Students' Use of GitHub Copilot for Working with Large Code Bases

Anshul Shah
ayshah@ucsd.edu
University of California, San Diego
USA

Anya Chernova
achernova@ucsd.edu
University of California, San Diego
USA

Elena Tomson
etomson@ucsd.edu
University of California, San Diego
USA

Leo Porter
leporter@ucsd.edu
University of California, San Diego
USA

William G. Griswold
bgriswold@ucsd.edu
University of California, San Diego
USA

Adalbert Gerald Soosai Raj
asoosairaj@ucsd.edu
University of California, San Diego
USA

Abstract

Large language models (LLMs) are already heavily used by professional software engineers. An important skill for new university graduates to possess will be the ability to use such LLMs to effectively navigate and modify a large code base. While much of the prior work related to LLMs in computing education focuses on novice programmers learning to code, less work has focused on how upper-division students use and trust these tools, especially while working with large code bases. In this study, we taught students about various GitHub Copilot features, including Copilot chat, in an upper-division software engineering course and asked students to add a feature to a large code base using Copilot. Our analysis revealed a novel interaction pattern that we call *one-shot prompting*, in which students ask Copilot to implement the entire feature at once and spend the next few prompts asking Copilot to debug the code or asking Copilot to regenerate its incorrect response. Finally, students reported significantly more trust in the code comprehension features than code generation features of Copilot, perhaps due to the presence of trust affordances in the Copilot chat that are absent in the code generation features. Our study takes the first steps in understanding how upper-division students use Github Copilot so that our instruction can adequately prepare students for a career in software engineering.

CCS Concepts

• **Social and professional topics** → **Software engineering education**; • **Human-centered computing** → *Empirical studies in HCI*; • **Computing methodologies** → **Artificial intelligence**.

Keywords

Github Copilot, large code bases, program comprehension, trust

ACM Reference Format:
Anshul Shah, Anya Chernova, Elena Tomson, Leo Porter, William G. Griswold, and Adalbert Gerald Soosai Raj. 2025. Students' Use of GitHub Copilot for Working with Large Code Bases. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE TS 2025), February 26-March 1, 2025, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3641554.3701800

SIGCSE TS 2025, February 26-March 1, 2025, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-0531-1/25/02
https://doi.org/10.1145/3641554.3701800

1 Introduction

The gap between the skills required of early career software developers and the abilities they acquire through their computer science degree program has been identified by numerous studies [3, 5, 20]. A consistent recommendation from these studies is to provide students with the opportunity to work with large code bases in their university coursework, so that they are prepared for the tasks they are typically required to do as professionals [3, 5]. For better or worse, the use of large language models (LLMs) is steadily growing among software companies. According to a survey conducted by GitHub, large language models (LLMs) are already an integral part of the software engineering workforce, with 92% of developers saying they use LLMs in and out of work and 70% of developers claiming to experience better productivity with these tools [24]. Github Copilot is the most widely-used AI developer tool, with over 20,000 companies having adopted the tool [6].

Given the importance of LLMs in industry, there is little doubt that students should learn the best practices for working with these tools [16]. However, little is known about how students that are nearing graduation currently work with such tools, with extensive prior work to understand how either professional developers [10, 24, 26, 27] or novice programmers [19, 25, 28] use LLMs for programming. Therefore, we conducted a study to understand how intermediate programmers use and experience Github Copilot to add a feature to a large code base, with a novel analysis on how students specifically used GitHub Copilot chat. As part of an upper-division software engineering course, we demonstrated to students the various GitHub Copilot features and assigned a programming task for students to complete with the help of GitHub Copilot. Following the task, we collected students' chat transcripts and administered a survey to understand students' trust in various GitHub Copilot features. Specifically, we ask the following research questions:

(1) How often do students use the various GitHub Copilot features when adding a small feature to a large code base?
(2) How do students typically engage with GitHub Copilot chat?
(3) How much do students trust the output of GitHub Copilot for code generation and code comprehension features?

2 Background and Related Work

2.1 Interaction with GitHub Copilot

Many studies related to LLMs in computing education research discuss the impacts of LLMs in introductory programming courses [7, 8, 19, 28]. However, Sarkar et al. point out that less work has covered the *user experience* of programming with LLMs [22]. The limited studies on user interactions with GitHub Copilot have examined how either novice programmers or professional developers use LLMs to write code [2, 4, 10, 17, 19, 26, 27]. Of these studies, there are two that are particularly relevant to our present study—one related to novice programmers experience with LLMs Prather et al. [19] and one related to professional developers' experience Barke et al. [2].

Using think-aloud interviews, Barke et al. performed a grounded theory analysis of how developers used GitHub Copilot [2]. Their theory distinguishes two "modes" of developer interaction with Copilot: the *exploration* mode for planning their code and the *acceleration* mode for executing their plan. The authors categorize various developer behaviors into these two modes; for example, a developer that has already developed an approach to a task that accepts a correct code suggestion constitutes *acceleration* whereas a developer closely reading and evaluating multiple suggestions for a block of code constitutes *exploration*. Importantly, at the time of the study, the Copilot chat feature had not been released. Therefore, the study does not comment on the impact of the chat feature, which allows users to ask questions about the existing code and to generate code. Therefore, our current study aims to not only connect our participants' behaviors to the interaction modes specified by Barke et al. but also to reveal specific behaviors related to Copilot chat that fall into the exploration and acceleration modes.

The study from Prather et al. is also highly relevant to our present work [19]. In their study, Prather et al. observed novice programming students complete a typical CS1 programming assignment using the code generation features of GitHub Copilot [19]. The authors introduce two *interaction patterns* called *shepherding* and *drifting* [19], which complement the *exploration* and *acceleration* patterns from Barke et al.. In *shepherding*, the novice students slowly accept, adapt, or backtrack from the code suggestion that GitHub Copilot made. In other words, the student is moving towards a correct implementation, though it typically requires some "wrestling" to produce the correct code [19]. On the other hand, *drifting* occurs when students are spending time accepting and understanding unhelpful code [19]. In drifting behaviors, the student is stagnant, unable to progress to a correct solution. Our present study will compare the student behaviors we reveal to the novice student behaviors found by Prather et al..

Our present work aims to build upon the studies by Barke et al. and Prather et al. by 1) analyzing a population *in between* novice programmers in CS1 and professional developers with years of experience and 2) revealing interaction patterns with GitHub Copilot chat, which has not been studied in these prior works.

2.2 Trust in GitHub Copilot

Developer trust in LLM tools for programming is vital to measure and understand since developer trust in a tool impacts the utility of that tool [1, 13, 18, 27]. For example, Perry et al. showed that developers using AI assistants tended to overestimate the security of the code they had written, leading the authors to conclude that LLM tools may lead to overconfidence from developers [18]. As a result, understanding student trust in GitHub Copilot features is an important step in determining 1) how educators should go about teaching students about these tools and 2) how LLM tools can be designed to improve developer trust.

Table 1: Survey on developer trust in AI systems from Amoozadeh et al. [1], adapted from Körber [12]

ID	Survey Question
S1	I trust the systems' output.
S2	The output the system produces is as good as that which a highly competent person could produce.
S3	I know what will happen the next time I use the system because I understand how it behaves.
S4	I believe the output of the system even when I don't know for certain that it is correct.
S5	I have a personal preference for using such AI systems for my tasks.
S6	Overall, I trust the AI system I use.

An important study that motivates our work comes from Amoozadeh et al., who adapted and administered a survey to measure student trust in LLMs for programming [1]. The survey, presented in Table 1, was adapted from an earlier study by Körber, who developed a theoretical model of trust in automation and a 19-item questionnaire to measure such trust [12]. In the study by Amoozadeh et al., the authors administered the survey to students in both the US and India to understand the factors that correlate to trust in AI systems [1]. They found that students in an upper-division Operating Systems course exhibited less trust in LLM tools than students in a lower-division CS2 course [1]. Though this study was conducted with students regardless of whether they had used LLM tools for programming, the study provides an important contribution to the line of work related to trust in AI systems by presenting a survey to measure trust.

The broader line of work on trust in LLM tools has discussed factors that impact trust in these tools. Liao and Sundar discuss the ideas of "trustworthiness attributes" (the factors that form the basis of trustworthiness) and "trust affordances" (cues that help the user establish trust and familiarity with the LLM tool) [13]. The authors present the MATCH model to summarize the trustworthiness attributes, trust affordances, and trust judgments from the user that all play a role in determining a user's overall trust in an AI model. Specifically, one of the three trust affordances is *transparency*, which refers to features that increase a user's understanding of decisions, behaviors, and development of the AI system. For example, explanations for why a model decision was made or external links to provide support for a claim made by the AI system would constitute affordances that increase *transparency*. Wang et al. conducted retrospective, stimulated-recall interviews with professional developers on their experience using GitHub Copilot [27]. The authors concluded that AI-powered code generation tools *lacked* trust affordances, forcing developers to rely on their personal experiences to make trust judgments [27].

Table 2: Relevant context about participants *(n = 48)*

Year in University			
Second	Third	Fourth	Fifth
9.5%	21.4%	61.9%	4.8%
Programming Languages Known			
Python	Java	C/C++	JavaScript
88.1%	95.2%	83.3%	47.6%
Largest code base students worked with before course (in LoC)			
0-1K	1K-10K	10K-1M	> 1M
52.4%	33.3%	11.9%	2.4%
Pronouns			
he/him/his		she/her/hers	
78.6%		19.0%	

3 Methods

3.1 Course Context

This study was conducted in an upper-division computer science course at UC San Diego—a large, public, research-intensive university. The course is titled "Working with Large Code Bases" and is an adapted version of the course discussed in one of our previous publications [23]. The course revolves around the `idlelib` code base and teaches students about program comprehension, project management, and unit testing in a large code base. The IDE we recommended for students to use was VSCode, although students were free to use any IDE of their choosing. The course website and all associated materials, including lecture recordings and assignment descriptions, can be found at https://cse190largecodebases.github.io/sp24/. The course was offered as an elective that students could take to fulfill their CS major requirements. The two prerequisite classes for our course were a core software engineering class in which students learn about Agile development and complete a greenfield project with a team, and a "tools and techniques" course in which students learn Git basics and command line tools.

At the beginning of the course, we administered a survey to collect student demographics, displayed in Table 2. The majority of students (83.4%) were third- or fourth-year students and roughly 85% of students had not worked on a code base larger than 10K lines of code. In line with our department's student population, 19% of our participants were women. Efforts to improve the gender diversity in the class are ongoing, yet this low percentage highlights the importance of future work to assess students' experience with LLMs for various subpopulations of students.

In the first week of class, students were asked to sign up for the GitHub Student Developer pack, which provides free access to Copilot for university students. In week 5 of the course, the instructor spent two lectures demonstrating the following Copilot features: 1) the */explain* command (which provides an explanation of the highlighted code), 2) the */fix* command (which suggests a code modification to fix the highlighted code), 3) the */tests* command (which generates a test for the highlighted code), 4) the */docs* command (which generates documentation for the highlighted code), 5) Copilot Chat (which allows natural language dialogue for users to request explanations or code snippets), 6) code generation via function docstring comments, and 7) code generation via line-by-line comments. In these lectures, the instructor first demonstrated how to use each feature in a live demonstration and then assigned a small lecture activity for students to complete using Copilot.

3.2 Programming Task

Similar to the course taught by Shah et al., students worked on the open-source `idlelib` code base throughout the term. The code base is written in Python and contains 24,000 lines of code (including test code). Halfway through the term, we assigned a task for students to complete with the help of GitHub Copilot. Students were asked to add a "Go to Definition" feature to the *idlelib* IDE that is similar to the "Go to Definition" feature in more powerful IDEs (such as VSCode, Eclipse, Pycharm, etc.). The task description that we provided students is:

> *Your specific task will be to add a menu item to the set of options that appear when a user does a <Control + Click> event (when a user holds down the Control button and clicks). This menu can also be invoked by clicking with two fingers on a Mac system or a right-click on a Windows system. When a user selects this option while on a method name, the cursor should then go to the definition of that method. If the cursor is not on a method implementation or the method does not appear in the current Python file, then it should ring the Tkinter bell to indicate that we cannot go to the definition of that method. There are other features in IDLE that also ring this bell for certain events.*

3.3 Data Collection and Analysis

3.3.1 Frequency of Features Used (RQ1). As students completed the programming task, they were required to maintain a *process journal*. Students were not awarded full credit on the assignment until they turned in their process journal. The specific instructions that we provided to students, which was distributed with the programming task, is shown below:

> *Provide a high-level description of how you used each of the following features in GitHub Copilot: /explain, /fix, /tests, /docs, Copilot chat feature, code generation via function docstring comments, and Code generation via single-line comments (i.e., one line at a time)*
> *If you did not use one of the features above, please explicitly mention that you did not use that feature.*

Our data analysis of the 48 process journals was straightforward. For each student, the lead author of this study marked which features they used. We decided that this process was not prone to any subjectivity since students explicitly mentioned that they did not use certain features and we enforced a specific format of the process journal to make analysis easier.

3.3.2 Interactions with Copilot Chat (RQ2). To understand *how* students interacted with GitHub Copilot chat (which was the feature most-used by students), we required students to upload the full chat transcript of their GitHub Copilot chat session during the programming task. Just like the process journal, we did not award full credit for the programming task until students submitted their

chat transcript. Since 1 student did not use Copilot chat at all, we collected 47 chat transcripts. The chat transcripts include each student prompt, each Copilot response, and, if any, the references that Copilot used to generate the response. See Figure 3 for an example of each of these three parts of the chat transcript.

Three authors of this paper conducted an open-coding qualitative analysis [9] of the transcripts. Given that open-coding is a bottom-up approach of generating and applying labels, the three coders first analyzed six chat transcripts and deliberated on a rough set of initial labels to form the basis of the code book. Following this initial round, the coders reviewed 12 of the transcripts (the lead author did all 12 transcripts and the two coders took up 6 transcripts each so that each transcript had two coders). Each transcript was labeled independently by two coders before the two reviewers met in a synchronous meeting to deliberate and resolve differences. This process continued until all transcripts were coded. Throughout the process, we tracked the number of disagreements between our coders. For the final round of analysis, which covered 26 transcripts, the coders agreed on 74.6% of prompts.

3.3.3 Student Trust in Copilot Features (RQ3). We administered the survey on trust in AI systems shown in Table 1 after students completed the programming task. Since we were interested in exploring potential differences in trust between Copilot features, students answered the set of survey questions in for both the code comprehension features *and* the code generation features. The survey was administered via Google Forms. Students were given 10 minutes *during lecture* to complete the survey and earn attendance credit for that lecture. For the few students who were absent from that lecture, we released an announcement to all students to ask them to complete the survey. In total, 45 of the 48 students (93.8%) completed the survey. We conducted a Paired Samples t-test [21] since each student provided a response to the survey for the comprehension features *and* generation features.

4 Results

4.1 RQ1 Results

The usage rates for each feature are shown in Figure 1. In total, 97.9% of students used GitHub Copilot chat, which was by far the most-used feature. The next two most-used features were the "/explain" command (66.7%) and "fix" command (62.5%). Interestingly, less than half of the students generated code via in-line comments (37.5%) or via function docstring comments (18.8%).

4.2 RQ2 Results

The frequency of each label in our qualitative analysis can be found in Table 3. The two most common prompts from students were for Copilot to 1) *Write code for a subtask*, occurring 24.5% of all prompts and 2) *Explain code*, occurring 20.9% of the time.

Interestingly, 15.9% of all labels were students asking Copilot to regenerate its response after being provided an unsatisfactory response. To more deeply understand what caused students to use a *Regenerate response* prompt, we counted the frequency of all the labels that immediately preceded the *Regenerate response* label. Table 4 shows that of the 87 times students asked Copilot to regenerate its response, 41.4% of those codes were caused by a *Write code for a subtask* prompt and 26.4% were caused by a *Debug code* prompt. Students asked Copilot chat to regenerate its response after an *Explain code* prompt only two times.

Finally, Table 5 shows that 63% of students asked Copilot to implement the entire feature at once.

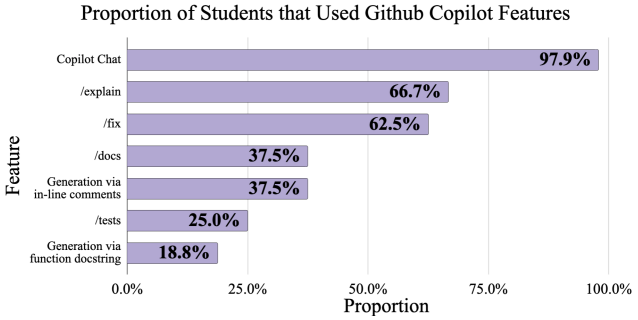

Figure 1: Frequency of self-reported student usage of the various GitHub Copilot features.

Table 3: Frequency of all labels across all student prompts in Github Copilot chat.

Label	Count	Percent
Write code for a subtask	134	24.5%
Explain code	114	20.9%
Regenerate response	87	15.9%
Debug code	86	15.8%
Implement entire feature	40	7.3%
Find code	49	9.0%
Specific implementation question	11	2.0%
Write tests	8	1.5%
Verify code	6	1.1%
Explain general Python concept	2	0.4%
Clean up code	2	0.4%
Decide type of variable	1	0.2%
None	6	1.1%
Total	545	100%

Table 4: Frequency of prompt types that led to students prompting Copilot chat with a *Regenerate response* prompt.

Label	Count	Percent
Write code for a subtask	36	41.4%
Debug code	23	26.4%
Implement entire feature	13	15.0%
Find code	12	12.6%
Explain code	2	2.3%
Verify code	1	1.2%
Clean up code	1	1.2%
Total	87	100%

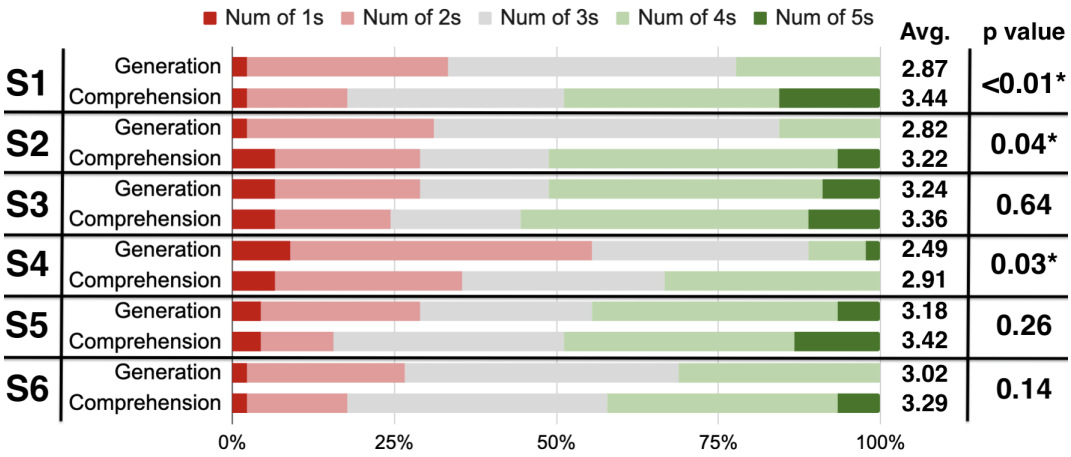

Figure 2: Comparison of student trust in GitHub Copilot features. An asterisk (*) indicates a statistically significant difference. A *1* represents *Strongly Disagree* and *5* represents *Strongly Agree.*

Table 5: Frequency of students asking Github Copilot Chat to implement the entire feature at once.

Asked to Implement Entire Feature at Once	Count	Percent
Yes	30	63%
No	18	37%

4.3 RQ3 Results

The results of the survey on trust in Copilot are shown in Figure 2. According to a Paired Samples t-test [21], which we used since each student provided a response to the survey for the comprehension features *and* generation features, we saw a statistically significant difference in items S1, S2, and S4. Based on these results, students trusted the system's output (S1), believed the system produced a response that a highly competent person could produce (S2), and believed the output even when uncertain (S4) at a *higher* rate for the *code comprehension features* than the code generation features.

5 Discussion

5.1 Students' Interaction Patterns with Copilot

Our unique data source of Copilot chat transcripts uncovered one *new* behavior and confirmed several patterns identified by Prather et al. [19] and Barke et al. [2].

The *new* behavior we noticed occurred when students asked Copilot to implement the entire feature in a single prompt, which 63% of students attempted to do. We call this interaction pattern *one-shot prompting*, as students attempt to complete the entire feature implementation in a single prompt. This interaction pattern may have emerged due to our emphasis on GitHub Copilot chat, which had not been explored in prior work to our knowledge. It may be the case that *one-shot prompting* is an affordance of interacting with GitHub Copilot chat, since users can explain the many parts to a task and provide relevant context about the code base. Such a rich description of the task may not be possible in the GitHub Copilot code generation features where users write a comment and then are shown suggested code snippets to add.

After a *one-shot prompt*, students exhibited one of two interaction patterns. One pattern was that students became stuck in a cycle of asking Copilot to regenerate its answer after being unsatisfied with the previous response or debugging the generated code. Table 4 shows that 15% of all *Regenerate response* prompts were due to asking Copilot *Implement entire feature*. A similar analysis for the *Debug code* label (though not displayed in a table) showed that 25.4% of *Debug code* prompts were used after asking Copilot to implement the entire feature at once. This pattern of students asking Copilot to *Regenerate response* or *Debug code* that it generated falls into the *drifting* interaction pattern identified by Prather et al. [19], which Vaithilingam et al. described as "debugging rabbit holes" [26].

The second interaction pattern after *one-shot prompting* that we noticed was students decomposing the task into subtasks, such as adding the menu item for "Go to Definition", accessing the word that the cursor is on, finding that word in the entire file, and moving the cursor to the right location. Table 3 shows that the *Write code for a subtask* label was the most common prompt type. This interaction type demonstrates the *acceleration* phase mentioned by Barke et al. [2], who wrote that "the main causal condition for a participant to end up in acceleration mode is being able to decompose the programming task into microtasks" [2].

Given that the majority of students attempted *one-shot prompting*, our instruction related to using LLMs for working with large code bases should prepare students for the realities of verifying and debugging the output from GitHub Copilot. Our observations support and extend the learning goals presented by Vadaparty et al. in their LLM-focused CS1 course [25]. In their work, Vadaparty et al. posit that decomposition, testing, and debugging are vital skills that students should learn to effectively interact with Copilot. Indeed, the most common behaviors we observed in students following a *one-shot prompting* approach were either 1) verifying and debugging the code produced by Copilot or 2) decomposing the task into subtasks to prompt Copilot. Therefore, these skills of decomposition, debugging, and testing may be the new skills

of writing software in the presence of LLMs as they seem to be relevant at both ends of the curriculum.

However, prior work in this area only examined the code *generation* features of Copilot, so the previously-identified interaction patterns do not examine interactions with Copilot chat. Two specific interaction patterns specific to Copilot chat that we found are 1) using the */explain* command to assist with program comprehension and 2) asking Copilot to *find code* in the code base. Overall, 66.7% of students used the */explain* command and 21.3% of students *started* their interaction with a prompt for Copilot to explain code. Similarly, 15 of the 48 (31.3%) students used Copilot chat to find code relevant to the programming task. These specific behaviors fall into the *exploration* mode identified by Barke et al. since students are finding and understanding relevant parts of the code base and letting Copilot guide their comprehension process. Given that prior studies place the amount of time dedicated to *code navigation* at 35% [11] and program comprehension up to 70% of a developer's time to a modify a large code base [15], these results illustrate the potential for LLMs to alter the program comprehension process.

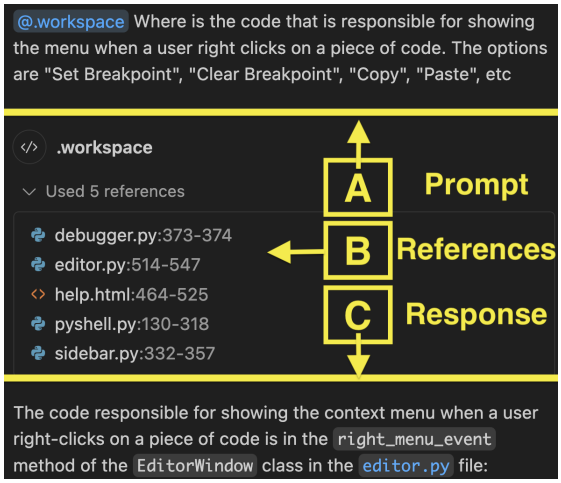

Figure 3: An example of a prompt, references, and response from GitHub Copilot chat.

5.2 Students' Trust in GitHub Copilot

The students reported a higher level of trust in the code comprehension features of GitHub Copilot than the code generation features. We argue that there are more "trust affordances" in the code comprehension features in Copilot than in the code generation features. Figure 3 demonstrates an example of one such affordance—the "References" section in Copilot chat responses. When a user asks a question to Copilot using the "@workspace" symbol, Copilot chat uses the entire code base as context for its response [14]. Such a feature promotes *transparency* in the AI model's process for generating an explanation, which Wang et al. mentions as an important pillar of establishing trust. Another key trust affordance is that in the Copilot chat output, any function or class names are "clickable" and allows students to navigate to that function or class when clicked. This feature is yet another trust affordance since it acts as a quick "fact check" mechanism for students to quickly verify Copilot's output. In contrast, there are extremely limited, if not nonexistent, trust affordances in the code generation features. This specific issue has been raised in prior work [27], and the user interface for Copilot's code generation has not changed since those previous findings. We echo previous recommendations to integrate more trust affordances in Copilot's code generation features [27].

Interestingly, students most-commonly had to ask Copilot to *Regenerate response* after asking it to *Write code for a subtask* and rarely asked Copilot to *Regenerate response* after an *Explain code* prompt, despite *Write code for a subtask* and *Explain code* occurring at similar frequencies. This difference may partially reveal why students felt they trusted code comprehension features over the code generation features: students found less errors or issues in the code comprehension output than in the Copilot-generated code.

6 Limitations

One major limitation of our findings is that our study is *observational* and does not reveal the effectiveness of specific interaction patterns. Instead, our findings are limited to *revealing* students' interaction patterns. Similarly, our results related to students' use of each feature is not reflective of student *preferences* for using those features. For example, the */fix* feature was used by 62.5% of students although some students mentioned that this feature rarely worked for them. One student wrote that "I tried to employ the /fix feature several times but never really got the results I was looking for."

Finally, our results are limited to the students in the course we offered, which may have attracted students that were motivated to work with large code bases. Similarly, the computer science department at our university is competitive, introducing another potential bias in our student population.

7 Conclusion

Our study provides a novel analysis of how upper-division students in a computer science program use and trust Github Copilot. Our findings identified a new interaction pattern called *one-shot prompting* in which students demonstrated over-reliance on the LLM tool, suggesting the importance of specific pedagogy to promote best practices when interacting with LLMs for programming. Our study also shed light on students' trust in Github Copilot after using it, with our findings showing greater student trust in code comprehension features than code generation features. Taken together, our findings reveal the varied student experiences with Github Copilot in large code bases, which involves finding and comprehending code, generating code, verifying Copilot output, and re-prompting Copilot for better suggestions. Given the rise in popularity of LLM tools, especially Github Copilot, our findings highlight the importance of specific pedagogy to improve student preparedness for a career in software engineering and future research to understand student struggles with using these tools.

Acknowledgments

This work was supported in part by NSF award 2417531.

References

[1] Matin Amoozadeh, David Daniels, Daye Nam, Aayush Kumar, Stella Chen, Michael Hilton, Sruti Srinivasa Ragavan, and Mohammad Amin Alipour. 2024. Trust in Generative AI among Students: An exploratory study. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (<conf-loc>, <city>Portland</city>, <state>OR</state>, <country>USA</country>, </conf-loc>) *(SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 67–73. https://doi.org/10.1145/3626252.3630842

[2] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 78 (apr 2023), 27 pages. https://doi.org/10.1145/3586030

[3] Andrew Begel and Beth Simon. 2008. Struggles of New College Graduates in Their First Software Development Job. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '08)*. Association for Computing Machinery, New York, NY, USA, 226–230. https://doi.org/10.1145/1352135.1352218

[4] Christopher Bull and Ahmed Kharrufa. 2024. Generative Artificial Intelligence Assistants in Software Development Education: A Vision for Integrating Generative Artificial Intelligence Into Educational Practice, Not Instinctively Defending Against It. *IEEE Software* 41, 2 (March 2024), 52–59. https://doi.org/10.1109/ms.2023.3300574

[5] Michelle Craig, Phill Conrad, Dylan Lynch, Natasha Lee, and Laura Anthony. 2018. Listening to Early Career Software Developers. *J. Comput. Sci. Coll.* 33, 4 (apr 2018), 138–149.

[6] Thomas Dohmke. 2023. *The economic impact of the AI-powered developer lifecycle and lessons from GitHub Copilot*. https://github.blog/news-insights/research/the-economic-impact-of-the-ai-powered-developer-lifecycle-and-lessons-from-github-copilot/

[7] 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 24th Australasian Computing Education Conference* (<conf-loc>, <city>Virtual Event</city>, <country>Australia</country>, </conf-loc>) *(ACE '22)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3511861.3511863

[8] 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* (<conf-loc>, <city>Melbourne</city>, <state>VIC</state>, <country>Australia</country>, </conf-loc>) *(ACE '23)*. Association for Computing Machinery, New York, NY, USA, 97–104. https://doi.org/10.1145/3576123.3576134

[9] Barney G Glaser, Anselm L Strauss, and Elizabeth Strutzel. 1968. The discovery of grounded theory; strategies for qualitative research. *Nursing research* 17, 4 (1968), 364.

[10] Mateusz Jaworski and Dariusz Piotrkowski. 2023. Study of software developers' experience using the Github Copilot Tool in the software development process. *ArXiv* abs/2301.04991 (2023). https://api.semanticscholar.org/CorpusID:255749277

[11] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32, 12 (2006), 971–987. https://doi.org/10.1109/TSE.2006.116

[12] Moritz Körber. 2018. Theoretical considerations and development of a questionnaire to measure trust in automation. https://doi.org/10.31234/osf.io/nfc45

[13] Q.Vera Liao and S. Shyam Sundar. 2022. Designing for Responsible Trust in AI Systems: A Communication Perspective. In *Proceedings of the 2022 ACM Conference on Fairness, Accountability, and Transparency* (<conf-loc>, <city>Seoul</city>, <country>Republic of Korea</country>, </conf-loc>) *(FAccT '22)*. Association for Computing Machinery, New York, NY, USA, 1257–1268. https://doi.org/10.1145/3531146.3533182

[14] Microsoft. 2024. *Chat using @workspace Context References*. https://code.visualstudio.com/docs/copilot/workspace-context

[15] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time. In *2015 IEEE 23rd International Conference on Program Comprehension*. 25–35. https://doi.org/10.1109/ICPC.2015.12

[16] Ipek Ozkaya. 2023. Application of Large Language Models to Software Engineering Tasks: Opportunities, Risks, and Implications. *IEEE Software* 40, 3 (2023), 4–8. https://doi.org/10.1109/MS.2023.3248401

[17] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. arXiv:2302.06590

[18] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do Users Write More Insecure Code with AI Assistants?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (<conf-loc>, <city>Copenhagen</city>, <country>Denmark</country>, </conf-loc>) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2785–2799. https://doi.org/10.1145/3576915.3623157

[19] 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 (nov 2023), 31 pages. https://doi.org/10.1145/3617367

[20] Alex Radermacher, Gursimran Walia, and Dean Knudson. 2014. Investigating the Skill Gap between Graduating Students and Industry Expectations. In *Companion Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE Companion 2014)*. Association for Computing Machinery, New York, NY, USA, 291–300. https://doi.org/10.1145/2591062.2591159

[21] Amanda Ross and Victor L. Willson. 2017. *Paired Samples T-Test*. SensePublishers, Rotterdam, 17–19. https://doi.org/10.1007/978-94-6351-086-8_4

[22] Advait Sarkar, Andrew D. Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence? arXiv:2208.06213

[23] Anshul Shah, Jerry Yu, Thanh Tong, and Adalbert Gerald Soosai Raj. 2024. Working with Large Code Bases: A Cognitive Apprenticeship Approach to Teaching Software Engineering. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (<conf-loc>, <city>Portland</city>, <state>OR</state>, <country>USA</country>, </conf-loc>) *(SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 1209–1215. https://doi.org/10.1145/3626252.3630755

[24] Inbal Shani. 2023. *Survey reveals AI's impact on the developer experience*. https://github.blog/2023-06-13-survey-reveals-ais-impact-on-the-developer-experience/

[25] Annapurna Vadaparty, Daniel Zingaro, David H. Smith IV, Mounika Padala, Christine Alvarado, Jamie Gorson Benario, and Leo Porter. 2024. CS1-LLM: Integrating LLMs into CS1 Instruction. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 297–303. https://doi.org/10.1145/3649217.3653584

[26] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs.Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 332, 7 pages. https://doi.org/10.1145/3491101.3519665

[27] Ruotong Wang, Ruijia Cheng, Denae Ford, and Thomas Zimmermann. 2024. Investigating and Designing for Trust in AI-powered Code Generation Tools. In *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency* (<conf-loc>, <city>Rio de Janeiro</city>, <country>Brazil</country>, </conf-loc>) *(FAccT '24)*. Association for Computing Machinery, New York, NY, USA, 1475–1493. https://doi.org/10.1145/3630106.3658984

[28] 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* (<conf-loc>, <city>Toronto ON</city>, <country>Canada</country>, </conf-loc>) *(SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 172–178. https://doi.org/10.1145/3545945.3569830