UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

An Analysis of Students' One-Shot Prompting Behaviors with GitHub Copilot in Large Code Bases

Permalink

https://escholarship.org/uc/item/6vp9j88f

Author

Rexin, Thomas

Publication Date

2025

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA SAN DIEGO

An Analysis of Students' One-Shot Prompting Behaviors with GitHub Copilot in Large Code Bases

A Thesis submitted in partial satisfaction of the requirements
for the degree Master of Science

in

Computer Science

by

Thomas Rexin

Committee in charge:

Professor Gerald Soosai Raj, Chair
Professor William Griswold
Professor Leo Porter

2025

Copyright

Thomas Rexin, 2025

All rights reserved.

The Thesis of Thomas Rexin is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2025

TABLE OF CONTENTS

THESIS APPROVAL PAGE iii

TABLE OF CONTENTS iv

LIST OF FIGURES vi

LIST OF TABLES vii

ACKNOWLEDGEMENTS viii

ABSTRACT OF THE THESIS ix

INTRODUCTION 1

Chapter 1 BACKGROUND AND RELATED WORK 3

One-Shot Prompting Behavior 3

Identified Interaction Behaviors with GitHub Copilot 4

Chapter 2 METHODS 6

Course Context 6

Programming Task 6

Data Collection and Analysis 8

Chapter 3 RESULTS 11

RQ1 Results 12

RQ2 Results 14

Chapter 4 DISCUSSION 18

Different Types of One-Shot Prompts 18

Methods to Follow-Up After a One-Shot Prompt 19

Chapter 5 CASE STUDY ANALYSIS 21

Case Study #1: Student27 21

Case Study #2: Student20 23

Case Study #3: Student6 26

Case Study #4: Student13 27
Chapter 6 LIMITATIONS 31
CONCLUSION 32
REFERENCES 33

LIST OF FIGURES

Figure 1: Sample implementation of the Go to Definition feature labeled with the sub-tasks..... 8
Figure 2: Diagram of One-Shot Prompt, Copilot Response, and Prompt after a One-Shot Prompt Labels..... 11
Figure 3: Examples of Copilot Responses..... 13
Figure 4: Sub-tree for Complete & Used by Student Copilot Response Label..... 15
Figure 5: Sub-tree for Complete & Not Used by Student Copilot Response Label..... 16
Figure 6: Sub-tree for Incomplete & Used by Student Copilot Response Label..... 16
Figure 7: Sub-tree for Incomplete & Not Used by Student Copilot Response Label..... 17

LIST OF TABLES

Table 1: Rubric used to classify Copilot responses to one-shot prompts 9
Table 2: Codebook for One-Shot Prompt Labels 12
Table 3: Codebook for Prompt After a One-Shot Prompt Labels 14
Table 4: Summary of Copilot Chat Transcript for Student27..... 22
Table 5: Summary of Copilot Chat Transcript for Student20..... 25
Table 6: Summary of Copilot Chat Transcript for Student6..... 27
Table 7: Summary of Copilot Chat Transcript for Student13..... 29

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Gerald Soosai Raj and Anshul Shah for their guidance and support. I learned so much about the research process from meeting with both of them to discuss my thesis and I plan to carry these lessons into my future work as well.

ABSTRACT OF THE THESIS

An Analysis of Students' One-Shot Prompting Behaviors with GitHub Copilot in Large Code Bases

by

Thomas Rexin

Master of Science in Computer Science

University of California San Diego, 2025

Professor Gerald Soosai Raj, Chair

Large language models (LLMs) are extensively used by software developers in industry and knowing how to best utilize LLMs as tools is an important skill for upper-division students about to enter industry. Prior work has shown that some students interact with LLMs using one-shot prompting, where students ask for the entire task implementation at once. One-shot prompting behavior could be detrimental to student learning and cause an overreliance on LLMs but there has not been much work investigating how students specifically construct one-shot prompts as well as how students' interactions with a LLM continue after a one-shot prompt. This study analyzes the interactions of upper-division students who performed one-shot prompting

with GitHub Copilot when working within a large code base. Through an open-coding qualitative analysis, this study found that students construct one-shot prompts in a variety of ways beyond copying and pasting the problem description as found in prior work. Some of the one-shot prompting behaviors identified in this work such as decomposing the task and one-shot prompting after asking comprehension questions align with developer practices from prior work. However, other one-shot prompting behaviors such as copying and pasting the problem description are similar to the behavior of novice programmers. A similar analysis on the students' prompts after a one-shot prompt revealed two main ways in which students respond to Copilot responses to their one-shot prompt which are not correct: asking for debugging help or re-prompting in an attempt to get an entirely new implementation. An in-depth case study analysis of some students' complete chat transcripts supported the existence of these behaviors and further revealed how these interaction patterns can extend throughout multiple interactions with Copilot. This study takes some first steps towards a more informed understanding of how upper-division students one-shot prompt with GitHub Copilot in a large code base and how these interaction behaviors relate to those of professional developers and novice programmers.

INTRODUCTION

The usage of large language models (LLMs) is growing among software companies. From a survey administered by GitHub, LLMs are a core part of the software engineering industry as 92% of software developers reported that they use LLMs both inside and outside work contexts and 70% software developers reported increased productivity when using LLMs [2]. GitHub Copilot is the most widely used developer tool in industry as 20,000 companies have adopted the tool [9]. Due to this increasing trend of LLM use in industry, students should learn the best practices for utilizing these tools.

To work towards this goal, Shah et al. conducted a study to understand how upper-division undergraduate students who are about to enter industry use GitHub Copilot when working with a large code base [1]. Through their analysis, they identified a one-shot prompting behavior in which students would ask Copilot for an implementation of the entire task at once. Prior studies in CS1 have identified similar behavior patterns in which students would copy and paste problem descriptions into LLM tools when working on programming tasks [3, 6, 7]. From these works, there is little doubt that some students will send one-shot prompts; however, these works have not extensively analyzed what types of one-shot prompts students construct or specifically investigated the interactions students have with an LLM after a one-shot prompt. Using the same dataset as Shah et al. [1], this study aims to investigate further into one-shot prompting behavior and understand how students interact with a LLM after one-shot prompting. Specifically, this study addresses the following research questions:

RQ1: What types of one-shot prompts do students write?

RQ2: How do students interact with Copilot *after* a one-shot prompt?

To investigate RQ1, I conducted an open-coding qualitative analysis of students' one-shot prompts to classify the different types of one-shot prompts students wrote. Similarly, to investigate RQ2, I conducted another open-coding qualitative analysis of students' follow-up prompts after their one-shot prompt to classify the different types of prompts after a one-shot prompt students wrote. Finally, to investigate more thoroughly into both research questions, I conducted a case study analysis of some students' Copilot Chat transcripts to understand more about a student's holistic one-shot prompting interaction with Copilot.

Chapter 1 BACKGROUND AND RELATED WORK

One-Shot Prompting Behavior

A prior work by Shah et al. [1] identified a one-shot prompting behavior which upper-level computer science students exhibited when interacting with GitHub Copilot Chat within a large code base. One-shot prompting happens when a student prompts Copilot Chat to implement the entire feature for a task in a single prompt. Shah et al. found that the majority of students in their dataset exhibited this one-shot prompting behavior and noted that the behavior may have emerged due to the nature of Copilot Chat which allows users to provide relevant context about the code base in the prompt which cannot be done with Copilot's code generation features. Furthermore, Shah et al. identified two interaction patterns that students exhibited after a one-shot prompt. The first occurred when students would enter a cycle of re-prompting Copilot to regenerate the code after not being satisfied with the prior response or debugging the code that was generated. The second was students decomposing the problem into sub-tasks for Copilot rather than asking for the implementation at once. I plan to build upon this prior work through 1) identifying the different types of one-shot prompts that students wrote, 2) investigating how students interact with Copilot after a one-shot prompt by identifying the types of follow-up prompts students wrote, and 3) conducting a case study analysis of some students from the dataset who did one-shot prompting to identify possible student interaction behaviors with Copilot when they do a one-shot prompt.

In addition to the prior work by Shah et al., other studies have identified a similar behavior to one-shot prompting by novice programmers [1]. These prior studies not only identified a behavior similar to one-shot prompting but also suggested that one-shot prompting could be indicative of an overreliance on LLM output, potentially hindering student learning.

Prior works by Amoozedeh et al. [7], Kazemitabaar et al. [6], and Prather et al. [3] all found that some novice programmers would copy and paste the full problem description into a single prompt for a LLM tool. Amoozedeh and Kazemitabaar both expressed similar concerns regarding student overreliance on LLM output for task completion and the potential detrimental consequences on student learning. Furthermore, Kazemitabaar discussed how problem decomposition can be a more effective strategy to support student program development over one-shot prompting but that some novice programming students from their study struggled to do so when interacting with a LLM [5, 6]. My study aims to build upon this prior work by identifying a variety of ways in which students one-shot prompt in addition to copying and pasting the full problem description into a LLM and how the type of one-shot prompt could impact the quality of the LLM response and how students interact with a LLM after their one-shot prompt.

Identified Interaction Behaviors with GitHub Copilot

Similar to the prior work by Shah et al., two studies which are particularly relevant to this current study are the study of professional developers' experiences with GitHub Copilot by Barke et al. [8] and the study about novice programmers' behaviors when using GitHub Copilot by Prather et al [4].

Through think-aloud interviews, Barke et al. identified two different modes of professional developers' interactions with Copilot: exploration and acceleration [8]. Exploration mode was utilized by developers when they would rely on Copilot to help plan out their code implementation. From the study, developers typically engaged in exploration mode when they encountered a novel or unfamiliar task and would allow Copilot to “drive” their development. On the other hand, acceleration mode occurred when a programmer utilized Copilot to execute their own already planned out actions. In acceleration mode, developers tended to direct Copilot

to follow their own development plan. It is important to note that the study by Barke et al. was conducted before the release of Copilot Chat and focuses on Copilot's code generation features. The prior work by Shah et al. already identified student prompt types from Copilot Chat that fall into the exploration and acceleration modes. This study intends to build upon these works by additionally analyzing how interaction patterns from upper-division students' whole prompting transcripts could potentially fall into the exploration and acceleration modes.

In another think-aloud study, Prather et al. observed CS1 students work on a programming task with the assistance of Copilot's code generation features and identified two behavior patterns: shepherding and drifting [4]. Shepherding is the interaction pattern of students trying to coerce Copilot towards the code they believe is necessary to complete the task. This interaction pattern typically involved some "wrestling" from the student to have Copilot produce the correct code. Drifting occurs when programmers become stuck in the cycle of trying to understand and utilize unhelpful code suggestions. Typically, programmers in this behavior drift from one suggestion to another without understanding of the validity of the suggestions. Prather et al. noted that these behaviors were not reflective of those that Barke et al. identified for professional developers due to novice programmers not being able to sufficiently understand the code that is suggested by Copilot. Shah et al. already identified how some upper-division student behaviors aligned with the shepherding and drifting behaviors identified by Prather et al. This work aims to build upon these studies by analyzing how interaction patterns from upper-division students' whole prompting transcripts could also compare to these shepherding or drifting behaviors.

Chapter 2 METHODS

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 the same course as discussed in this previous publication by Shah et al. [1] Throughout the course, students worked on the idlelib code base and learned about program comprehension, project management, and unit testing in a large code base. Students were recommended by course staff to use VSCode as their IDE, although students were free to use any IDE of their choosing. The course was offered as an elective which students could take to fulfill their computer science major requirements. There were two prerequisite classes for the course–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 the basics of Git and command line tools.

At the beginning of the quarter, students were asked to sign up for the GitHub Student Developer pack which provides free access to GitHub Copilot for university students. In week 5 of the course, the instructor spent two lectures demonstrating Copilot features within a large code base including Copilot Chat which allows natural language prompts for users to request explanations or code snippets. In the lectures, the instructor would first demonstrate each feature live before having students work on a lecture activity using Copilot.

Programming Task

Students worked on the open-source idlelib code base throughout the quarter. The code base is written in Python and contains approximately 24,000 lines of code, including test code. Halfway through the quarter, students were assigned an individual task to complete using GitHub Copilot. Students were asked to add a "Go to Definition" feature to the idlelib IDE which is

similar to the “Go to Definition” feature in more advanced IDEs, such as VSCode, Eclipse, Pycharm, etc. The task description that was provided to 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.*

For this task, the Go to Definition feature is only required to search for the method definition within the current file that is open since IDLE only allows one file to be open in the editor window at a time. For the context of this task, a one-shot prompt is considered any single prompt which asks for an implementation of the core functionality for the Go to Definition feature which can be broken down into the following list of sub-tasks:

> *Sub-Task #1:* Obtain the method name under the user's cursor
> *Sub-Task #2:* Search for the method definition in the file
> *Sub-Task #3:* Go to the method definition if it exists
> *Sub-Task #4:* If the method definition does not exist, then ring the Tkinter bell using the bell() method
>
> The below figure shows a sample solution to the task labeled with each of the sub-tasks:

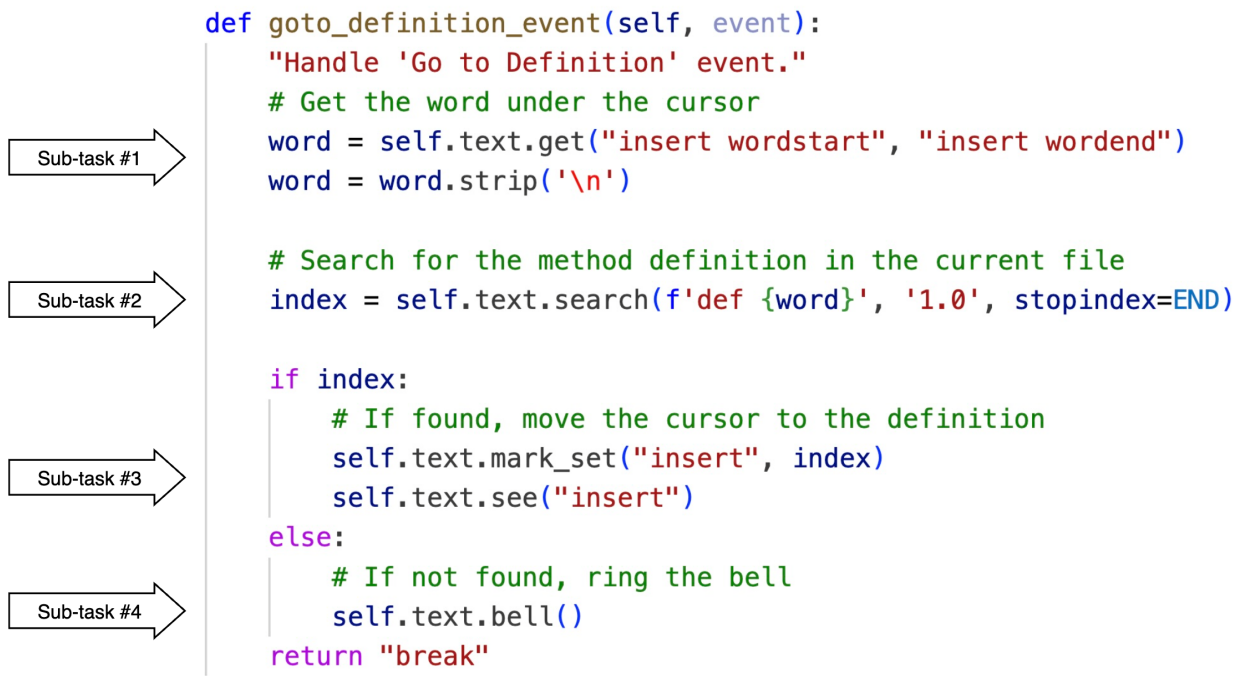


Figure 1: Sample implementation of the Go to Definition feature labeled with the sub-tasks.

Data Collection and Analysis

The data for this analysis comes from the prior work by Shah et al. [1] in which students were required to upload their complete chat transcripts with GitHub Copilot as part of the task. Shah et al. developed Copilot Chat prompt labels from the students' chat transcripts through an open-coding qualitative analysis. The dataset for this analysis is based off of the one-shot prompting behavior that Shah et al. identified in which students had at least one "Implement entire feature" prompt present in their chat transcript. For this programming task, the "Implement entire feature" label was applied if the student asked Copilot to implement all of the code for the core logic of the "Go to Definition" feature. The code to add the menu item to the right-click menu can be included but is not necessary to be considered a one-shot prompt since it is considered a minor task within the scope of the assignment. Out of the 48 students in the course, 30 of them asked Copilot to implement the entire feature at once.

From this one-shot prompting dataset of 30 students, I conducted an open-coding qualitative analysis to characterize the different types of one-shot prompts students wrote as well as the different types of prompts after a one-shot prompt. Through this labeling process, I developed a codebook for both the one-shot prompts and prompts after a one-shot prompt. In addition, I labeled the different types of responses that Copilot provided after a one-shot prompt based off of the completeness of Copilot's response and whether the provided code was incorporated in the student's final code submission for the task. The table below provides for definitions for how a Copilot Response that provides code was classified with one of either Complete or Incomplete and one of either Used by Student or Not Used by Student. To be clear, the 4 possible labels for a Copilot Response are Complete and Used by Student, Complete and Not Used by Student, Incomplete and Used by Student, and Incomplete and Not Used by Student.

Table 1: Rubric used to classify Copilot responses to one-shot prompts

Copilot Response Rubric	
Label	**Definition**
Complete	All of the necessary code to get the method under the cursor, search for the method name, and go to the method definition is provided by Copilot.
Incomplete	Some of the code to get the method under the cursor, search for the method name, or go to the method definition is not provided by Copilot.
Used by Student	At least one of Copilot's suggestions for getting the method under the cursor or searching for the method name is present in that student's final submission
Not Used by Student	None of Copilot's suggestions for getting the method under the cursor or searching for the method name are present in that student's final submission

Utilizing the one-shot prompt, Copilot response, and prompt *after* a one-shot prompt labels, I constructed a tree diagram showing the different labels and their frequencies from the

dataset. To show the relation between one-shot prompt, Copilot responses, and prompt after one-shot prompt labels, I created smaller sub-trees for each Copilot response label which shows the different one-shot prompts which led to that type of Copilot response and the different ways in which students prompted Copilot after receiving that type of Copilot response.

To investigate further into one-shot prompting behavior, I also conducted a case study analysis of the Copilot Chat transcripts for 4 students who each received a different classification of Copilot response to their one-shot prompt.

Chapter 3 RESULTS

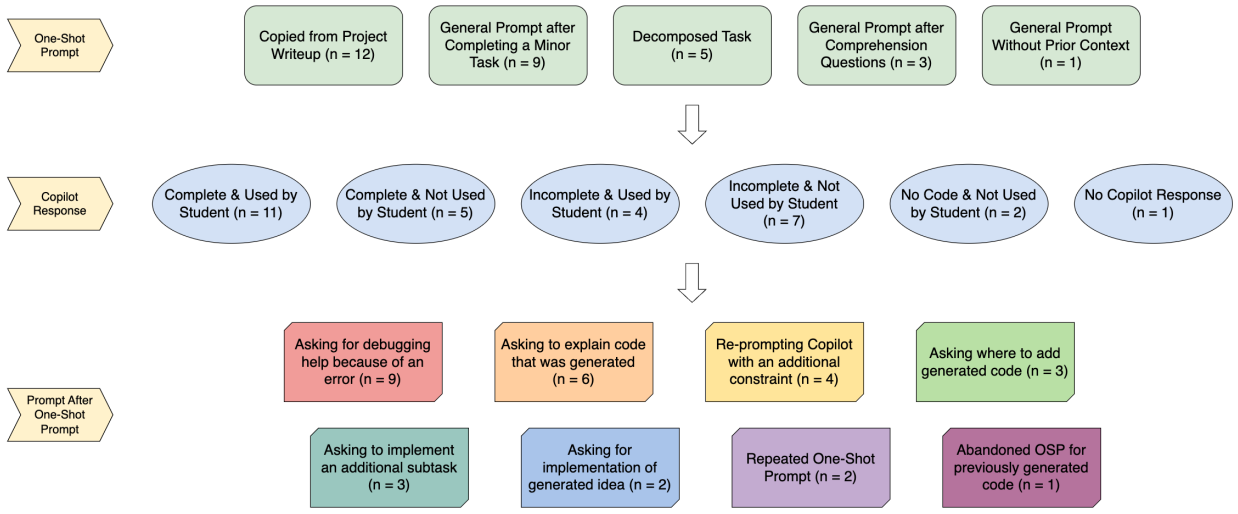


Figure 2: Diagram of One-Shot Prompt, Copilot Response, and Prompt after a One-Shot Prompt Labels

The above diagram shows all of the identified one-shot prompt, Copilot response, and prompt after a one-shot prompt labels with the frequency of students shown for each label. The results for RQ1 can be found in the first line in the diagram labeled “One-Shot Prompt.” The results for RQ2 can be found in the last line in the diagram labeled “Prompt After One-Shot Prompt.” Through conducting this analysis, I also realized that students’ interaction behaviors after a one-shot prompt depended on the type of Copilot response that they received to their one-shot prompt so those labels are also shown in the middle line in the diagram. In the following sub-sections, I will describe the different labels as well as present an example for each classification.

RQ1 Results

Table 2: Codebook for One-Shot Prompt Labels

One-Shot Prompt Labels		
Label	**Description**	**Student Example**
Copied from Project Writeup (n = 12)	The student copied the text from the assignment writeup and pasted it into the Copilot chat.	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. See the image below for an example of the menu.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.
General Prompt after Completing a Minor Task (n = 9)	In the prompts before the OSP, the student asked Copilot how to add the Go to Definition option to the right click menu and their OSP is shorter due to the existence of the prior prompts.	@workspace How can I implement the "Go To Definition" functionality in the `goto_definition` method?
Decomposed Task (n = 5)	The student broke down the task into subtasks for Copilot within their OSP.	@workspace Hello, I notice that `pyshell.py` is the file which defines rmenu_specs which has features implemented by using ctrl+click on a line in the editor. I want to add a new feature to this menu such that: 1. detect if the location of the cursor is on a function call. 2. If a function call is detected, locate the location of the function definition within the file. 3. If the function definition is present in the file, move the cursor to the location of the function definition and highlight the line. If the function definition is not present in the file, make a call to bell() to signal the user that function definition is not present in the file. For this particular feature, I only want to deal with function method calls. Classes are irrelevant to my use case.
General Prompt after Comprehension Questions (n = 3)	In the prompts before the OSP, the student planned out their task implementation through asking Copilot which file to add code to or for an explanation of the feature.	[previously asked which file they should start in for this task] @workspace How can I add a new method to handle the "Go to Definition" functionality in the EditorWindow class?
General Prompt Without Prior Context (n = 1)	The student generally asked for an implementation of the task without providing any context.	how would i go about adding a the go to defintion in my current idlelib repository

The codebook for the different one-shot prompt labels from my qualitative analysis is shown in the above table. There were a total of 5 classifications of one-shot prompts identified, one of which is Copied from Project Writeup (n = 12) in which students would copy and paste the full problem description into Copilot Chat. An interesting label which was identified is Decomposed Task (n = 5) which occurred when students would decompose the task itself into specific steps for Copilot to complete within their one-shot prompt. The remaining students all had general one-shot prompts varying context

provided to Copilot with some previously prompting to complete the minor task of adding the " Go to Definition" option to the right-click menu, some previously asking comprehension questions about the code base or feature, and others providing no prior context at all.

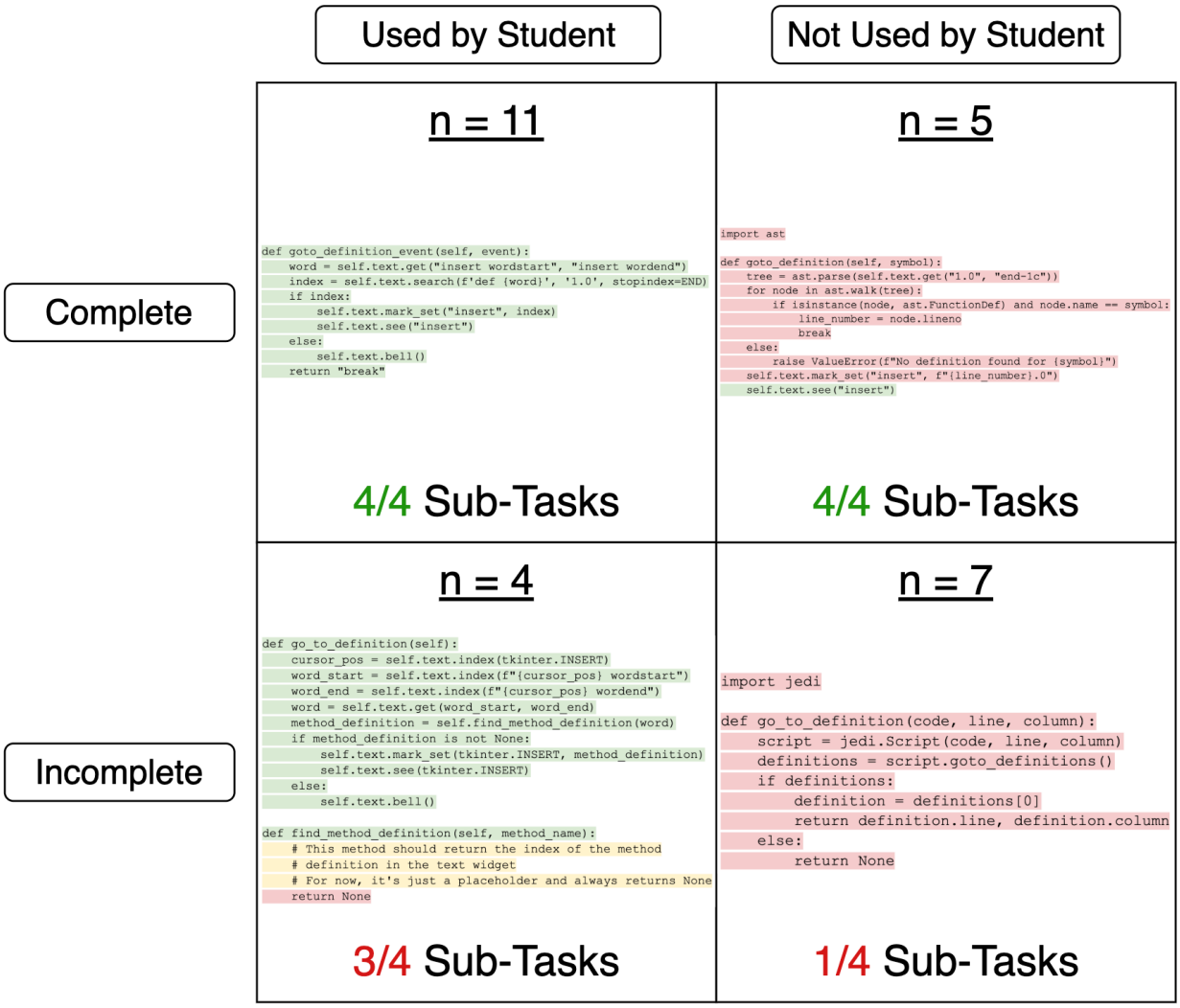


Figure 3: Examples of Copilot Responses

The figure above provides examples of Copilot responses to one-shot prompts for each of the 4 unique classifications (Complete and Used by Student, Incomplete and Used by Student, Complete and Not Used by Student, and Incomplete and Not Used by Student). Green highlighted lines of code were used by the student in the final submission, red highlighted lines of code were not used by the student in the final submission, and yellow highlighted lines of code were comments describing what code should be included by the student in the final submission. Each example also includes the number of sub-tasks

which are satisfied by the code. For example, the bottom left quadrant in the figure shows a Copilot response that was Incomplete and Used by the Student in their final solution. The entirety of the first function provided in the response was utilized by the student so it is highlighted green and this response is labeled with Used by Student. However, the response left a core function for the task undefined with just a function header as a placeholder. Therefore, because the function was not completed, Sub-task #2 was not satisfied and the response was additionally classified as Incomplete, for a final label of Incomplete and Used by Student.

RQ2 Results

Table 3: Codebook for Prompt After a One-Shot Prompt Labels

Prompt After a One-Shot Prompt Labels		
Label	**Description**	**Student Example**
Asking for debugging help because of an error (n = 9)	The student asked Copilot for debugging help either due to a generated error message or a logical error the student noticed when testing the generated code.	AttributeError: 'MultiCall' object has no attribute 'tag_defs'
Asking to explain code that was generated (n = 6)	The student asked Copilot to explain the code that was generated for the OSP.	@workspace /explain
Re-prompting Copilot with an additional constraint (n = 4)	The student asked Copilot to regenerate the response with an additional constraint (not using an import\built-in function, putting the code in a different file, etc.)	dont use that import
Asking where to add generated code (n = 3)	The student asked Copilot where to add the code that was generated for the OSP.	@workspace where should i insert this into the make_rmenu function?
Asking to implement an additional subtask (n = 3)	The student asked for code to display the Go to Definition feature when users right-click in IDLE which is needed to complete the task.	Can you help me add in the event binding? I already added it to the rmenu specs
Asking for implementation of generated idea (n = 2)	The student asked for code to implement a generated idea that Copilot had mentioned but not provided code for in the OSP.	@workspace How can I implement the find_method_definition method to search for method definitions in the current Python file?
Repeated One-Shot Prompt (n = 2)	The student was not given code for their original OSP and asks again for the code to implement the entire feature.	@workspace HOW DO I IMPLEMENT THE FEATURE
Abandoned OSP for previously generated code (n = 1)	The student primed Copilot by reminding it of previously generated code before asking for debugging help.	Here is the code you presented to me earlier: *[pasted previously generated code]* Do you remember it?

The codebook for the different prompt after a one-shot prompt labels from my qualitative analysis is shown in the above table. In addition to the Asking for debugging help because of an error (n = 9), Re-

prompting Copilot with an additional constraint (n = 4), and Asking to implement an additional subtask (n = 3) patterns which were also identified by Shah et al. [1], I identified some additional behaviors that some students exhibited. One interesting label was Asking to explain code that was generated (n = 6) in which these students would ask for an explanation for the code that was generated by Copilot in response to their one-shot prompt.

The following are all sub-trees from the diagram above which focus on the type of Copilot response and show the different one-shot prompts that led to that Copilot response in addition to the prompts after a one-shot prompt that students wrote after that Copilot response:

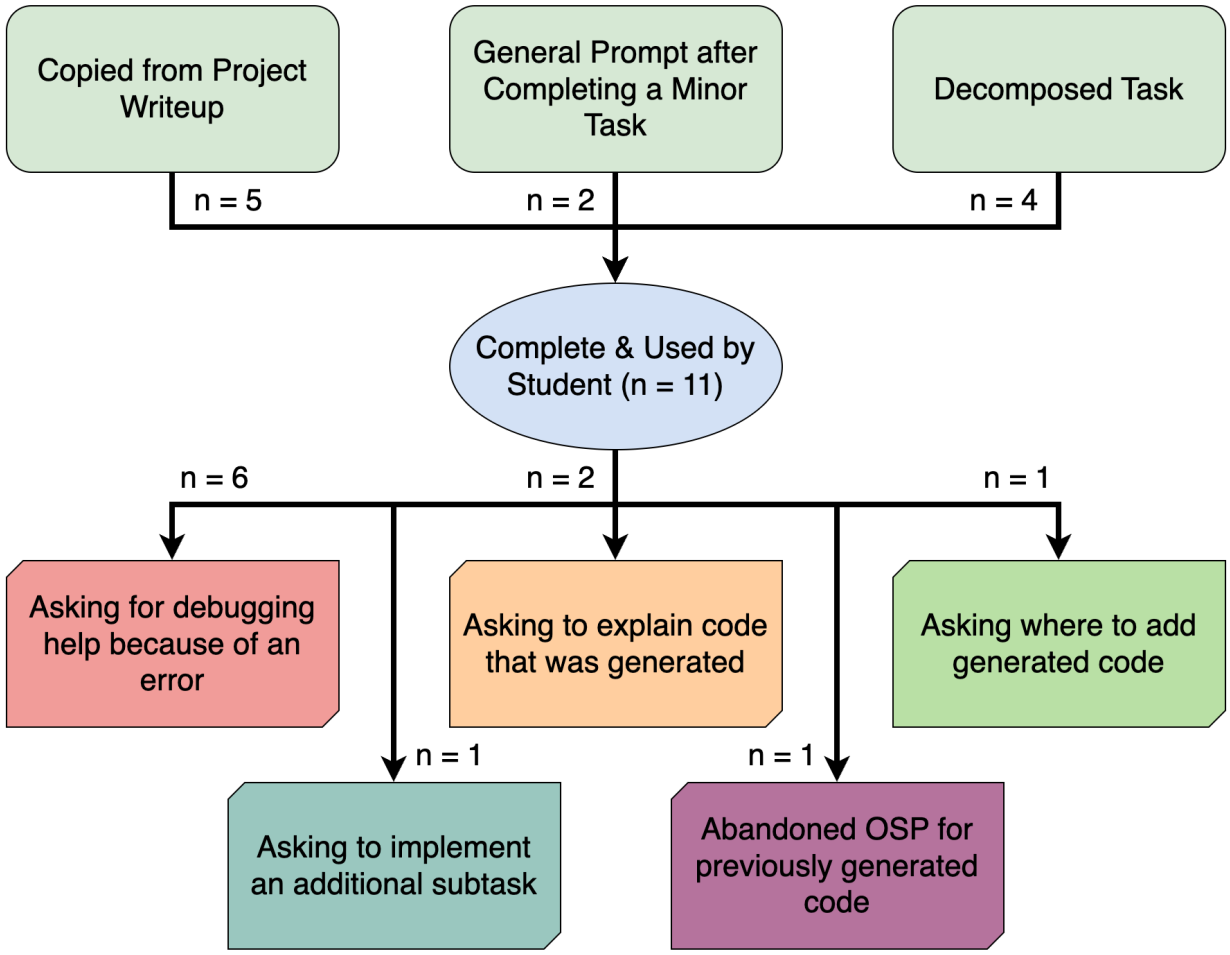


Figure 4: Sub-tree for Complete & Used by Student Copilot Response Label

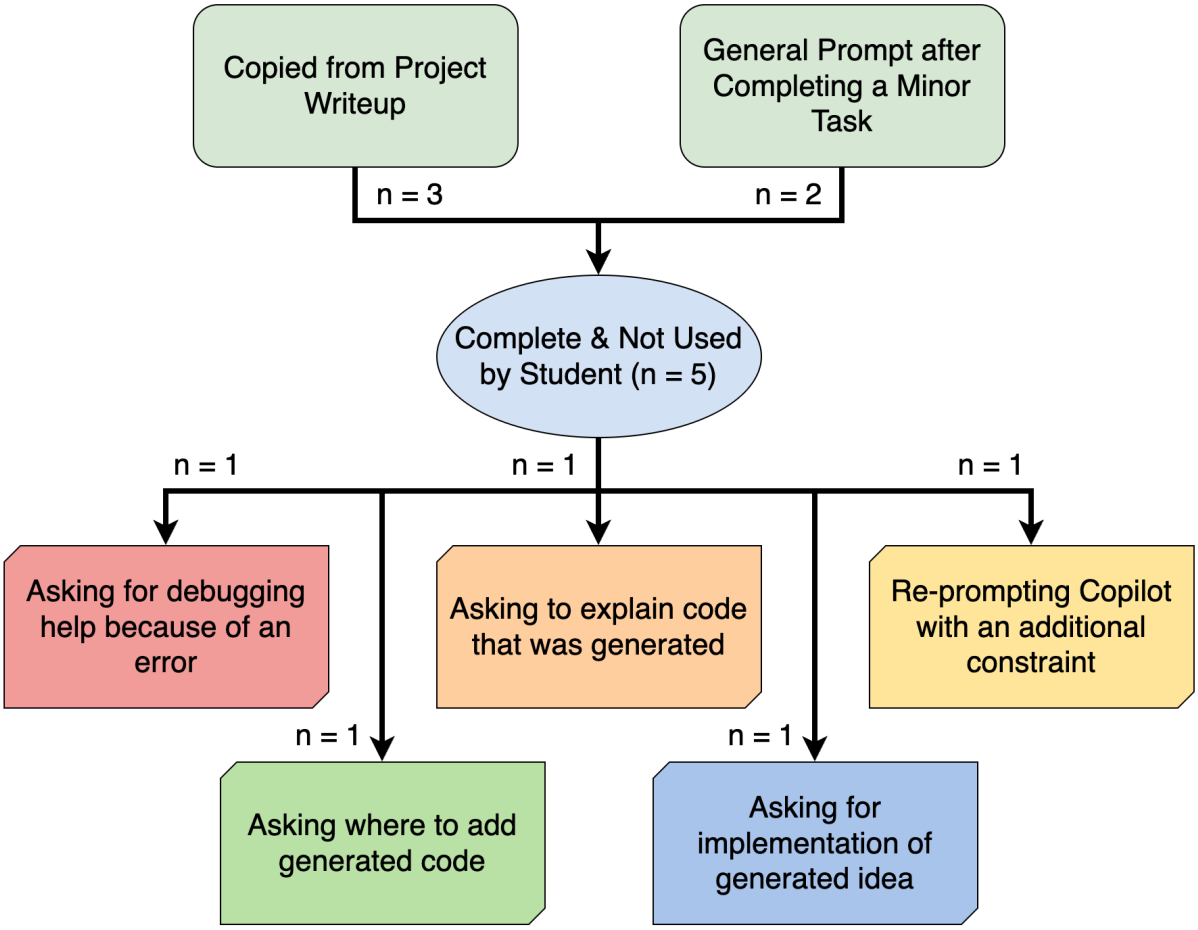


Figure 5: Sub-tree for Complete & Not Used by Student Copilot Response Label

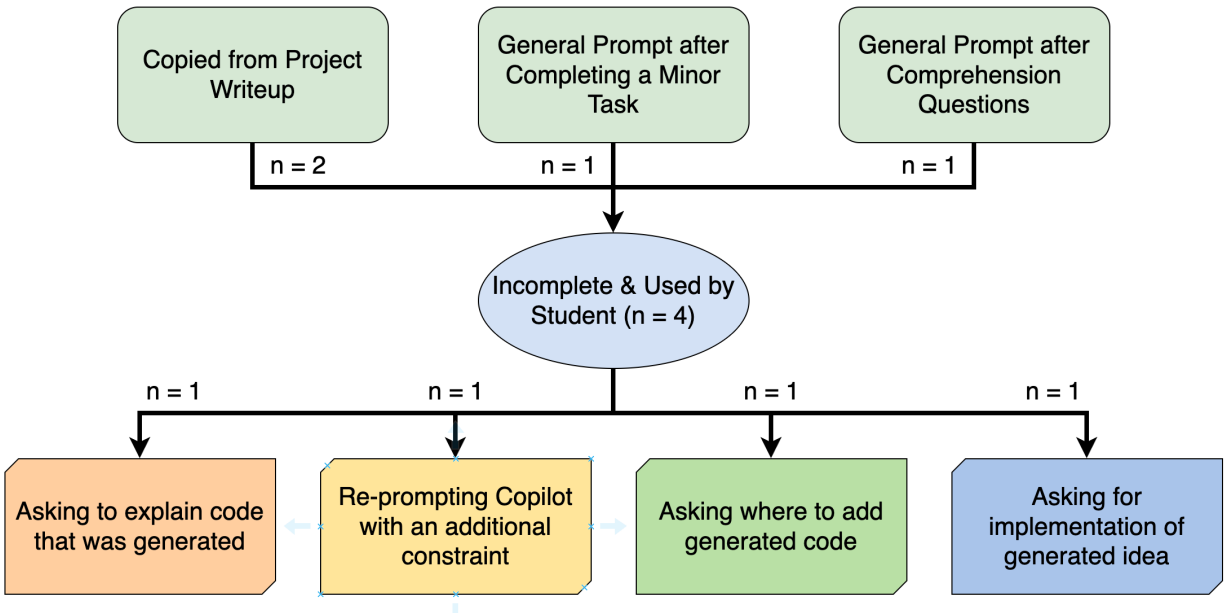


Figure 6: Sub-tree for Incomplete & Used by Student Copilot Response Label

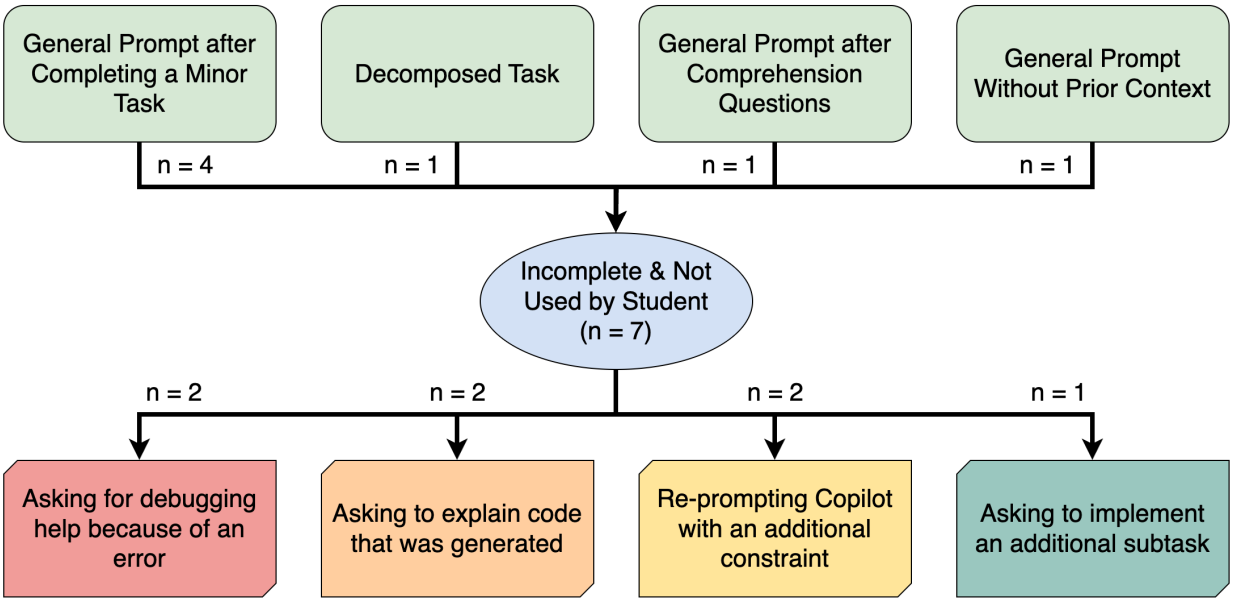


Figure 7: Sub-tree for Incomplete & Not Used by Student Copilot Response Label

Chapter 4 DISCUSSION

Different Types of One-Shot Prompts

One key takeaway from the open-coding qualitative analysis of students' one-shot prompts is that there are multiple ways that students can ask a one-shot prompt in addition to copying and pasting the problem description. From the analysis, only 12 of the 30 students who did a one-shot prompt copied and pasted the project writeup and I identified 5 total one-shot prompt labels, some of which I will discuss further in terms of their relation to either novice programmer or professional developer behavior.

Decomposed Task

One notable is that that some students would decompose the task within their one-shot prompt itself rather than decomposing the task into separate prompts. This problem decomposition exemplifies the acceleration mode discussed by Barke et al. [8] since these students had their own plan for the implementation of the task. These students utilized Copilot to execute their plan rather than relying on Copilot to help them with the planning itself. Furthermore, as discussed by Kazemitabaar et al. [5, 6], problem decomposition can be an effective strategy for novice programmers to support their program development. Therefore, the Decomposed Task method of one-shot prompting could be a potentially better one-shot prompting practice since it aligns with the behavior of professional developers and could support their own development.

Copied from Project Writeup

As similarly identified in prior work on CS1 students by Amoozedeh et al. [7], Kazemitabaar et al. [6], and Prather et al. [3], some students would one-shot prompt by copying and pasting the problem description into Copilot. This one-shot prompting type is not necessarily detrimental for upper-division students although it aligns with novice programmer behavior.

However, in contrast to the Decomposed Task label, students who Copied from Project Writeup did not have a formalized plan when prompting Copilot and relied on Copilot for much of the task completion. Potential behaviors after receiving the Copilot response to their one-shot prompt could be harmful such as exhibiting drifting behaviors [4] and falling into a debugging rabbithole as identified by Shah et al. [1].

General Prompt After Comprehension Questions

Some students in this study would present a general one-shot prompt with varying context provided to Copilot and one interesting example was a General Prompt After Comprehension Questions. Students within this label would utilize Copilot to learn more about relevant files of the code base for the feature implementation and plan out their implementation with the assistance of Copilot. This behavior aligns with the exploration mode of professional developers as identified by Barke et al. [8] since the student would allow Copilot to direct their development since they had encountered an unfamiliar task. This label suggests that program comprehension does play a role in students' workflow with GitHub Copilot within a large code base and call for future work to investigate how this population of students interact with Copilot to improve their comprehension if at all.

Methods to Follow-Up After a One-Shot Prompt

Follow-Ups to an Incorrect Copilot Response:

The open-coding qualitative analysis of this study revealed some types of prompts after a one-shot prompt that students wrote. Notably, some students would ask for debugging help due to an error while others could re-prompt Copilot with an additional constraint or even send a repeat one-shot prompt. These two paths comprise some main ways that students responded to incorrect Copilot responses since they continued to prompt Copilot to alter the code in the response. The behavior of asking for debugging help could lead to drifting behaviors identified

by Prather et al. [4] if the student does not fully understand their response from Copilot. Interestingly, the interaction behavior of re-prompting Copilot with either an additional constraint or even with a new one-shot prompt suggests that these students were dissatisfied and asked for a completely new response. This behavior could have been an effort from students to avoid potential drifting behaviors and falling into a debugging rabbithole [4].

Prompting for Program Comprehension

In their follow-up prompts, some students would ask Copilot to explain the code that was generated or ask where to add the code. These labels do not quite fit into the acceleration or exploration modes from Barke et al. yet it is encouraging that some students are prompting Copilot in this way. These labels suggest that some students may utilize Copilot to understand the code generated in response to their one-shot prompt rather than simply accepting the code without understanding its functionality. As discussed earlier, future work investigating the role that LLMs such as Copilot can play in students' program comprehension within a large code base could reveal crucial insights into best practice interactions with these tools.

Chapter 5 CASE STUDY ANALYSIS

Each case study will comprise of three sections: Takeaway, Narrative, and Commentary. The Narrative section will summarize the student's chat transcript with Copilot in a table format with student prompts in the left column and the Copilot response in the right column. Student prompts of similar themes may be grouped together into one entry and the number of prompts which comprise that entry will be specified at the end. If there is no specification of the number of prompts for an entry, then that entry corresponds to only one student prompt. Entries in the table are highlighted in green if the interaction was generally useful for the student, yellow if the interaction was neutral, and red if the interaction was generally not useful. Throughout the narrative there may be a row with a gray background that spans both columns. This row is used to mark a significant shift in the student's interaction with Copilot and includes some commentary about what the shift was. At the end of each narrative section, there is a final row which spans both columns that conveys whether the student's final code submission for the project was provided by Copilot through the chat transcript or not.

Case Study #1: Student27

Takeaway:

Instead of writing specific prompts, this student generally asked Copilot to add a menu item to the right-click menu and received a general framework back as a response. Instead of applying the example to their own use case, the student prompted Copilot to write the code. The generated code incorporated a minor error which caused the student to end up in a debugging rabbit hole and unable to get their final code from Copilot.

Narrative:

Table 4: Summary of Copilot Chat Transcript for Student27

Student Prompt	Copilot Response
One-Shot Prompt: Copied from Project Writeup	
	Responded with a useful implementation of the task but left a helper function undefined with only the function header
Asks for the implementation of the undefined helper function	
	Responds with a useful implementation of the undefined helper function
Asks how to add a new menu item to the right-click menu	
	Provides a framework showing how to add a new menu item to the right-click menu but was not implemented by the student
Asks for the code to add the new menu item to the right-click menu and how to add a keyboard shortcut **(5 prompts)**	
	Gives additional general examples of adding a new menu item to the right-click menu
Re-prompts Copilot by asking for the code to specifically add a Go to Definition menu item	
	Responds with useful code to add the Go to Definition menu item but the response includes a minor error when ringing the Tkinter bell
Asks Copilot for help debugging the error with ringing the Tkinter bell **(7 prompts)**	
	Fails to debug the student's code and offers a variety of responses including incorrect code fixes, suggestions to add print statements to aid with debugging, and completely irrelevant suggestions
Student27's final submission was not provided by Copilot	

Commentary:

Student27's interactions with Copilot started off productive and they had an almost complete implementation of the Go to Definition feature with only some minor code needed to add the feature as an option to the right-click menu. Student27 prompted Copilot for help with adding a new menu item to the right-click menu and Copilot responded with a straightforward example framework that the student could follow to add the menu item for their Go to Definition implementation. However, Student27 seemed to misunderstand that the code that Copilot provided was merely a template and asked for an implementation of the example provided in the framework. Student27 explored this framework over the next 5 prompts, still misunderstanding that the provided code is meant to serve simply as an example rather than the specific implementation itself. During this interaction, Student27 exhibited an overreliance on Copilot since they continued to prompt Copilot for the code implementation rather than utilizing the framework to write the code on their own.

Finally, Student27 specified to Copilot that the new right-click menu item should be for the Go to Definition feature that had already been generated. Copilot responded with useful code but it included a minor error when ringing the Tkinter bell if the method name under the user's cursor was not found. Over the next 7 prompts, Student27 very generally asked Copilot for help with debugging this error. Due to the general nature of their prompts, the responses that Student27 received from Copilot were incorrect and offered a variety of responses including incorrect code fixes, suggestions to add print statements to aid with debugging, and completely irrelevant suggestions. These responses from Copilot led Student27 toward drifting behaviors and into a debugging rabbithole as they were ultimately unable to debug their implementation through Copilot, resorting to writing the code on their own or consulting other help-seeking resources. Perhaps if Student27 had been more specific with their prompting, then they may have potentially had a more successful interaction with Copilot.

Case Study #2: Student20

Takeaway:

Student20 abandoned their one-shot prompt after re-prompting Copilot and switched to decomposing the problem into sub-tasks for Copilot. This student's switch into acceleration mode with their own plan proved to be more effective for this student than the one-shot prompt itself as they were able to shepherd Copilot towards their final solution.

Narrative:

Table 5: Summary of Copilot Chat Transcript for Student20

Student Prompt	Copilot Response
One-Shot Prompt: Copied from Project Writeup	
	Responds with an implementation of the task that is within the incorrect file and class
Re-prompts Copilot to add the code into a different file **(2 prompts)**	
	Responds with an incorrect implementation of the task in the new file
Student abandons OSP interaction and starts asking Copilot to implement subtasks	
Asks Copilot for code to complete just one of the task requirements (getting the method under the cursor)	
	Provides a useful helper function which correctly gets the word under the cursor
Asks for code for the Go to Definition feature using the previously generated helper function **(2 prompts)**	
	Responds with an implementation that is not useful because it uses previously generated incorrect code from the OSP
Telling Copilot not to use the incorrect class from its OSP response **(2 prompts)**	
	Provides a useful implementation of the Go to Definition function
Asks for debugging help with why the Go to Definition function is not being called when the option is clicked from the right-click menu	
	Gives an example of how to correctly bind a right-click menu option to a function
Re-prompts Copilot with a new suggestion on how to get all of the lines in text file	
	Responds with a useful implementation of Go to Definition but has placeholders instead of moving the cursor to the method definition if found or ringing the Tkinter bell if not found
Tells Copilot to modify the code to ring the Tkinter bell if no definition is found	
	Correctly modifies the code with this change
Tells Copilot to modify the code to move the cursor to the method definition if the definition is found	
	Correctly modifies the code with this change and the code outputted is the student's final submission
Student20's final submission was provided by Copilot	

Commentary:

Copilot's initial response to Student20's one-shot prompt suggested adding code in an irrelevant file. Even after re-prompting Copilot by specifying the correct file that the code should be added to, the code suggestions that Copilot output were completely incorrect. The incorrect responses that Copilot provided likely motivated the student to transition to asking Copilot to implement smaller subtasks instead of the entire feature at once. This problem decomposition strategy proved to be successful for this student as they were able to shepherd Copilot towards their final solution through more focused interactions, entering acceleration mode. After re-prompting Copilot to not use the incorrect file that it had originally suggested, the rest of Student20's interactions with Copilot were useful and straightforward with only a few minor debugging and subtask prompts that built on top of each other to form a cohesive solution. Switching to decomposing the problem into sub-tasks allowed the student to shepherd Copilot towards their final code submission.

Case Study #3: Student6

Takeaway:

Student6 was very specific in their one-shot prompt through decomposing the problem into specific subtasks to be completed. While in acceleration mode, this student was able to successfully shepherd Copilot toward their final solution through a specific follow-up prompt.

Narrative:

Table 6: Summary of Copilot Chat Transcript for Student6

Student Prompt	Copilot Response
One-Shot Prompt: Decomposed Task	
	Responded with a complete and useful implementation of the task with only a minor error
Asks Copilot for debugging help and gives in-depth examples of the correct functionality	
	Provides a useful response which fixes the error with the previous generated response
Asks if a crucial line of code for the Go to Definition logic is correct	
	Responds by revising the line to account for a potential edge case
Student6's final submission was provided by Copilot	

Commentary:

Starting off in acceleration mode, Student6 was very clear in their one-shot prompt about the expected behavior of the Go to Definition feature and decomposed the problem into numbered steps for how the feature should function. Even after Copilot's first response was erroneous, Student6 again was very specific in their next prompt and detailed why the current implementation was incorrect as well as a clear example of the correct functionality of the feature using examples. The specificity of Student6's prompts and initial decomposition of the problem may have contributed towards the effectiveness of their interactions with Copilot. Both breaking down the task into manageable subtasks within the one-shot prompt and presenting examples in the follow-up debugging prompt to Copilot may have made the prompts more straightforward for Copilot to complete.

Case Study #4: Student13

Takeaway:

This student quickly abandoned their one-shot prompt without trying to debug and switched to decomposing the problem into sub-tasks for Copilot. This strategy was unsuccessful for this student and they were unable to shepherd Copilot toward their final code submission.

Narrative:

Table 7: Summary of Copilot Chat Transcript for Student13

Student Prompt	Copilot Response
Asks for explanations of some parts of the IDLE code base (i.e. the code responsible for the right-click menu, Tkinter methods) **(3 prompts)**	
	Responds with helpful explanations about some core aspects of the right-click menu functionality
One-Shot Prompt: Previously Prompted Copilot to Complete a Minor Task	
	Provides an incomplete implementation of the task with two core functions left undefined as function headers
Student abandons OSP interaction and starts asking Copilot to implement subtasks	
Asks for information about how the right-click menu could be expanded **(2 prompts)**	
	Responds with useful examples of how an option could be added to the right-click menu
Asks for Copilot to complete the code for a subtask (checking if the user's cursor is touching a method name)	
	Provides an implementation that is incorrect and not useful since the code only checks if the cursor is over a word which could be a valid method name
Re-prompts Copilot by specifying that the code should check if the user's cursor is touching a method that is defined in the current file	
	Responds with code that uses a complex built-in functionality
Re-prompts to not use the complex built-in functionality	
	Provides code that does not use the previous built-in functionality but is incorrect
Student switches from asking Copilot for the Go to Definition functionality to asking for code to bind their own implementation of the function to the right-click menu option	
Asks for subtask of binding their new Go to Definition function to the option in the right-click menu	
	Responds with useful code which the student implements into their final submission
The core Go to Definition logic of Student13's final submission was not provided by Copilot	

Commentary:

Student13's one-shot prompt received a response from Copilot which was both incomplete and not useful for the student since two of the core functions for the Go to Definition feature were left undefined. Because of the incompleteness of Copilot's response, Student13 abandoned their one-shot prompt and transitioned to decomposing the problem for Copilot into subtasks like explaining the right-click menu part of the code base and later for an implementation of checking if the user's cursor is touching a method name. Copilot seemed to misunderstand Student13's prompt and responded with code which correctly got the word under the user's cursor but simply checked if the word could syntactically be a method name, not if it is a method that is defined. Even after attempting to shepherd Copilot by re-prompting it to instead check if the word matches the name of a method defined in the current file, Copilot responded with an implementation which used a complex built-in functionality which may have been difficult for the student to understand because they followed-up by prompting Copilot not to use that built-in functionality.

Finally, Student13 transitioned away from asking Copilot to help write the Go to Definition feature and seemed to either write the correct code on their own or consult other help-seeking resources to prevent potential drifting behavior. Student13 instead ended their interaction by prompting Copilot to write code to bind their own Go to Definition implementation (which they had not received from Copilot) to the option in the right-click menu. Student13's interactions with Copilot were not effective since they had to construct the code themself or consult other help-seeking resources in order to get the core functionality of the Go to Definition feature for their final submission even after attempting to shepherd Copilot.

Chapter 6 LIMITATIONS

One key limitation of these findings is that there was only one researcher who conducted all of the open-coding qualitative analysis. The researcher did meet regularly with advisors to iteratively develop the open-coding labels and presented examples for each label when formulating them. Another major limitation of these findings is that this study is observational in nature and does not investigate the effectiveness of specific interaction patterns. These findings are limited to revealing some potential student interaction behaviors.

These results are limited to the scope of the idlelib code base utilized in this course at UC San Diego. In addition, the computer science department at UCSD is competitive which could hinder the generalizability of these results. Finally, the quality of LLMs such as Copilot tends to change often as improvements are made to these systems.

CONCLUSION

This study presents an in-depth analysis of upper-division students' one-shot prompting behaviors when working with a large code base. Our findings identified different ways that students write one-shot prompts in addition to copying and pasting the problem description as identified in prior work such as decomposing the task within the one-shot prompt or asking comprehension questions before the one-shot prompt. Furthermore, this study identified two main interaction patterns in which students respond to an incorrect Copilot response: debugging or re-prompting for a new response. Student interaction patterns of prompting for program comprehension questions both before and even after a one-shot prompt also call for potential future work investigating how students utilize LLMs for program comprehension within a large code base. Through a case study analysis of some students' Copilot Chat transcripts, we also presented more thorough insights into students' specific interaction behaviors and how they align with the behaviors of both professional developers and novice programmers. All in all, these findings reveal some of the specific student experiences with one-shot prompting GitHub Copilot when working with a large code base. Due to the rise of LLM tools within both academic and industry contexts and the prevalence of one-shot prompting behavior with these tools, this study's findings accentuate the importance of teaching students best interaction behaviors with LLMs to prepare them for a career in software engineering.

REFERENCES

[1] 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 (SIGCSETS 2025). Association for Computing Machinery, New York, NY, USA, 1050–1056. https://doi.org/10.1145/3641554.3701800

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

[3] James Prather, Brent N Reeves, Juho Leinonen, Stephen MacNeil, Arisoa S Randrianasolo, Brett A. Becker, Bailey Kimmel, Jared Wright, and Ben Briggs. 2024. The Widening Gap: The Benefits and Harms of Generative AI for Novice Programmers. In Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1 (ICER '24), Vol. 1. Association for Computing Machinery, New York, NY, USA, 469–486. https://doi.org/10.1145/3632620.3671116

[4] 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 (February 2024), 31 pages. https://doi.org/10.1145/3617367

[5] Kwon, K. and Cheon, J. 2019. Exploring problem decomposition and program development through block-based programs. International Journal of Computer Science Education in Schools. 3, 1 (Apr. 2019), 3–16. https://doi.org/10.21585/ijcses.v3i1.54.

[6] Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara Jane Ericson, David Weintrop, and Tovi Grossman. 2024. How Novices Use LLM-based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment. In Proceedings of the 23rd Koli Calling International Conference on Computing Education Research (Koli Calling '23). Association for Computing Machinery, New York, NY, USA, Article 3, 1–12. https://doi.org/10.1145/3631802.3631806

[7] Matin Amoozadeh, Daye Nam, Daniel Prol, Ali Alfageeh, James Prather, Michael Hilton, Sruti Srinivasa Ragavan, and Amin Alipour. 2024. Student-AI Interaction: A Case Study of CS1 students. In Proceedings of the 24th Koli Calling International Conference on Computing Education Research (Koli Calling '24). Association for Computing Machinery, New York, NY, USA, Article 13, 1–13. https://doi.org/10.1145/3699538.3699567

[8] 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 (April 2023), 27 pages. https://doi.org/10.1145/3586030

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