally, we offer design implications based on all of the findings and insights we have presented.

5.1 User Interactions

Our first research question was, "How do novices interact with GitHub Copilot when they first encounter it?"

As the code suggestions by an AI code generator could be seen as feedback on the student's current program, we discuss the results of the first research question with the theoretical lens of feedback. We consider the suggestions of Copilot through Hattie and Timperley's model of feedback that focuses on three feedback questions "Where am I going?", "How am I going?" and "Where to next?" [47].

Copilot mainly gives students feedback on "Where to next?" We found that novices happily utilized Copilot and both accepted and rejected Copilot's code suggestions. Rejecting some suggestions implies that at least some of the novices thought that the feedback by Copilot could be wrong. Novices used Copilot both for initial guidance on the right direction to take and for creating code when they knew what they wanted. These align with prior work by Barke et al. who categorized experienced programmers' Copilot use into "exploration" and "acceleration" [8]. From the point of view of feedback, exploration could be seen as the students trying to get feedback from

Copilot on their initial ideas of how to solve the problem ("Where am I going?"), while acceleration would align more with getting feedback on the current implementation strategy ("How am I going?").

In addition to exploration and acceleration, we observed two novel types of behavior that we call "shepherding" and "drifting". In shepherding, students spent the majority of the time trying to coerce Copilot to generate code, which, for this set of novice users, we view as a potential signal of tool over-reliance. This is similar to earlier results that have found that students sometimes develop an over-reliance on automatically generated feedback from automated assessment systems [7]. In the case of these systems, prior work has presented multiple ways of trying to combat over-reliance such as submission penalties [7, 59] or limiting the number of submissions students can have [35, 49, 59]. It is a good question whether similar limits should be imposed on novices using AI code generators such as Copilot to try to curb over-reliance. From the feedback point of view, focusing solely on "Where to next?", which Copilot is most apt in, might lead novice students down incorrect solution paths – this is similar to prior work where more experienced programmers were led down "debugging rabbit holes" by Copilot [103].

In the other novel behavior we observed, "drifting," students hesitantly accepted Copilot's suggestions, possibly played around with them, but then ended up backtracking and deleting the code, only to repeat the cycle from the beginning. From the point of view of feedback, here, students might be suspicious of the feedback of an AI-system—prior work has found that human trust in AI advice depends on the task and to what extent the human believes the AI to be capable of giving advice on the task [104]. In addition, most existing automated feedback systems in programming focus more on identifying errors ("How am I going?" and "Where am I going?") and less on providing actionable feedback [53], and thus students might not be accustomed to receiving automated feedback on "Where to next?".

Even though participants would occasionally reject Copilot's suggestions, most participants seemed to have a high confidence in Copilot, believing it would generate useful and correct code most of the time. This might be especially true for novices, who could be familiar with automated assessment systems where the feedback is based, e.g., on instructor-created unit tests [53] where the assumption of the feedback being correct is typically valid. Novices tend to view feedback in programming contexts to be the truth and the systems generating it to be infallible [56]. When using LLM-based AI code generators such as Copilot, however, this belief is troublesome – recent studies in the context of learning programming have found the correctness of Codex (the LLM that powers Copilot) to be around 70%–80% [42, 94], meaning that in about 20% –30% of cases, the suggestion by Copilot would be incorrect. This is especially troublesome for novice programmers who might have a hard time identifying whether the suggestion is correct or not.

One might assume that since participants were prompting the system for C++ code that it might generate highly advanced features from newer releases, e.g., C++20. However, language models like Codex can best be seen as continuing the prompt they were given. This has the effect that Codex will generate code of a similar level of sophistication to the code which it was prompted with, including the possibility of introducing more bugs if prompted with buggy code [25].

While some of these sorts of issues will likely be minimized over time, or perhaps even through just one session with Copilot as students gain familiarity with the tool, it is still important to think through the very first interaction with the system. We consider this point further when presenting design implications (Section 5.5). Moreover, these interaction patterns fit well with those of Vaithilingam et al. who studied experienced programming students [103]. They reported that students often spent too much time reading the large code blocks suggest by Copilot and the constant context switching between thinking, reading, and debugging led to worse task performance. Overall, this seems to be worse for the novice programmers in our study who spent more time deciphering code and were more easily confused and led astray. This is because, as Chen et al. write, the model is not aligned with the user's intentions [25]. Here, one possibility is that the user (the student) is using Copilot as a feedback mechanism where the user expects feedback to be actionable and of good quality, while Copilot's original purpose is to be the user's "AI pair programmer", where it might be expected that some of the suggestions are not worthwhile to explore.

The primary benefit that novices saw in using Copilot was that it accelerated their progress on the programming task, mirroring results with experienced programmers as reported by Barke et al. [8] and Vaithilingam et al. [103]. A related benefit of using Copilot suggestions over typing code directly is the avoidance of syntax errors. The computing education literature documents syntax errors—and the messages they generate—as presenting a significant challenge to both novices and students transitioning to new programming languages [11, 31]. Although Copilot's suggestions are capable of inserting syntactically-invalid code, the vast majority of its suggestions are syntactically-valid, and use identifiers appropriate for the existing code context. Essentially, Copilot might enable the student to work on a higher level of abstraction where they can spend their mental effort on thinking about the semantics of the program instead of the syntax.

Conversely, the "slow accept" phenomenon in which participants simply typed out the code suggestions character by character, could also be beneficial to student learning. This form of typing practice has pedagogical benefits for learners and researchers have explored similar typing exercises, in which learners must type out code examples, as a way to help novice students practice low-level implementation skills [44, 57]. While Copilot provides good opportunities for this type of practice, it is tempting to simply accept Copilot's suggestions.

5.2 User Perceptions

Our second research question was, "How do novices perceive their first-time experience of using GitHub Copilot?"

For introductory programming, Copilot could help novice programmers in creating code faster and help avoid the programming version of a "writer's block." Copilot could also work as a metacognitive scaffold, guiding students to think about the problem they are solving at a higher level, such as planning the structure of the code and then creating individual components with Copilot. Thus, we discuss the results related to the second research question mainly with the theoretical lens of metacognition and self-regulation.

Programming is a complex cognitive activity [79] that often involves deep metacognitive knowledge and behaviors [66]. One example of a programming activity that includes both the cognitive and metacognitive aspects is code tracing with concrete inputs, something constrained to working memory [28]. Several participants in our experiment took out a notepad and pencil when they became stuck in an attempt to work the problem, which is a clear example of a reflective metacognitive behavior like self-regulation [64]. Others faced the kind of metacognitive difficulties described by Prather et al. like feeling a false sense of accomplishment at having a lot of code, but still being far from a working solution [88]. Several participants misunderstood the problem prompt and had to return to it multiple times, a pattern also seen by Prather et al. [87]. Still, others utilized the system like a colleague who can help them when stuck, a behavior previously documented between students co-regulating their learning in study groups [86], and which matches the idea of Copilot being "your AI pair programmer." While metacognitive skills in novice programmers are becoming increasingly important [66, 85], there are clear opportunities for tools like Copilot to scaffold and enhance these behaviors from the very start. We discuss this in design implications below (Section 5.5).

In the article introducing the Codex model, Chen et al. outline a number of potential risks that code generation models present [25]. The first of these risks is over-reliance by developers on the generated outputs, which they suggest may particularly affect novice programmers who are learning to code. Indeed, this was the most common concern echoed by the participants in our study when asked to describe their fears and worries around this new technology. Our participants acknowledged that such over-reliance could hinder their own learning, a concern that has also been expressed by computing educators [39]. From the point of view of self-regulation, students will need better self-regulation skills to self-control their use of tools like Copilot to not develop an over-reliance on them—at least when they are freely available for use at the student's discretion. In fact, we hypothesize that over-reliance on tools like Copilot could possibly *worsen* a novice's metacognitive programming skills and behaviors.

Naturally, as we enact these cognitive and metacognitive behaviors, emotional arousal can be triggered in response. Emotion is a key part of creating usable designs [2, 21, 69]. When novice programming students experience negative emotions, it can directly negatively impact their feelings of self-efficacy [55]. Self-efficacy, which is a related metacognitive belief, is one of the most direct measurements that correlates to a student's success or failure in computer science [91]. The potential for the design of a tool like Copilot to arouse positive emotion, encourage, and therefore increase self-efficacy, especially in traditionally underrepresented minorities in computing, should not be understated. Women and underrepresented minorities consist of just 18% and 11% of bachelors degrees in computing, respectively, and are often part of the so-called "leaky pipeline" [23]. Negative experiences tend to impact underrepresented groups more than majority groups, leading to dropping from the major [70]. Copilot's current interaction style may actually promote cognitive and metacognitive difficulties as well as negative emotion in novice users, which would have the opposite effect on their self-efficacy. We believe our design recommendations (Section 5.5) can help mitigate these concerns.

5.3 Differentiation From Expert Usage

Our findings reveal that Copilot provides a variety of assistance to novices working on typical introductory-level programming tasks, from the avoidance of trivial syntax errors to guidance through the problem-solving process. Dealing with syntax errors has historically been a notoriously difficult task for novices learning to program [11, 32]. Thus, on its own, the use of Copilot to help generate syntactically correct code already promises to be a significant learning aid. Moreover, problem-solving is a core skill that novices must develop and is sometimes taught explicitly with the help of targeted tools [65, 83]. We observed students using Copilot directly to aid their problem-solving, and commenting positively on this support. Thus, there is considerable utility in the support that Copilot provides for learners transitioning into the world of programming. We observed a number of interesting interaction patterns, broadly categorized under the themes of "shepherding" and "drifting", tailored to the requirements of novice learners. Thus we can contrast the human-Copilot interactions between novices and experts in several ways.

Firstly, Copilot is designed to be used by experts with the primary goal of boosting productivity or resolving specific issues [84]. Conversely, novices are engaged in the process of grasping foundational programming concepts, and thus leverage Copilot to bridge their knowledge gaps and enhance their understanding of complex ideas that are just beyond their current level of knowledge. For example, participants commented on the guidance that it provided, such as helping them "know what to do next". Secondly, novices acquire essential programming knowledge in a progressive manner while interacting with Copilot. From very basic beginner-level tasks to more advanced challenges, Copilot can help to sustain this growth by offering customized assistance. This differs from typical interactions between Copilot and professional programmers who have already developed a broad level of expertise in multiple areas.

Exploration and experimentation are also key difference between novice and expert interactions with Copilot. As learners work to develop their programming skills, an important aspect of learning is experimentation with different approaches and making and recovering from mistakes. Copilot enables this type of exploration as learners can try alternative approaches in a safe environment where errors can be easily identified and corrected. For example, we observed participants using Copilot to get initial feedback on their ideas. This type of exploration can enrich a novice's understanding of programming principles, which is different to the experience of expert programmers who are typically looking for an immediate solution, for example, to optimize or refine existing code and are less likely to explore and compare alternative approaches.

Finally, we observed some interaction behaviors that are unique to novices, notably the "slow accept" where a learner would type character for character over the top of a suggestion. This interaction behavior suggests a learner who is building confidence with syntax, and is not likely to be common for expert programmers who are seeking improved productivity. In general, the contrast between pedagogical usage and expert application showcases Copilot's versatility, but also suggests there is a need for design considerations to cater for different audiences. We address these design considerations explicitly in Section 5.5.

5.4 Ethical Considerations

A number of complex ethical issues have emerged from the recent development of powerful models for AI code generation. These include issues relating to the data on which the models are trained, raising legal and societal concerns, and immediately pressing issues relating to academic misconduct. We found it interesting that even with the short exposure to Copilot in our study, participants raised concerns about a range of ethical issues such as privacy and employability, suggesting that Copilot may initially be perceived as threatening by some students. We suggest that it is important for educators to be aware of these concerns, and to help students appreciate the implications of tools like Copilot. The issues we raise are also relevant for designers, as we discuss in Section 5.5.

5.4.1 Academic Misconduct. Academic misconduct is a widespread problem in many disciplinary areas [72, 96]. The availability of AI code generators makes this a particularly complex problem for computing educators because they increase the opportunity for misconduct to occur while at the same time decrease the likelihood that it is detected. Code generators like Codex have been shown to perform better than the average student on introductory-level programming problems, thus they provide an effective tool for students who might be contemplating cheating [42]. Compared to traditional forms of cheating, such as contract cheating or copying work from other students, AI-generated solutions do not require communication with another person and thus there are fewer risks of being caught [107]. Moreover, AI-generated code is diverse in structure and resilient to standard plagiarism detection tools. Biderman and Raff show that introductory level programming assignments can be completed by AI code generators without triggering suspicion from MOSS, a state-of-the-art code plagiarism detection tool [15].

A recent systematic review of the literature on plagiarism in programming assignments reported the common ways that students rationalize acts of plagiarism [3]. These included a belief by students that it was acceptable to use code produced by others if it required some effort to adapt. This raises questions about whether code generated by Copilot and then modified by a student can count as their own for academic submission purposes. Most development environments provide some standard code completion tools for basic syntax elements. Copilot extends this autocomplete interaction to suggesting large blocks of code, some of which we observe students choose to type out character by character. In such a case, can a student claim to have created the program themselves? Reminiscent of the classic *Ship of Theseus* thought experiment, it remains an open question as to how much is too much when it comes to code generated by a tool versus written from scratch by a student if we are to claim that the student wrote the code submission. We expect significant implications ahead for issues of academic integrity, and a clear need for an updated definition of plagiarism [30].

5.4.2 Code Reuse and Intellectual Property. As educators, one of our roles is to teach students about their professional responsibilities when reusing code outside of the classroom. Code that is publicly available, such as the code repositories used to train the Codex model, may be subject to various licenses. In particular, code from open-source software packages is often released under a GPL license which states that any derivative works must be distributed under the same or equivalent license terms. However, when code is generated by AI models, it is not always clear how the source should be attributed. A recent legal controversy has arisen due to the fact that Copilot can sometimes generate large sections of code verbatim from open source repositories, but not clearly signal the source. This means that developers may end up using code but violating the license terms without being aware of it. A class-action lawsuit was filed in November 2022 claiming that Copilot violates the rights of the creators who shared the code under open source licenses [102]. When teaching students to use AI code generators, educators should also teach students about how the models are trained so that they appreciate the legal and ethical implications. In a similar vein, the use of Copilot and similar tools typically requires that the programmer gives their data (e.g., source code and interaction with the model) to the developer of the tool (for example, GitHub when using Copilot). In power relationships such as the student-instructor relationship, students might feel pressured to yield their data.

5.4.3 Harmful Biases. Biases present in training data sets are known to lead to bias in subsequent AI models [14, 92]. This was demonstrated rather spectacularly with the recent launch of Meta AI's Galactica language model, trained on millions of academic papers, textbooks and lecture notes [101]. Three days after it was launched, following a great deal of ethical criticism, it was taken offline due to evidence that it would generate realistic but false information and content that contained potentially harmful stereotypes. Code generation models are not immune to these issues, which can suffer from misalignment between the training data and the needs of the end users, to perpetuating more serious and harmful societal biases. The developers of Codex note that it can be prompted in ways that "generate racist, denigratory, and otherwise harmful outputs as code comments" [25], and that it can generate code with structure that reflects stereotypes about gender, race, emotion, class, the structure of names, and other characteristics [25].

With respect to the data itself, code generation models are mostly trained on public code repositories and this raises the question of whether the contents of these repositories are appropriate for novices who are learning to program. For example, the style of code published in public repositories may not match educational materials well, or may use advanced features which could be confusing to novices. In addition, it has been shown that AI generated code can sometimes contain security vulnerabilities, which may mislead learners into adopting bad coding habits [80].

As students begin to more widely adopt Copilot and similar code generating tools, it will become increasingly important for educators to teach students about their broader social and ethical implications.

5.5 Design Implications

In this section we reflect on the design implications that arise from the themes identified in the observation and participant interview data. These interface design considerations are targeted at better supporting novice programmers or first-time users. We imagine they will be less relevant to experienced and expert programmers. Therefore, users should be able to select what kind of feedback they wish to receive from Copilot and adjust it or even hide it as they learn and grow in programming skill.

5.5.1 Prompt Control. When students expressed frustration with Copilot, it was often due to usability issues. In particular, students did not like being shown suggestions when they didn't need the help as this slowed them down. This suggests the need for a new interaction experience for novices. Currently, Copilot generates suggestions in real-time and displays them without prompting by the user. In addition, Copilot provides the same interaction experience for all users. There is scope to make use of a wealth of contextual information—such as the type of problems being solved and knowledge about the background of the user—to adjust how and when the code suggestions are made. Novices who are learning to program may benefit from being able to attempt problems initially on their own and request help when needed, rather than having to ignore suggestions when they are not wanted.

Related to the *drifting* behavioral pattern that we observed, when Copilot suggested large amounts of code, students typically spent a substantial amount of time and effort deciphering the suggestion. Frequently, longer suggestions that were accepted were subsequently modified, or deleted entirely. The utility of shorter suggestions was able to be determined more rapidly than longer suggestions by participants, and short suggestions that were accepted were less likely to be changed or deleted. This suggests that Copilot's suggestions could be more useful to students (especially novice students) with a selection algorithm that preferences shorter solutions, or filters longer solutions. We hypothesize that lengthy auto-generated code suggestions may lead to an increased cognitive load [98] among novices and call on researchers to explore this in future work.

5.5.2 Metacognitive Scaffolding. As discussed above, Copilot's user interface could enhance or harm novice programmer metacognition. Previous work shows that a system providing enhanced feedback specifically targeted at novice programmers can increase efficacy [34, 89] and metacognitive behaviors [87]. Some participants in our study used Copilot to move past metacognitive difficulties by "rubber ducking," but this seems to be an uncommon behavior that could be better supported through the interface itself. As shown in Figures **??** and **??**, Copilot manifests its suggestions as a single line or entire blocks of gray text. Scaffolding the user's movement through the problem solving process could involve providing an unobtrusive UI element directly above the suggested code to allow users to cycle through different code suggestions. Copilot is already capable of obtaining multiple suggestions per input prompt, however, it can benefit from explicit, discoverable UI that states how many suggestions are available at a given point. This could encourage students to engage more in both the *exploration* and *shepherding* interaction patterns. Another form of scaffolding could be Copilot generating only comments outlining the program's general structure, similar to subgoal labels [76].

5.5.3 Better Mental Models via Explainable AI. During the observation sessions, participants were often confused by Copilot's code generation capabilities and the code that it generated. Similarly, during the interviews participants in our study worried that Copilot would generate working code they could not understand. There is a need for systems like Copilot to help the user understand what it's doing and this could be especially effective for novice programmers. **Explainable AI** (**XAI**) is the idea that AI systems and the decisions those systems make should be understandable by people. Although XAI has been studied by researchers for nearly 40 years [26], it is increasing in importance as modern machine learning becomes more frequent in our daily lives [62]. While most work in this area focuses on the algorithms, HCI researchers have called for more studies on how it impacts the humans using or benefiting from these AI systems [1]. According to

Wang et al., the way humans reason informs XAI techniques [105]. They suggest design techniques for XAI that include supporting hypothesis generation and supporting forward (data-driven) reasoning. These are ideal for LLM-powered systems like Copilot because users are engaged in a cyclical pattern of writing code, reading Copilot's auto-generated suggestions, and either accepting or rejecting those suggestions. During this cycle, users are building a mental model of how Copilot works and this informs how they will attempt to utilize Copilot next. Since precise prompt creation to LLMs may become an increasingly important tool (much like "Googling" is today), we argue it is important to utilize user-centered XAI design techniques when exposing the model to users. Therefore, we recommend that systems like Copilot should help users see a little bit into the black box, such as what it is using as input, a confidence value (or visualization), and its own estimation of the skill level of the user. For example very recent products, notably OpenAI's ChatGPT [78], have begun to present user interfaces that support conversational dialogue and thus are ideally suited to explaining underlying decisions to users.

5.5.4 Ethical Design. The current legal controversy regarding code reuse and licensing (see Section 5.4.2) arises from the fact that code generator models are trained on repositories of code that may be covered by licenses that dictate their use. They are prone to generating code that may be identical to portions of code from this training data. This can be a problem in cases where well meaning users are shown such code suggestions but without the corresponding license or link to the source repository. Indeed, there exist numerous reports of users engineering prompts to Copilot that guide it toward producing large sections of code from licensed repositories without appropriate attribution. Unintentionally, developers may create projects that contain fragments of code with incompatible licenses.

AI code generators can be designed to address this problem by better signaling to users when generated code matches an existing source, or hiding suggestions that may not meet a user-defined criteria around license use. For example, early versions of GitHub Copilot included a filter that was able to detect when code suggestions matched public code from GitHub. Users could choose to enable this filter, in which case matches or near matches would not be shown as suggestions. Planned versions of Copilot, scheduled for release in 2023, will include references to source code repositories when they contain code that matches suggestions [93]. Although certain fragments of code are likely to appear across multiple repositories, code generator tools can link to authoritative sources by filtering based on the date on which code was committed. Such design features may help educators in their efforts to teach students about ethical code reuse, and assist students in better citing sources for their work in the case that generated code is not novel.

5.6 Limitations

There are multiple limitations to this work. First, we did not record screens or audio due to IRB considerations. However, we believe that the observations of what students did, combined with the transcribed interviews, is sufficient to understand their interactions with the user interface at an informative level. Second, some of the conditions of the study were more like a lab-based experience and not like how students normally solve their in-class programming assignments. We attempted to mitigate this as much as possible (see Section 3.2), but this may have affected the way participants worked and interacted with the tool. Third, although there are multiple code-generating tools now available, we only looked at Copilot. This is because it was the only easily available such tool at the time of data collection. Finally, this study took place at the end of April 2022. The release of Copilot on March 29, 2022, did not leave much time to conduct a complex, multi-channel data collection. We moved quickly to uncover early findings on the potential implications of this technology in introductory programming courses. During this study, all of these experiences were novel to our students. By the time of writing, even if we had not conducted this study, it is likely that many of our students would have been exposed to Copilot through other means. Novelty effects will wear off over time. Our results reflect actions and thoughts that occur when students are first exposed to Copilot.

5.7 Future Work

There are many interesting avenues for future work. For example, studying longer term student use of Copilot, e.g., over a full semester to understand if and how interaction with Copilot evolves over time and as students become more skilled in programming. Additionally, future work should seek to understand the reasons behind some of the observations we made. For example, whether having the code suggestions visible all the time leads to increased cognitive load, which could explain why students were frustrated—and why prior studies with more experienced programmers have not reported similar findings. Altogether, we see it as very important to examine how AI code generators can most effectively be incorporated into introductory programming classrooms.

6 CONCLUSION

In this work we provide the first exploration of Copilot use by novices in an introductory programming (CS1) class on a typical novice programming task through observations and interviews. We found that novices struggle to understand and use Copilot, are wary about the implications of such tools, but are optimistic about integrating the tool more fully into their future development work. We also observed two novel interaction patterns, explored the ethical implications of our results, and presented design implications. These insights are important for integrating AI code generators into the introductory programming sequence and for designing more usable tools for novice users to solve issues related to helping them get "unstuck" in programming tasks at scale.

REFERENCES

[1] Ashraf Abdul, Jo Vermeulen, Danding Wang, Brian Y. Lim, and Mohan Kankanhalli. 2018. Trends and trajectories for explainable, accountable and intelligible systems: An HCI research agenda. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, New York, NY, 1–18. DOI: https://doi.org/10.1145/3173574.3174156

[2] Anshu Agarwal and Andrew Meyer. 2009. Beyond usability: Evaluating emotional response as an integral part of the user experience. In *Proceedings of the CHI'09 Extended Abstracts on Human Factors in Computing Systems*. Association for Computing Machinery, New York, NY, 2919–2930. DOI: https://doi.org/10.1145/1520340.1520420

[3] Ibrahim Albluwi. 2019. Plagiarism in programming assessments: A systematic review. *ACM Transactions on Computing Education* 20, 1 (2019), 28 pages. DOI: https://doi.org/10.1145/3371156

[4] Joe Michael Allen and Frank Vahid. 2021. Concise graphical representations of student effort on weekly many small programs. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, New York, NY, 349–354. DOI: https://doi.org/10.1145/3408877.3432551

[5] Desai Ankur and Deo Atul. 2022. Introducing Amazon CodeWhisperer, the ML-powered Coding Companion. Retrieved from https://aws.amazon.com/blogs/machine-learning/introducing-amazon-codewhisperer-the-ml-powered-coding-companion/

[6] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732. Retrieved from https://arxiv.org/abs/2108.07732

[7] Elisa Baniassad, Lucas Zamprogno, Braxton Hall, and Reid Holmes. 2021. STOP the (autograder) insanity: Regression penalties to deter autograder overreliance. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, New York, NY, 1062–1068. DOI: https://doi.org/10.1145/3408877.3432430

[8] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 27 pages. https://doi.org/10.1145/3586030

[9] Kent Beck. 2000. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional, Boston, Massachusetts.

[10] Brett A. Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming is hard - or at least it used to be: Educational opportunities and challenges of AI code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1.* Association for Computing Machinery, New York, NY, 500–506. DOI: https://doi.org/10.1145/3545945.3569759

[11] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler error messages considered unhelpful: The landscape of text-based programming error message research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education.* Association for Computing Machinery, New York, NY, 177–210. DOI: https://doi.org/10.1145/3344429.3372508

[12] Brett A. Becker, Cormac Murray, Tianyi Tao, Changheng Song, Robert McCartney, and Kate Sanders. 2018. Fix the first, ignore the rest: Dealing with multiple compiler error messages. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education.* Association for Computing Machinery, New York, NY, 634–639. DOI: https://doi.org/10.1145/3159450.3159453

[13] Brett A. Becker and Keith Quille. 2019. 50 years of CS1 at SIGCSE: A review of the evolution of introductory programming education research. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education.* Association for Computing Machinery, New York, NY, 338–344. DOI: https://doi.org/10.1145/3287324.3287432

[14] Emily M. Bender. 2019. A Typology of Ethical Risks in Language Technology with an eye towards where transparent documentation can help. *Future of Artificial Intelligence: Language, Ethics, Technology Workshop.*

[15] Stella Biderman and Edward Raff. 2022. Fooling MOSS detection with pretrained language models. In *Proceedings of the 31st ACM International Conference on Information and Knowledge Management.* Association for Computing Machinery, New York, NY, 2933–2943. DOI: https://doi.org/10.1145/3511808.3557079

[16] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. 2023. Taking flight with copilot. *Communications of the ACM* 66, 6 (2023), 56–62. DOI: https://doi.org/10.1145/3589996

[17] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. 2022. On the Opportunities and Risks of Foundation Models. arXiv:2108.07258. Retrieved from https://arxiv.org/abs/2108.07258

[18] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (2006), 77–101. DOI: https://doi.org/10.1191/1478088706qp063oa

[19] Virginia Braun and Victoria Clarke. 2019. Reflecting on reflexive thematic analysis. *Qualitative Research in Sport, Exercise and Health* 11, 4 (2019), 589–597. DOI: https://doi.org/10.1080/2159676X.2019.1628806

[20] Virginia Braun and Victoria Clarke. 2022. Conceptual and design thinking for thematic analysis. *Qualitative Psychology* 9, 1 (2022), 3–26. DOI: https://doi.org/10.1037/qup0000196

[21] Scott Brave and Clifford Nass. 2007. Emotion in human-computer interaction. In *Proceedings of the Human-computer Interaction Handbook.* CRC Press, Boca Raton, FL, 103–118.

[22] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901.

[23] Tracy Camp, W. Richards Adrion, Betsy Bizot, Susan Davidson, Mary Hall, Susanne Hambrusch, Ellen Walker, and Stuart Zweben. 2017. Generation CS: The growth of computer science. *ACM Inroads* 8, 2 (2017), 44–50. DOI: https://doi.org/10.1145/3084362

[24] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. CodeT: Code Generation with Generated Tests. arXiv:2207.10397. Retrieved from https://arxiv.org/abs/2207.10397

[25] 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. Retrieved from https://arxiv.org/abs/2107.03374

[26] William J. Clancey. 1983. The epistemology of a rule-based expert system—A framework for explanation. *Artificial Intelligence* 20, 3 (1983), 215–251.

[27] Code4Me. 2022. Code4Me. Retrieved May 24, 2023 from https://code4me.me/

[28] Will Crichton, Maneesh Agrawala, and Pat Hanrahan. 2021. The role of working memory in program tracing. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems.* Association for Computing Machinery, New York, NY, 13 pages. DOI: https://doi.org/10.1145/3411764.3445257

[29] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. 2018. Intelligent tutoring systems for programming education: A systematic review. In *Proceedings of the 20th Australasian Computing Education Conference.* Association for Computing Machinery, New York, NY, 53–62. DOI: https://doi.org/10.1145/3160489.3160492

[30] Nassim Dehouche. 2021. Plagiarism in the age of massive generative pre-trained transformers (GPT-3). *Ethics in Science and Environmental Politics* 21 (2021), 17–23.

[31] Paul Denny, Brett A. Becker, Nigel Bosch, James Prather, Brent Reeves, and Jacqueline Whalley. 2022. Novice reflections during the transition to a new programming language. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1.* Association for Computing Machinery, New York, NY, 948–954. DOI: https://doi.org/10.1145/3478431.3499314

[32] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2012. All syntax errors are not equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education.* Association for Computing Machinery, New York, NY, 75–80. DOI: https://doi.org/10.1145/2325296.2325318

[33] Paul Denny, James Prather, Brett A. Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N. Reeves, Eddie Antonio Santos, and Sami Sarsa. 2023. Computing Education in the Era of Generative AI. arXiv:2306.02608. Retrieved from https://arxiv.org/abs/2306.02608

[34] Paul Denny, James Prather, Brett A. Becker, Catherine Mooney, John Homer, Zachary C. Albrecht, and Garrett B. Powell. 2021. On designing programming error messages for novices: Readability and its constituent factors. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems. Association for Computing Machinery*, New York, NY, 15 pages. DOI: https://doi.org/10.1145/3411764.3445696

[35] Paul Denny, Jacqueline Whalley, and Juho Leinonen. 2021. Promoting early engagement with programming assignments using scheduled automated feedback. In *Proceedings of the 23rd Australasian Computing Education Conference.* Association for Computing Machinery, New York, NY, 88–95.

[36] Iddo Drori, Sarah Zhang, Reece Shuttleworth, Leonard Tang, Albert Lu, Elizabeth Ke, Kevin Liu, Linda Chen, Sunny Tran, Newman Cheng, et al. 2022. A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level. *Proceedings of the National Academy of Sciences* 119, 32 (2022), e2123433119.

[37] Rodrigo Duran, Albina Zavgorodniaia, and Juha Sorva. 2022. Cognitive load theory in computing education research: A review. *ACM Transactions on Computing Education* 22, 4 (2022), 27 pages. DOI: https://doi.org/10.1145/3483843

[38] Karl Anders Ericsson and Herbert Alexander Simon. 1993. *Protocol Analysis* (1st ed.). MIT Press, Cambridge, MA.

[39] Neil A. Ernst and Gabriele Bavota. 2022. AI-driven development is here: Should you worry? *IEEE Software* 39, 2 (2022), 106–110.

[40] FauxPilot. 2023. FauxPilot - An Open-source Alternative to GitHub Copilot server. Retrieved May 24, 2023 from https://github.com/fauxpilot/fauxpilot

[41] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155. Retrieved from https://arxiv.org/abs/2002.08155

[42] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. 2022. The robots are coming: Exploring the implications of OpenAI codex on introductory programming. In *Proceedings of the Australasian Computing Education Conference.* Association for Computing Machinery, New York, NY, 10–19. DOI: https://doi.org/10.1145/3511861.3511863

[43] James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A. Becker. 2023. My AI wants to know if this will be on the exam: Testing OpenAI's codex on CS2 programming exercises. In *Proceedings of the 25th Australasian Computing Education Conference.* Association for Computing Machinery, New York, NY, 97–104. DOI: https://doi.org/10.1145/3576123.3576134

[44] Adam M. Gaweda, Collin F. Lynch, Nathan Seamon, Gabriel Silva de Oliveira, and Alay Deliwa. 2020. Typing exercises as interactive worked examples for deliberate practice in CS courses. In *Proceedings of the 22nd Australasian Computing Education Conference.* Association for Computing Machinery, New York, NY, 105–113. DOI: https://doi.org/10.1145/3373165.3373177

[45] GitHub. 2021. GitHub Copilot - Your AI Pair Programmer. Retrieved July 21, 2022 from https://github.com/features/copilot/

[46] Lisa M. Given. 2015. *100 Questions (and Answers) About Qualitative Research.* SAGE publications, Thousand Oaks, CA.

[47] John Hattie and Helen Timperley. 2007. The power of feedback. *Review of Educational Research* 77, 1 (2007), 81–112.

[48] Arto Hellas, Juho Leinonen, and Petri Ihantola. 2017. Plagiarism in take-home exams: Help-seeking, collaboration, and systematic cheating. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education.* Association for Computing Machinery, New York, NY, 238–243. DOI: https://doi.org/10.1145/3059009.3059065

[49] Michael S. Irwin and Stephen H. Edwards. 2019. Can mobile gaming psychology be used to improve time management on programming assignments? In *Proceedings of the ACM Conference on Global Computing Education.* Association for Computing Machinery, New York, NY, 208–214. DOI: https://doi.org/10.1145/3300115.3309517

[50] Dhanya Jayagopal, Justin Lubin, and Sarah E. Chasins. 2022. Exploring the learnability of program synthesizers by novice programmers. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology.* Association for Computing Machinery, New York, NY, 15 pages. DOI: https://doi.org/10.1145/3526113.3545659

[51] Ioannis Karvelas, Annie Li, and Brett A. Becker. 2020. The effects of compilation mechanisms and error message presentation on novice programmer behavior. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education.* Association for Computing Machinery, New York, NY, 759–765. DOI: https://doi.org/10.1145/3328778.3366882

[52] Ayaan M. Kazerouni, Riffat Sabbir Mansur, Stephen H. Edwards, and Clifford A. Shaffer. 2019. Student debugging practices and their relationships with project outcomes. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education.* Association for Computing Machinery, New York, NY, 1263. DOI: https://doi.org/10.1145/3287324.3293794

[53] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education* 19, 1 (2018), 43 pages. DOI: https://doi.org/10.1145/3231711

[54] Päivi Kinnunen and Beth Simon. 2010. Experiencing programming assignments in CS1: The emotional toll. In *Proceedings of the 6th International Workshop on Computing Education Research.* Association for Computing Machinery, New York, NY, 77–86. DOI: https://doi.org/10.1145/1839594.1839609

[55] Päivi Kinnunen and Beth Simon. 2011. CS majors' self-efficacy perceptions in CS1: Results in light of social cognitive theory. In *Proceedings of the 7th International Workshop on Computing Education Research.* Association for Computing Machinery, New York, NY, 19–26. DOI: https://doi.org/10.1145/2016911.2016917

[56] Michael J. Lee and Amy J. Ko. 2011. Personifying programming tool feedback improves novice programmers' learning. In *Proceedings of the 7th International Workshop on Computing Education Research.* Association for Computing Machinery, New York, NY, 109–116. DOI: https://doi.org/10.1145/2016911.2016934

[57] Antti Leinonen, Henrik Nygren, Nea Pirttinen, Arto Hellas, and Juho Leinonen. 2019. Exploring the applicability of simple syntax writing practice for learning programming. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education.* Association for Computing Machinery, New York, NY, 84–90. DOI: https://doi.org/10.1145/3287324.3287378

[58] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. 2023. Comparing Code Explanations Created by Students and Large Language Models. arXiv:2304.03938. Retrieved from https://arxiv.org/abs/2304.03938

[59] Juho Leinonen, Paul Denny, and Jacqueline Whalley. 2022. A Comparison of immediate and scheduled feedback in introductory programming projects. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1.* Association for Computing Machinery, New York, NY, 885–891. DOI: https://doi.org/10.1145/3478431.3499372

[60] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A. Becker. 2023. Using large language models to enhance programming error messages. In *Proceedings of the 54th ACM Technical Symposium*

on Computer Science Education V. 1. Association for Computing Machinery, New York, NY, 563–569. DOI: https://doi.org/10.1145/3545945.3569770

[61] 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. *Science* 378, 6624 (2022), 1092–1097. DOI: https://doi.org/10.1126/science.abq1158 arXiv:https://www.science.org/doi/pdf/10.1126/science.abq1158

[62] Q. Vera Liao, Daniel Gruen, and Sarah Miller. 2020. Questioning the AI: Informing design practices for explainable AI user experiences. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems.* Association for Computing Machinery, New York, NY, 1–15. DOI: https://doi.org/10.1145/3313831.3376590

[63] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys* 55, 9 (2023), 35 pages. DOI: https://doi.org/10.1145/3560815

[64] Dastyni Loksa and Amy J. Ko. 2016. The role of self-regulation in programming problem solving process and success. In *Proceedings of the 2016 ACM Conference on International Computing Education Research.* Association for Computing Machinery, New York, NY, 83–91. DOI: https://doi.org/10.1145/2960310.2960334

[65] Dastyni Loksa, Amy J. Ko, Will Jernigan, Alannah Oleson, Christopher J. Mendez, and Margaret M. Burnett. 2016. Programming, problem solving, and self-awareness: effects of explicit guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems.* Association for Computing Machinery, New York, NY, 1449–1461. DOI: https://doi.org/10.1145/2858036.2858252

[66] Dastyni Loksa, Lauren Margulieux, Brett A. Becker, Michelle Craig, Paul Denny, Raymond Pettit, and James Prather. 2022. Metacognition and self-regulation in programming education: Theories and exemplars of use. *ACM Transactions on Computing Education* 22, 4 (2022), 31 pages. DOI: https://doi.org/10.1145/3487050

[67] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. 2023. Experiences from using code explanations generated by large language models in a web software development e-book. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1.* Association for Computing Machinery, New York, NY, 931–937. DOI: https://doi.org/10.1145/3545945.3569785

[68] Mariam Mahdaoui, Said Nouh, My Seddiq Elkasmi Alaoui, and Mounir Sadiq. 2022. Comparative study between automatic hint generation approaches in intelligent programming tutors. *Procedia Computer Science* 198 (2022), 391–396. DOI: https://doi.org/10.1016/j.procs.2021.12.259 12th International Conference on Emerging Ubiquitous Systems and Pervasive Networks / 11th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare.

[69] Lauri Malmi, Judy Sheard, Päivi Kinnunen, Simon, and Jane Sinclair. 2020. Theories and models of emotions, attitudes, and self-efficacy in the context of programming education. In *Proceedings of the 2020 ACM Conference on International Computing Education Research.* Association for Computing Machinery, New York, NY, 36–47. DOI: https://doi.org/10.1145/3372782.3406279

[70] Jane Margolis and Allan Fisher. 2002. *Unlocking the Clubhouse: Women in Computing.* MIT press, Cambridge, MA, USA.

[71] Jessica McBroom, Irena Koprinska, and Kalina Yacef. 2021. A survey of automated programming hint generation: The HINTS framework. *ACM Computing Surveys* 54, 8 (2021), 27 pages. DOI: https://doi.org/10.1145/3469885

[72] Donald L. McCabe, Linda Klebe Trevino, and Kenneth D. Butterfield. 2001. Cheating in academic institutions: A decade of research. *Ethics & Behavior* 11, 3 (2001), 219–232. DOI: https://doi.org/10.1207/S15327019EB1103_2

[73] Robert McCartney, Anna Eckerdal, Jan Erik Mostrom, Kate Sanders, and Carol Zander. 2007. Successful students' strategies for getting unstuck. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education.* Association for Computing Machinery, New York, NY, 156–160. DOI: https://doi.org/10.1145/1268784.1268831

[74] 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), 23 pages. DOI: https://doi.org/10.1145/3359174

[75] Charlie McDowell, Linda Werner, Heather Bullock, and Julian Fernald. 2002. The effects of pair-programming on performance in an introductory programming course. *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education* 34, 1 (2002), 38–42. DOI: https://doi.org/10.1145/563517.563353

[76] Briana B. Morrison, Lauren E. Margulieux, and Mark Guzdial. 2015. Subgoals, context, and worked examples in learning computing problem solving. In *Proceedings of the 11th Annual International Conference on International Computing Education Research.* Association for Computing Machinery, New York, NY, 21–29. DOI: https://doi.org/10.1145/2787622.2787733

[77] OpenAI. 2022. DALL·E 2. Retrieved May 24, 2023 from https://openai.com/product/dall-e-2
[78] OpenAI. 2022. Introducing ChatGPT. Retrieved May 24, 2023 from https://openai.com/blog/chatgpt
[79] Tom Ormerod. 1990. Human cognition and programming. In *Psychology of Programming*. Elsevier, London, GB, 63–82.
[80] 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 *Proceedings of the 2022 IEEE Symposium on Security and Privacy*. IEEE, San Francisco, CA, 754–768. DOI: https://doi.org/10.1109/SP46214.2022.9833571
[81] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *Proceedings of the 2023 2023 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Los Alamitos, CA, 1–18. DOI: https://doi.org/10.1109/SP46215.2023.00001
[82] Hammond Pearce, Benjamin Tan, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, and Brendan Dolan-Gavitt. 2022. Pop Quiz! Can a Large Language Model Help With Reverse Engineering?arXiv:2202.01142. Retrieved from https://arxiv.org/abs/2202.01142
[83] Yulia Pechorina, Keith Anderson, and Paul Denny. 2023. Metacodenition: Scaffolding the problem-solving process for novice programmers. In *Proceedings of the 25th Australasian Computing Education Conference.* Association for Computing Machinery, New York, NY, 59–68. DOI: https://doi.org/10.1145/3576123.3576130
[84] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. arXiv:2302.06590. Retrieved from https://arxiv.org/abs/2302.06590
[85] James Prather, Brett A. Becker, Michelle Craig, Paul Denny, Dastyni Loksa, and Lauren Margulieux. 2020. What do we think we think we are doing? Metacognition and self-regulation in programming. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*. Association for Computing Machinery, New York, NY, 2–13. DOI: https://doi.org/10.1145/3372782.3406263
[86] James Prather, Lauren Margulieux, Jacqueline Whalley, Paul Denny, Brent N. Reeves, Brett A. Becker, Paramvir Singh, Garrett Powell, and Nigel Bosch. 2022. Getting by with help from my friends: Group study in introductory programming understood as socially shared regulation. In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1.* Association for Computing Machinery, New York, NY, 164–176. DOI: https://doi.org/10.1145/3501385.3543970
[87] James Prather, Raymond Pettit, Brett A. Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci. 2019. First things first: Providing metacognitive scaffolding for interpreting problem prompts. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education.* Association for Computing Machinery, New York, NY, 531–537. DOI: https://doi.org/10.1145/3287324.3287374
[88] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive difficulties faced by novice programmers in automated assessment tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research.* Association for Computing Machinery, New York, NY, 41–50. DOI: https://doi.org/10.1145/3230977.3230981
[89] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. 2017. On novices' interaction with compiler error messages: A human factors approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research.* Association for Computing Machinery, New York, NY, 74–82. DOI: https://doi.org/10.1145/3105726.3106169
[90] Brent Reeves, Sami Sarsa, James Prather, Paul Denny, Brett A. Becker, Arto Hellas, Bailey Kimmel, Garrett Powell, and Juho Leinonen. 2023. Evaluating the performance of code generation models for solving parsons problems with small prompt variations. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1.* Association for Computing Machinery, New York, NY, 299–305. DOI: https://doi.org/10.1145/3587102.3588805
[91] Ashley D. Rittmayer and Margaret E. Beier. 2008. Overview: Self-efficacy in STEM. *SWE-AWE CASEE Overviews* 1, 3 (2008), 12.
[92] Drew Roselli, Jeanna Matthews, and Nisha Talagala. 2019. Managing bias in AI. In *Companion Proceedings of The 2019 World Wide Web Conference.* Association for Computing Machinery, New York, NY, 539–544. DOI: https://doi.org/10.1145/3308560.3317590
[93] Ryan J. Salva. 2022. Preview: Referencing Public Code in GitHub Copilot | The GitHub Blog. Retrieved May 24, 2023 from https://github.blog/2022-11-01-preview-referencing-public-code-in-github-copilot/
[94] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1.* Association for Computing Machinery, New York, NY, 27–43. DOI: https://doi.org/10.1145/3501385.3543957
[95] Jaromir Savelka, Arav Agarwal, Christopher Bogart, Yifan Song, and Majd Sakr. 2023. Can Generative Pre-trained Transformers (GPT) Pass Assessments in Higher Education Programming Courses?arXiv:2303.09325. Retrieved from https://arxiv.org/abs/2303.09325

[96] Simon, Judy Sheard, Michael Morgan, Andrew Petersen, Amber Settle, Jane Sinclair, Gerry Cross, and Charles Riedesel. 2016. Negotiating the maze of academic integrity in computing education. In *Proceedings of the 2016 ITiCSE Working Group Reports.* Association for Computing Machinery, New York, NY, 57–80. DOI: https://doi.org/10.1145/3024906.3024910

[97] E. Soloway. 1986. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM* 29, 9 (1986), 850–858. DOI: https://doi.org/10.1145/6592.6594

[98] John Sweller. 1994. Cognitive load theory, learning difficulty, and instructional design. *Learning and Instruction* 4, 4 (1994), 295–312.

[99] Tabnine. 2023. AI Assistant for Software Developers | Tabnine. Retrieved May 24, 2023 from https://www.tabnine.com/

[100] Leonard Tang, Elizabeth Ke, Nikhil Singh, Bo Feng, Derek Austin, Nakul Verma, and Iddo Drori. 2022. Solving probability and statistics problems by probabilistic program synthesis at human level and predicting solvability. In *Proceedings of the International Conference on Artificial Intelligence in Education.* Springer, New York, NY, 612–615.

[101] Ross Taylor, Marcin Kardas, Guillem Cucurull, Thomas Scialom, Anthony Hartshorn, Elvis Saravia, Andrew Poulton, Viktor Kerkez, and Robert Stojnic. 2022. Galactica: A Large Language Model for Science. arXiv:2211.09085. Retrieved from https://arxiv.org/abs/2211.09085

[102] The Joseph Saveri Law Firm and Matthew Butterick. 2022. GitHub Copilot Litigation. Retrieved May 24, 2023 from https://githubcopilotlitigation.com/

[103] 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 *Proceedings of the CHI Conference on Human Factors in Computing Systems Extended Abstracts.* Association for Computing Machinery, New York, NY, 1–7.

[104] Kailas Vodrahalli, Roxana Daneshjou, Tobias Gerstenberg, and James Zou. 2022. Do humans trust advice more if it comes from AI? An analysis of human-AI interactions. In *Proceedings of the 2022 AAAI/ACM Conference on AI, Ethics, and Society.* Association for Computing Machinery, New York, NY, 763–777. DOI: https://doi.org/10.1145/3514094.3534150

[105] Danding Wang, Qian Yang, Ashraf Abdul, and Brian Y. Lim. 2019. Designing theory-driven user-centric explainable AI. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems. Association for Computing Machinery*, New York, NY, 1–15. DOI: https://doi.org/10.1145/3290605.3300831

[106] Michel Wermelinger. 2023. Using GitHub copilot to solve simple programming problems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1.* Association for Computing Machinery, New York, NY, 172–178. DOI: https://doi.org/10.1145/3545945.3569830

[107] Jonathan Yorke, Lesley Sefcik, and Terisha Veeran-Colton. 2022. Contract cheating and blackmail: A risky business? *Studies in Higher Education* 47, 1 (2022), 53–66. DOI: https://doi.org/10.1080/03075079.2020.1730313

[108] Wojciech Zaremba, Greg Brockman, and OpenAI. 2021. OpenAI Codex. Retrieved from https://openai.com/blog/openai-codex/

[109] Sarah Zhang, Reece Shuttleworth, Zad Chin, Pedro Lantigua, Saisamrit Surbehera, Gregory Hunter, Derek Austin, Yann Hicke, Leonard Tang, Sathwik Karnik, Darnell Granberry, and Iddo Drori. 2022. Automatically Answering and Generating Machine Learning Final Exams. arXiv:2206.05442. Retrieved from https://arxiv.org/abs/2206.05442

Received 19 January 2023; revised 14 June 2023; accepted 30 June 2023