Exploring Student-AI Interactions in Vibe Coding

Francis Geng
University of California San Diego
La Jolla, California, USA
fgeng@ucsd.edu

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

Haolin Li
University of California San Diego
La Jolla, California, USA
hal180@ucsd.edu

Nawab Mulla
University of California San Diego
La Jolla, California, USA
nmulla@ucsd.edu

Steven Swanson
University of California San Diego
La Jolla, California, USA
sjswanson@ucsd.edu

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

Daniel Zingaro
University of Toronto Mississauga
Toronto, Ontario, Canada
daniel.zingaro@utoronto.ca

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

arXiv:2507.22614v1 [cs.HC] 30 Jul 2025

Abstract

Background and Context. Chat-based and inline-coding-based GenAI has already had substantial impact on the CS Education community. The recent introduction of "vibe coding" may further transform how students program, as it introduces a new way for students to create software projects with minimal oversight.

Objectives. The purpose of this study is to understand how students in introductory programming and advanced software engineering classes interact with a vibe coding platform (Replit) when creating software and how the interactions differ by programming background.

Methods. Interview participants were asked to think-aloud while building a web application using Replit. Thematic analysis was then used to analyze the video recordings with an emphasis on the interactions between the student and Replit.

Findings. For both groups, the majority of student interactions with Replit were to test or debug the prototype and only rarely did students visit code. Prompts by advanced software engineering students were much more likely to include relevant app feature and codebase contexts than those by introductory programming students.

CCS Concepts

• **Social and professional topics** → **Computing education**.

Keywords

Large Language Models, Vibe Coding, Novice Programmers, Observation Study

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
Conference 2025, Location
© 2025 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The advent of GenAI has had considerable impact on computing education with GenAI being capable of solving course assignments [5, 7], offering skilled AI tutors to students [6, 16, 20, 23], altering how we teach [15, 36], and changing the skills we need to teach [17, 26, 37]. LLMs, such as Copilot and ChatGPT, have changed the workflow of programming for professionals [2] as well as for students [26, 37]. As the CS education research community works to identify the impact of GenAI on student learning (with mixed findings [13, 21, 29, 36]), platforms such as Replit [30], Copilot Agents [22], and Cursor [4] have the potential to alter the landscape even further.

The term "vibe coding" [12] was introduced this past year to describe the process of creating software with minimal effort, specifically where the user does not systematically review, test, or understand all the code produced and is focused on the end product [12, 31, 32]. Existing studies of vibe coding have primarily examined how experienced developers and professional programmers engage in this workflow, leaving a notable gap in our understanding of how computing students might approach it. This gap raises new questions for the CS education community, including how programming students of various levels might approach vibe coding. If we can understand how students use AI-powered tools for vibe coding, we will be better equipped to help support students learn to use them effectively. To study these behaviors, we examine how programming students interact with a vibe coding platform in an observational experiment. As vibe coding could potentially lower the barrier to building software projects, our study focuses on how both introductory *and* advanced CS students approach vibe coding.

Our observational study recruited students from an introductory programming class and an advanced software engineering class. Each student then came to a study session where they were asked to create a web application by vibe coding in Replit. We qualitatively analyzed the recordings to label the types of interactions students have with Replit. We report on the behaviors of students overall and also compare the behaviors of students from an introductory class with those from an advanced class, focusing on their interactions with the AI tool.

We find that students, in general, interact with the code produced by Replit very rarely and the majority of interactions with Replit are to prompt it for changes (primarily debugging) and interactions with the prototype (mostly testing). This shift in focus to testing and debugging emphasizes the importance of teaching these skills in CS curricula. When comparing the introductory and advanced students, we find that advanced students are more willing to work with the code produced by the AI (although this is still a very small fraction of their interactions with the tool) and that introductory student prompts are less apt to include the relevant context. These findings suggest that advanced students are interacting with such tools in a more sophisticated manner, supporting the need to teach students of all levels how to read code and express their AI prompts with programming context.

2 Background and Literature Review

2.1 Research on Vibe Coding

In this section, we explore how vibe coding has been defined, summarize the existing research on vibe coding, and compare vibe coding to agentic coding as well as first-generation coding with generative AI.

2.1.1 What is Vibe Coding? The term "vibe coding" was introduced in February 2025 by Andrej Karpathy, a computer scientist and AI researcher who co-founded OpenAI. The following tweet text is the definition of vibe coding as written by Karpathy [12]:

"There's a new kind of coding I call 'vibe coding', where you fully give in to the vibes, embrace exponentials, and forget that the code even exists. It's possible because the LLMs (e.g., Cursor Composer w Sonnet) are getting too good. Also I just talk to Composer with SuperWhisper so I barely even touch the keyboard. I ask for the dumbest things like 'decrease the padding on the sidebar by half' because I'm too lazy to find it. I 'Accept All' always, I don't read the diffs anymore. When I get error messages I just copy paste them in with no comment, usually that fixes it. The code grows beyond my usual comprehension, I'd have to really read through it for a while. Sometimes the LLMs can't fix a bug so I just work around it or ask for random changes until it goes away. It's not too bad for throwaway weekend projects, but still quite amusing. I'm building a project or webapp, but it's not really coding — I just see stuff, say stuff, run stuff, and copy paste stuff, and it mostly works."

That is, vibe coding involves asking a GenAI tool for code, but not reading that code. To debug, one can paste error messages into the AI, or ask for random changes until it's fixed. Karpathy notes that this approach is good enough for weekend projects.

To be clear, Karpathy (an expert computer scientist and AI researcher) doesn't need to vibe code. But, because it works for him, he uses it. It's okay if vibe coding only *"usually"* fixes the bug, or only *"mostly"* works, because Karpathy can just fix the code in these cases. That is, it's perhaps unsurprising that experts, with deep knowledge of the underlying code, can ride the vibes until something goes wrong, fix the problem, and continue vibing. This leaves us with two questions: *what does vibe coding actually look like when carried out by experts?* And, *what happens when novices and intermediate programmers vibe code?* There is initial research into the first question that we describe next; and answering the second question is a goal of the present study.

2.1.2 What Do Expert Developers Do When They Vibe Code? The most comprehensive study of developers vibe coding is the study by Sarkar and Drosos [32]. These authors studied YouTube and Twitch videos of experienced developers who were vibe coding while thinking aloud. The authors chose to analyze those videos where the programmer self-described what they are doing as vibe coding, rather than attempting to impose some criteria of what vibe coding is. That's because the definition of vibe coding is not up to us, but is and will continue to be negotiated by communities of developers who use it in practice.

The authors found that these programmers benefited from their expertise in several ways, such as when evaluating, testing, or manually editing code, and when including detailed technical specifications in their prompts. They are able to rapidly assess code and make immediate judgments about its suitability. The authors use the term "material disengagement" to emphasize the increasing distance between developers and their code in a vibe coding workflow. That is, these developers work on their code not directly, but mediated through GenAI.

The authors' qualitative analysis led to nine top-level categories, including what developers want to build with vibe coding, plans for how they'll build it, the vibe coding workflow, prompting strategies, and debugging. As a sample of what the authors found, we highlight just three of their many insights: 1) Some developers begin with expectations of complete success, but often need to temper those expectations to partial (e.g., 80%) success; 2) Due to its conversational nature, vibe coding often helped these developers go beyond their initial visions. At the same time, vibe coding could be so fast as to lock a developer into a suboptimal plan before they know it; 3) Developers used both single-objective prompts and multi-objective prompts, the latter of which may include directives to the AI about multiple unrelated requirements.

Again, all of these findings are from expert programmers with considerable understanding of programming languages, or the affordances of AI models, or both. In contrast, the novice and intermediate programmers in our study have substantially less experience in software development and AI, making our study a valuable complement to prior work and helping to fill a critical gap in our understanding of vibe coding among non-experts and students.

2.1.3 Vibe Coding vs. Other AI Programming Workflows. While definitions of AI-assisted programming workflows are in flux, we do wish to contrast vibe coding against two other workflows in order to further position vibe coding.

First, we distinguish vibe coding from first-generation GenAI programming workflows from 2022 and 2023, where programmers would prompt for each function and the AI completed the code [32]. (Chat interfaces hadn't been integrated into the GenAI tools yet!) Vibe coding, as we have described, is much further abstracted from the code, allowing programmers to delegate significantly larger tasks to the AI. That said, it is *not* entirely hands off [32].

Second, we distinguish vibe coding from "agentic coding" [31], in which the intention *is* to be hands off. The human is not in the loop: *"agentic coding enables autonomous software development through goal-driven agents capable of planning, executing, testing, and iterating tasks with minimal human intervention"* [31]. At least, this is how agentic coding is defined for experienced developers. We wonder to what extent our comparably less experienced students' "vibe coding" looks like agentic coding.

2.2 How Programmers Interact with AI Programming Tools

Though research related specifically to "vibe coding" is in its early stages, a larger body of work has studied how programmers interact with generative AI programming tools. In this section, we will describe the literature related to how experienced and inexperienced programmers interact with AI tools such as GitHub Copilot and ChatGPT.

2.2.1 How Experienced Programmers Use AI Tools. Broadly, the software engineering literature on AI programming tools discussed how these tools can change the software development process [3, 11, 24, 40], how developers use AI tools [2, 8, 34, 35], and challenges they face when using those tools [19, 25, 38, 39]. In this section, we will focus on how experienced programmers use AI tools. This helps inform how advanced software engineering students (intermediate-level programmers) might interact with AI in a vibe coding platform.

Barke et al. conducted an observational study of 20 programmers with a range of prior experience — from professional to occasional programmers — to theorize how programmers use GitHub Copilot [2]. Their theory posits that programmers interact with Copilot in two modes: exploration mode, in which the programmer uses Copilot to understand how to get started with a task; and acceleration mode, in which programmers are aware of the necessary steps and use Copilot to speed up implementation [2]. Recent observational studies of how programmers use AI tools support the theory above, shedding light on the prompting and verification behaviors among developers [18, 28, 38]. Notably, Liang et al. found that programmers used various techniques to verify AI-generated code output, such as by scanning the code for keywords, using a compiler to detect issues, executing the code, or examining the code in depth [18]. In addition, Vaithilingam et al. and Perry et al. both discussed challenges developers face, with Vaithilingam et al. highlighting the difficulty of comprehending and debugging AI-generated code, and Perry et al. finding that programmers who used AI tools were more likely to write insecure code that included more system vulnerabilities [25]. Other studies focused on how developers process the output of AI tools. For example, Liao and Sundar discussed how users make trust judgments of AI output, engaging in either systematic processing (requiring a careful evaluation of the output) or heuristic processing (involving the use of heuristics to make quick, yet error-prone, trust assessments) [19].

In the computing education research space, two recent studies by Shihab et al. [35] and Shah et al. [33] studied upper-division CS students' interactions with AI tools while working on existing codebases. Shihab et al. conducted a within-subjects experiment and found that students completed the tasks with Copilot 35% faster, and that Copilot reduced the amount of time programmers spent writing code by 11 percentage points and the amount of time spent performing web searches reduced by 12 percentage points [35]. Similarly, Shah et al. analyzed the prompting strategies of 48 students as they completed a task to add a feature to an open-source codebase, showing that students preferred to interact with Copilot chat to comprehend or generate code and reporting a higher trust in Copilot's code comprehension features than its code generation features [33].

The prior studies mentioned in this section describe workflows with AI tools where programmers are still actively interacting with the generated code. For example, both Barke et al., Liang et al., and Vaithilingam et al. highlight how programmers still spent time reviewing and understanding the code output, especially when trying to debug. However, as discussed in Section 2.1, vibe coding platforms introduce an even greater degree of abstraction between programmers and their code compared to earlier AI tools like Copilot and ChatGPT. This increased separation makes it especially important to study how the workflow affects intermediate programmers — such as our advanced software engineering students — who have enough experience to understand and occasionally review code, but may not be proficient or confident in debugging complex implementations by hand.

2.2.2 How Novice Programmers Use AI Tools. Plenty of work has analyzed how students program with AI tools [1, 5, 21, 28, 29], how these tools impact the help-seeking landscape [6, 10, 16, 23], and how to teach in the age of generative AI [7, 16, 17, 26, 37]. Given our study's focus on how novice and intermediate programmers approach vibe coding, we will discuss prior works that have observed how novices interact with various AI tools for programming.

Novice programmers tend to exhibit ineffective prompting and verification strategies, with multiple studies showing that students tried to write prompts that would solve the entire problem at once [1, 14, 33]. Studies have also highlighted novices' ineffective verification strategies when working with AI generated code [14, 28, 29]. Kazemitabaar et al. showed that CS1 students only ran the AI generated code 60% of the time, and 13% submitted AI-generated code without executing the code at all [14]. With an emphasis on non-technical end users, Zamfirescu-Pereira et al. discussed their struggles with AI prompting [41]. The authors found a lack of effective prompting strategies among end users, including limited examples or relevant details in their prompts [41]. The study also discussed users' tendency to over-generalize the AI tool's behavior, especially when verifying the tool's handling of a particular input (e.g., lack of extensive testing before making a judgment) [41]. Though the participants in our study have more programming experience, the results presented by Zamfirescu-Pereira et al. provide an example of how lay users prompt and verify AI systems they work with.

Studies have aimed to explain why novices struggle when programming with AI. Lucchetti et al. attributed novices' ineffective prompting to a lack of effective vocabulary and relevant context for the AI to generate code [21]. Prather et al. and Tankelevitch et al. argued that some of the challenges are metacognitive. For example, novice students struggle to verify the correctness of output, leading them down an incorrect path that may only compound their existing struggles [29].

In short, the studies above focused on how novices use and struggle with first-generation AI tools that involve more programmer-code interactions than vibe coding. In response, our study takes the first steps in understanding how novice programmers prompt and verify the output of a vibe coding platform that, by design, lowers the interactions between programmers and code.

3 Methods

3.1 Research Questions

The research questions for our study are as follows:

RQ1 How do students interact with AI tools when engaging in a vibe coding workflow?

RQ2 How does prior programming experience influence the way students develop software in a vibe coding workflow?

3.2 Study Context

3.2.1 Recruitment. Our research study recruited students from two computer science classes at a large research-centric North American institution, per our approved Human Subjects research protocol. The first course is a CS1 course that primarily enrolls first- and second-year students who have some prior programming experience before the course. The 10-week course is taught in the Java programming language and covers topics such as variables, conditionals, arrays, recursion, and basic object-oriented programming. This CS1 class requires students to implement test cases and pass hidden tests for programming assignments. The second course is an advanced Software Engineering (SWE) course that focuses on how to comprehend, manage, and contribute to legacy code bases. Students who are enrolled in this course are primarily students in their final year of their degree and have already completed the core software engineering class in which they learn about AGILE development and create a mobile or web application from scratch in a team.

3.2.2 Replit. We required students to use the Replit platform to create a web application. Replit, a browser-based, AI-integrated IDE, is designed specifically to support a vibe coding workflow [30]. It offers users the ability to *"Prompt your app ideas to life — no coding required"* — a quote taken directly from the Replit website[1]. The Replit interface features an AI-powered chat panel, where users interact with Replit's chatbots, and a live preview of the application's user interface. Initially, this preview is static, but it eventually becomes interactive and reflects current state of the underlying application (*"interative prototype"*). Notably, Replit does not display code by default, unlike traditional IDEs such as VS Code or Eclipse. Replit centers the development experience around chatting with AI and interacting with the prototype, rather than source code.

Within this interface, Replit offers two distinct chatbot tools, "Replit Agent" and "Replit Assistant". The Agent is optimized for building applications from the ground up; it can autonomously scaffold full-stack projects, configure environments, and directly create or modify files based on high-level prompts. In contrast, the Assistant is tailored for working within an existing project. It supports code-level refinement tasks such as debugging, explaining code behavior, refactoring, and minor feature edits. It functions similarly to an in-line coding collaborator: users can ask questions or request specific changes, and the Assistant proposes contextual edits that must be explicitly approved before being applied.

[1]https://replit.com/usecases/ai-app-builder

3.3 Participants

We recruited 9 students from the CS1 course and 10 students from the SWE course, as defined in Section 3.2.1.

Within the CS1 cohort (S11-S19), 4 students are pursuing computing or closely related majors and 5 are not. 6 identified as men and 3 as women. In terms of racial demographics, 7 identified as Asian or Asian American and 2 as White or Caucasian. Students self-declared their experience levels in programming, with 6 participants reporting confidence in building a small class project and 3 reporting only basic programming skills. All 9 participants reported having no software engineering internship experiences that involve working with legacy code bases.

Unlike our introductory cohort, all 10 students from the SWE course (S1-S10) have declared majors in computing. 8 identified as men and 2 as women. The racial demographics consisted of 7 Asian or Asian Americans, 2 Chicanx or Latinx, and 1 White or Caucasian. Students self-reported being capable of implementing projects of varying levels, with 4 comfortable building small projects, 4 developing intermediate projects, and 2 working on large projects and contributing to large codebases. Most of the participants (6 out of 10) reported internship experience, and 9 out of 10 aspire to become professional software engineers.

3.4 Study Procedure

The study sessions were conducted one-on-one on Zoom by the first author and a student. To refine task clarity and assess the study procedure, we first piloted it with 6 students from the same institution, where 2 students were from the same CS1 course, and 4 students were third and fourth year undergraduate students with a similar background as the SWE students. The finalized study procedure takes 2 hours and breaks down as follows:

Pre-task training (30 minutes): The student first watched a 6-minute tutorial recorded by an author, which demonstrates using Replit Agent and Assistant, accessing the codebase, and interacting with the Replit prototype. Next, the researcher introduced the think-aloud method through a live demo. Students then practiced using Replit while thinking aloud with a sample task. This training phase was not recorded.

Development session (60 minutes, timed): We designed an open-ended task to develop a personal budget management web application using Replit in students' personal browsers. Specifically, in our task description, we mentioned that the web application should be functioning and allow a user to do the following: 1) set a general monthly budget; 2) break that budget down into categories (e.g., food, rent, entertainment); 3) input their actual expenses as the month progresses; and 4) track how their spending compares to their goals. The task description was intentionally open-ended to encourage a diverse range of student interpretations and resulting application designs. Students were free to engage with the requirements as they saw fit and could make use of any features within the Replit environment, including the AI agent, assistant, code editor, console, and preview window. Additionally, students can use any external tools like ChatGPT as support.

The timer starts as a student opens and reads the task description document. While completing the task, the student followed

Table 1: Labeling scheme based on student-AI interactions, divided into four major categories.

Label	Definition
Interacting with Prototype	
Refresh Prototype	Reloading the tab displaying the prototype
Test Common Case	Interacting with the prototype using typical inputs or actions that reflect normal usage; repeated clicks or filling out a form count as a single case
Test Edge Case	Interacting with the prototype using unusual, invalid, or boundary inputs such as negative numbers and empty fields
Writing a Prompt	
Debug	Prompting to address an error, bug, or system failure
Add/Remove/Update Core Feature	Prompting to request any changes to a core feature (e.g. adding, checking, deleting, or editing budgets, budget categories, or expenses) that does not have an error or bug
Add/Remove/Update Non-core Feature	Prompting to request any changes to a non-core feature (e.g. UI elements, spending graphs) that does not have an error or bug
Ask Clarification Questions	Prompting to ask the AI to explain, define, or elaborate on a concept, term, feature, or technical details
Brainstorm Ideas	Prompting for open-ended ideas or approaches without a clearly defined solution
Other	Prompting actions not covered by other labels, such as simulating a test case or responding to the AI's questions
Managing Replit Workflow	
Accept Code Change	Accepting a code modification proposed by the Replit Assistant using the built-in apply-suggestion feature
Pause AI During Generation	Interrupting the AI's response before completion by clicking a pause or stop button in the Replit interface
Resume AI Generation After Pause	Resuming a paused AI response by clicking a continue or resume button in the Replit interface
Revert to Checkpoint	Reverting the code to a previous version by selecting a checkpoint created during an earlier AI response
Load Preview from Checkpoint	Loading a preview of code from a previous checkpoint without reverting to it
Approve Plan with X Additional Features	Approving 0 or more optional features from the Replit Agent's initial project plan
Engaging with Code/log	
Interpret	Spending significant time analyzing code, console logs, or other development-related outputs.
Edit	Modifying code, console logs, or other development-related outputs

the think-aloud protocols to verbalize thoughts, decisions, and intentions. The recording captured the student's voice and screen actions, and optionally, the student could also turn on the camera. Throughout the session, the researcher observed student behaviors and took notes to capture key observations.

Post-task interview (30 minutes): After the development session, the student immediately participated in a recorded semi-structured interview, where they recapped and reflected on their interactions with AI tools, reasoning behind their actions, and emotional responses during the session.

3.5 Data Analysis

After transcribing all recordings, two authors of this paper conducted a qualitative thematic analysis to identify and label the actions students performed during the development session. The two authors first jointly open-coded one session from a CS1 student and one session from an SWE student. Then, the authors discussed to form an initial set of labels, and each author independently applied these labels to the remaining 17 videos. The authors engaged in periodic discussions to refine the labeling scheme and resolve discrepancies, revisiting earlier videos as needed. Finally, the two authors met synchronously to reach a negotiated agreement [9, 32] on all applied labels, resolving any remaining differences and finalizing a consistent labeling scheme shown in Table 1.

To directly address our research questions, we labeled observable, screen-recorded student actions when interacting with the Replit IDE and when optionally interacting with an external Generative AI tool, ChatGPT. In addition to the labeling scheme, we recorded the prompts, code files, error messages following an interaction along with other information that provides context to the labels as "artifacts" to provide further context. Although the focus of the study is on the student behaviors themselves, we also refer to students' think-aloud utterances, and responses during the semi-structured interviews.

In addition to the labeling scheme (Table 1), we also recorded which AI tool the student used for *Writing a Prompt* (Replit Agent, Replit Assistant, ChatGPT). We also created sublabels to analyze prompts under the *Writing a Prompt — Debug* label (Table 2).

4 Results

4.1 RQ1: Student-AI Interactions in Vibe Coding

Overview. The overall interactions with AI tools across the 19 participants were shown in Figure 1. Across all students, the most prevalent label was *Interacting with Prototype* (63.61% of all labels, n

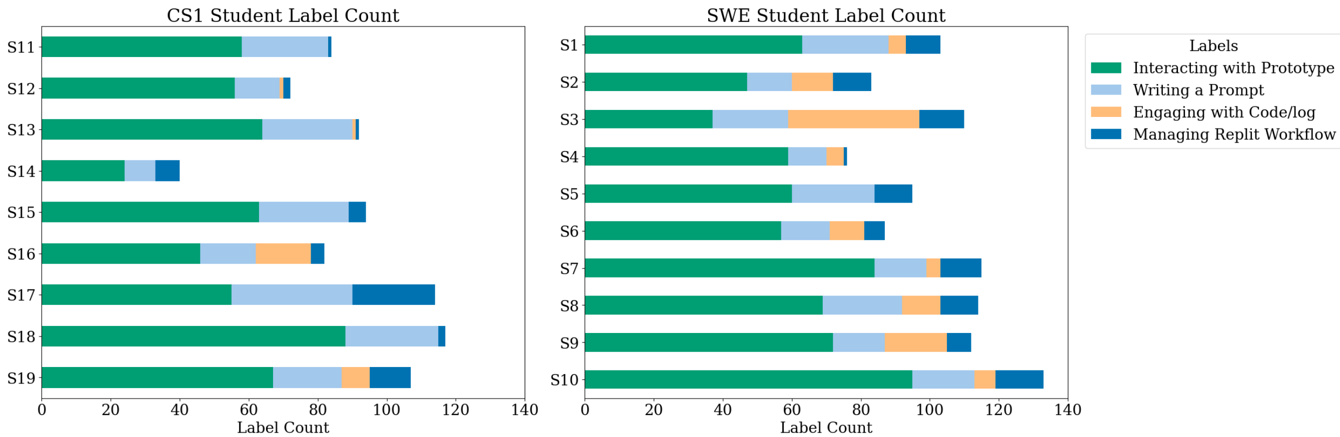

Figure 1: Distribution of labels across all 19 students, grouped by course. Each bar represents a single student, segmented by interaction categories: *Interacting with Prototype, Writing a Prompt, Engaging with Code/log,* and *Managing Replit Workflow.* S1-S10 are SWE students and S11-S19 are CS1 students.

Table 2: *Writing a Prompt — Debug* Sublabeling Scheme.

Sublabel	Definition
Error Message	Including a partial or full system error message when describing a bug or issue
Failing Case	Describing the input, condition, or steps that led to a failure or error, including what preceded the problem
Code-focused	Referencing specific files, code snippets, or implementation details believed to be related to the issue
Other Details	Providing other relevant information not captured above, such as triggering AI-generated debugging prompts (e.g., *"Troubleshoot this issue"* or " *"Ask Agent to explain code"*)
Low Context	Debugging prompts with none of the above checkboxes selected, showing little or no new information or direction when initiating or continuing a debugging effort

= 1164), followed by *Writing a Prompt* (20.60%, n = 377), *Managing Replit Workflow* (8.42%, n = 154), and *Engaging with Code/log* (7.38%, n = 135). However, differences emerged across cohorts: students in CS1 showed markedly higher proportions of *Writing a Prompt* labels compared to SWE students, whereas *Engaging with Code/log* interactions were more common among SWE students.

Restarters. While prompting was a common behavior across participants, a subset of students (4 out of 19) exhibited a more dramatic interaction pattern: they restarted the entire project using the Replit Agent mid-task. We term these students *restarters* (S1, S2, S14, S17). 3 out of the 4 *restarters* restarted primarily to simplify their interaction with the AI, citing overwhelming or ambiguous behavior from earlier prompts. For instance, one student reflected that they *"asked the Replit to do way too many things,"* making it difficult to identify specific issues. Another opted to *"break it down one task at a time"* after experiencing repeated failures. These behaviors suggest that some students use restart strategies not out of failure alone, but as a form of iterative refinement and task decomposition. The fourth *restarter* (S2) retained their original prompt but added a sentence to explicitly request a Flask framework over React due to familiarity and perceived simplicity, and they considered their restart as a means of architectural realignment rather than functional simplification. In all cases, the restart decision reflected metacognitive awareness about the limitations of debugging through prompting: students had attempted to fix issues in the original prototype but found the bugs unresolvable via continued AI interaction, prompting a fresh start instead.

Table 3: Frequency of labels for *Interacting with Prototype* actions.

Label	Count	Percent
Test Common Case	1067	91.67%
Refresh Prototype	71	6.10%
Test Edge Case	26	2.23%
Total	1164	100%

Interacting with Prototype. First, we examine how students interact with the prototypes generated by the AI tools (Table 3). The overwhelming majority of these interactions (91.67%) involved testing common use cases, while only 6.10% involved refreshing the prototype and a mere 2.23% involved edge case testing. Transcript data suggests that prototype refreshes were typically prompted by technical limitations (e.g., Replit failing to maintain state across pages) rather than as part of a deliberate debugging strategy. Notably, no student wrote or executed unit tests during the study, indicating that their approach to testing was exclusively centered on feature-level behaviors visible through the UI. The absence of structured test practices, combined with the minimal edge case coverage, suggests that many students were unable to progress beyond basic functionality, frequently encountering bugs that prevented more in-depth testing. These patterns highlight the inherently iterative and sometimes unstable nature of vibe coding, where students often remain occupied with basic interactions and repeated troubleshooting, rather than advancing toward comprehensive feature validation.

Table 4: Frequency of labels for *Writing a Prompt* actions.

Label	Count	Percent
Debug	230	61.01%
Add/Remove/Update Non-core Feature	63	16.71%
Add/Remove/Update Core Feature	53	14.06%
Other	16	4.24%
Ask Clarification Question	12	3.18%
Brainstorm Ideas	3	0.80%
Total	377	100%

Writing a Prompt. To better understand what students' goals are when they are prompting, we analyzed the labels of all prompting behaviors for all 19 students (Table 4). The majority of prompts (61.01%) were used for debugging AI-generated code, followed by modifications to non-core features (16.71%) and core features (14.06%). Prompts related to brainstorming, clarification questions, or miscellaneous tasks were relatively rare (<5% each). These data indicate that students primarily engaged with AI tools to troubleshoot and refine partial implementations, rather than to build functionality from scratch. As one student described their strategy, *"...finding [bugs] myself, realizing what the problem was, and then putting it back into the Agent to solve it in like 2 seconds..."* reflects how students often used AI to efficiently resolve implementation issues they had already identified.

Table 5: Frequency of student prompts with different AI tools.

AI Tool	Count	Percent
Replit Assistant	271	50.94%
Replit Agent	246	46.24%
ChatGPT	15	2.82%
Total	532	100%

In terms of which AI tools students used for prompting (Table 5), the Replit Assistant accounted for a slight majority of prompting interactions (50.94%), closely followed by Replit Agent (46.24%). ChatGPT was used only in 2.82% of the prompt instances, likely reflecting its auxiliary role in the workflow. These numbers suggest that while multiple AI tools were available, most of the prompting occurred within the embedded Replit interfaces, suggesting that accessibility and immediacy of tooling may play a significant role in shaping student behavior. As one student reflected, *"I didn't really understand the real difference between Agent and Assistant... it kind of felt like they were the same thing"*, emphasizing how conceptual ambiguity may have contributed to balanced usage. Another student explained, *"I'll kind of use ChatGPT and Copilot for assistance on making small fixes,"* illustrating ChatGPT's more occasional and supporting role relative to the Replit-native tools.

Table 6: Frequency of labels for *Managing Replit Workflow* actions.

Label	Count	Percent
Accept Code Change	105	68.18%
Approve Plan with X Additional Features	23	14.94%
Pause AI During Generation	12	7.79%
Revert to Checkpoint	7	4.55%
Load Preview from Checkpoint	5	3.25%
Resume AI Generation After Pause	2	1.30%
Total	154	100%

Managing Replit Workflow. In addition to prompting actions, students interacted with Replit in several other ways that reflect the vibe coding workflow with the Replit interface (Table 6). The most common action in this category was accepting Replit-proposed code changes (68.18%), a manual decision-making step mandatory for students working with Replit Assistant. After writing an initial prompt, a student had to approve an AI-generated implementation plan with optional additional features recommended by the Replit Agent (14.94%) based on the content of their first prompt. Some students also paused (7.79%) and resumed (1.30%) the AI generation process, indicating moments of review or reconsideration during code generation. In addition, a few students used Replit's version control features to either *Revert to Checkpoint* (4.55%) or *Load Preview from Checkpoint* (3.25%), actions functionally similar to Git's `revert` and `checkout`, respectively. These behaviors suggest that students occasionally found it necessary to backtrack after a prompt yielded undesirable or destabilizing changes, emphasizing the trial-and-error nature of vibe coding.

Engaging with Code/log. Among the rare cases where students engaged with code and logs generated by Replit, an overwhelming 90.37% of the interactions were reading and interpreting code (n = 122), and the remaining actions were direct edits (9.63%, n = 13). This strong preference for interpretation over modification suggests that students were often hesitant to alter AI-generated code, likely due to limited familiarity with the underlying implementation. The vibe coding workflow may exacerbate this hesitation by distancing students from the logic and structure of AI-generated codebases. As one student explained, *"Because so much of it was just done by the LLM, I had a lesser understanding of the codebase — rather than what I would do on my own, where I know what each line does."* We further expand and compare *Engaging with Code/log* behaviors in Section 4.2.

4.2 RQ2: Relations between Programming Experience and Vibe Coding Strategies

Engaging with Code/log. To investigate how programming experience influences students' engagement with code, we compared the proportion of *Engaging with Code/log* actions (either interpreting or editting AI-generated code) between students in CS1 and SWE. As shown in Figure 2, SWE students exhibited a higher overall proportion of *Engaging with Code/log* behaviors (M = 0.35%, SD = 0.33%) compared to CS1 students (M = 0.09%, SD = 0.19%), although the difference approaches but does not reach statistical significance (p = .0548). Among individual behaviors, 9 of 10 SWE students engaged

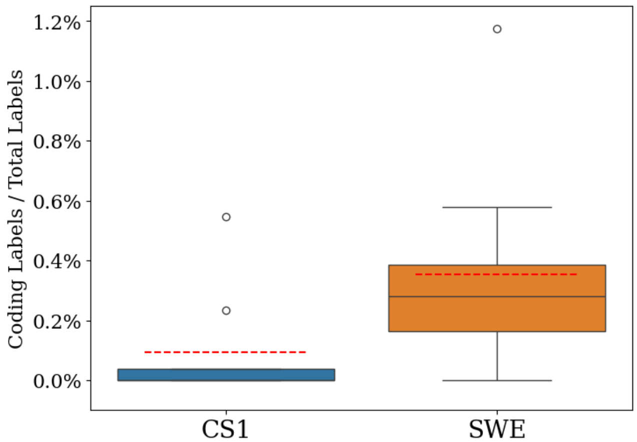

Figure 2: Proportion of *Engaging with Code/log* labels per student, normalized by each student's total behavior labels.

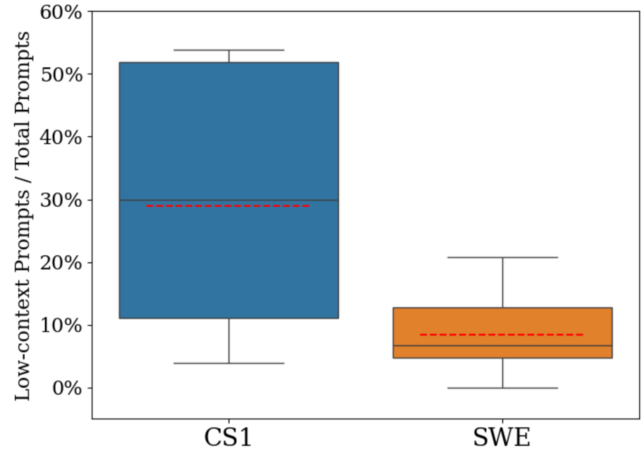

Figure 3: Proportion of *Low Context* prompts per student, normalized by each student's total number of prompts.

in code interpretation, and 3 of them also edited AI-generated code. By contrast, 4 of 9 CS1 students interpreted code, and only 2 CS1 students (S16 and S19) edited code. Notably, these two students were the only ones in their cohort who self-identified as capable of developing intermediate-level programs and debugging, suggesting that students with greater programming experience are more likely to engage directly with the structure and content of AI-generated code.

Writing a Prompt. Students from CS1 and SWE also showed noticeably different behaviors in two aspects when articulating their prompts during vibe coding, characterized as *Low Context* and *Code-focused* (defined in Table 2). *Low Context* prompts are often vague and offer limited actionable context, such as *"The buttons appear but they don't work at all,"* or follow-up phrases like *"it's still not working."* Figure 3 compares the proportion of low context prompts across the two groups and reveals that CS1 students had a significantly higher proportion of such prompts (M = 28.89%, SD = 20.66%) than SWE students (M = 8.39%, SD = 6.99%, $p < .01$).

In contrast, *Code-focused* prompts that reference specific segments of AI-generated code or logs were more prevalent among SWE students. For example, one student wrote, *"In line 118 of routes.ts, are you passing the correct data into storage.createStorage?"*. Another student wrote, *"In the <Header> component, error occurred 'cannot read properties of undefined (reading 'month')', set the month to be the current month at the time the user is using the web app."* These kinds of prompt demonstrate precise understanding and reasoning about the program's structure and behavior. 7 out of 10 SWE students used code-focused prompts, while only 1 student from the CS1 group (S16) did so. This difference suggests that students with greater programming experience were more capable of constructing prompts that referenced concrete implementation details, allowing them to more effectively communicate intent and troubleshoot with the AI system.

Interacting with Prototype. We also analyzed the extent to which students conducted edge case testing while *Interacting with Prototype* using uncommon or boundary inputs. 6 of 10 SWE students performed at least one edge case test, compared to 4 of 9 in CS1. While this difference is modest, it may reflect task progression: students who encountered repeated errors in basic functionality often focused on common case testing and did not reach a point where edge cases were necessary. Since edge case testing is introduced early in many computing curricula, including this CS1 course, the observed difference is less likely due to instruction and more likely tied to whether the student's implementation had stabilized enough to allow for more nuanced evaluation.

5 Discussion

Our study reveals how vibe coding — delegating coding, debugging, and design to generative AI via high-level prompts — reshapes student software development workflows. Below, we interpret key patterns, address implications for education/AI design, and acknowledge limitations.

5.1 Implications for Educators

With the recent arrival of agentic AI and vibe coding, there is considerable discourse around vibe coding as a viable practice for creating software. The fact that minimal supervision is part of vibe coding may suggest that someone completely unfamiliar with programming could just dive right into the practice. Our findings raise caution here, in that students from both the introductory and the advanced programming classes used the same set of skills when interacting with AI: prompting to debug, performing feature-level tests, and engaging with code. Although students interacted with code infrequently, the advanced programming students were still willing to do so in a way a complete novice (or even introductory programming students) might be unable to do.

The many examples of professional software engineers vibe coding, apparently with ease [32], may be misleading to non-experts as experts can judge whether and when to intervene. Even for experts, the conversation remains around creating "weekend projects" and not large software products that require maintenance. For students, who lack this safety net of expertise, vibe coding can obscure code comprehension, encourage reliance on feature-level testing over more structured validation methods, and hinder long-term skill development if not scaffolded appropriately. The utility of vibe coding as a pedagogical tool or professional practice thus remains an open question for the community. Our work offers early insight into how programming students (both introductory and advanced) navigate this space and where support may be most needed.

Interacting with the prototype was the most common behavior for both introductory and advanced students. Both groups experimented with the prototype to identify what worked and what didn't, then used that information to ask Replit to correct shortcomings. The fact that students were so quick to move to testing is an encouraging finding for teaching testing in computing courses. It's unclear why students were so eager to test, but it can potentially be attributed to 1) the interface making the prototype more accessible than code, and 2) students are equipped with enough technical intuition and interest to engage with a website. While both experts and non-experts performed frequent feature-based testing, experts are far more apt to perform edge-case tests. Teaching proper testing techniques with common cases and edge cases remains an important topic in computing education. Furthermore, since Replit encouraged feature-based testing over unit tests, the future of teaching unit-test-driven development in the age of AI remains uncertain.

Our findings demonstrate that introductory and advanced programming students have different prompting styles. The advanced students are far more apt to communicate with Replit using prompts that contain computational ideas and details than introductory students. These sophisticated prompts suggest, perhaps unsurprisingly, that students will need to be taught these computational ideas to communicate with Replit effectively. In particular, prompting in vibe coding is not simply about phrasing requests but about conveying intent grounded in code structure, logic, and context. This means instruction should go beyond syntax and include how to analyze generated output, translate errors into specific follow-up questions, and iteratively refine vague or failed prompts. Indeed, this finding supports the work of Lucchetti et al. who found that programming novices often author prompts that lack important details for the AI [21]. Overall, our work provides further evidence that students will need to learn computational skills and prompting skills to successfully interact with these models.

5.2 Limitations

While our study offers timely insights into how students interact with AI tools in a vibe coding workflow, several limitations must be acknowledged.

Sample representation. Our study was conducted at a single North American institution with students recruited from two computing courses, and participation was voluntary. This recruitment strategy may introduce self-selection bias, as students who opted into the study may be more confident, motivated, or interested in AI tools than their peers. Consequently, the behaviors observed may reflect a more engaged or higher-performing subset of the student population. Expanding future studies to multiple institutions and course settings would strengthen the applicability of findings across contexts.

Platform specificity. Our findings are grounded entirely in student experiences with the Replit platform, which is designed around an AI chat interface and an interactive prototype that deprioritizes code visibility. While this design makes Replit ideal for examining vibe coding, it does not reflect the full spectrum of vibe coding platforms. Other tools (e.g., Cursor, Copilot Agents) may provide greater transparency into generated code, tighter integration with traditional IDE workflows, or different prompting paradigms altogether. As such, our results should not be generalized to *all* vibe coding environments without caution. Future work could incorporate multiple platforms to assess how specific design features shape student behavior.

Focus on observed behaviors, not underlying reasoning. Our analysis centers on students' observable actions (e.g., prompting, testing, engaging with code) and classifies them using a structured labeling scheme. However, we did not systematically analyze students' spoken reasoning or reflective interview data to explain *why* those behaviors occurred. While the think-aloud protocol and post-task interviews provide valuable context, we primarily used them to validate labeled behaviors rather than drive deeper thematic or cognitive analysis. As a result, we cannot fully characterize students' intentions, mental models, metacognitive strategies, or emotional responses — essential elements for understanding the learning and decision-making processes behind their interactions with AI tools. A complementary analysis of verbal data could provide richer insight into students' goals, frustrations, and strategies while vibe coding.

Interaction frequency is not equivalent to time on task. All quantitative results in this paper are based on counts of interaction labels (e.g., number of prompts, code edits, test cases). However, raw frequency does not capture *how long* students spent on different activities. For instance, a single instance of code interpretation may involve several minutes of analysis, while five quick prompts could span less than a minute in total. As such, interpreting our frequency data as proportional effort or cognitive investment would be misleading. Future work should incorporate duration-based metrics or temporal coding to more accurately represent time allocation and sustained engagement with specific activities.

6 Conclusion

This study offers first insights into how computing students interact with a vibe-coding tool (Replit) while creating a functional and useful web app. Through qualitatively analyzing observations of how students vibe code, we uncovered students' interaction patterns with Replit and the differences between students in introductory and advanced computer science courses. Our findings include 1) the bulk of student interactions with Replit are testing and debugging, and 2) advanced students are more capable of prompting with computational sophistication and are more likely to interact with the codebase. Our findings offer insights to educators on the skills that students use when vibe coding while buttressing findings in GenAI broadly — that testing and debugging remain critical in the AI era [26, 27] and that students need training in prompting [21].

7 Acknowledgments

This work is supported by the National Science Foundation Graduate Research Fellowship Program and NSF Award #2417374, and by the Google Award for Inclusion Research Program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors(s) and do not necessarily reflect the views of the National Science Foundation or Google.

References

[1] 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*. Article 13. https://doi.org/10.1145/3699538.3699567

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

[3] Xiang Chen, Chaoyang Gao, Chunyang Chen, Guangbei Zhang, and Yong Liu. 2025. An Empirical Study on Challenges for LLM Application Developers. *ACM Trans. Softw. Eng. Methodol.* (2025). https://doi.org/10.1145/3715007

[4] Cursor. 2025. Cursor - The AI Code Editor. https://cursor.com/en. Accessed: 2025-07-25.

[5] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with copilot: Exploring prompt engineering for solving CS1 problems using natural language. In *Proceedings of the 54th ACM technical symposium on computer science education V. 1*. 1136–1142.

[6] Paul Denny, Stephen MacNeil, Jaromir Savelka, Leo Porter, and Andrew Luxton-Reilly. 2024. Desirable characteristics for AI teaching assistants in programming education. In *Proceedings of the 2024 conference on Innovation and Technology in Computer Science Education V. 1*. 408–414.

[7] Paul Denny, James Prather, Brett A Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N Reeves, Eddie Antonio Santos, and Sami Sarsa. 2024. Computing education in the era of generative AI. *Commun. ACM* 67, 2 (2024), 56–67.

[8] Ya Gao. 2024. Research: Quantifying GitHub Copilot's impact in the enterprise with Accenture. https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-in-the-enterprise-with-accenture/

[9] D.R. Garrison, M. Cleveland-Innes, Marguerite Koole, and James Kappelman. 2006. Revisiting methodological issues in transcript analysis: Negotiated coding and reliability. *The Internet and Higher Education* 9, 1 (2006), 1–8. https://doi.org/10.1016/j.iheduc.2005.11.001

[10] Irene Hou, Owen Man, Kate Hamilton, Srishty Muthusekaran, Jeffin Johnykutty, Leili Zadeh, and Stephen MacNeil. 2025. 'All Roads Lead to ChatGPT': How Generative AI is Eroding Social Interactions and Student Learning Communities. In *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. 79–85. https://doi.org/10.1145/3724363.3729024

[11] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2025. From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future. arXiv:2408.02479 [cs.SE] https://arxiv.org/abs/2408.02479

[12] Andrej Karpathy. 2025. https://x.com/karpathy/status/1886192184808149383

[13] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI code generators on supporting novice learners in introductory programming. In *Proceedings of the 2023 CHI conference on human factors in computing systems*. 1–23.

[14] 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*. Article 3, 12 pages. https://doi.org/10.1145/3631802.3631806

[15] Majeed Kazemitabaar, Oliver Huang, Sangho Suh, Austin Henley, and Tovi Grossman. 2025. Exploring the design space of cognitive engagement techniques with AI-generated code for enhanced learning. In *Proceedings of the 30th International Conference on Intelligent User Interfaces*. 695–714.

[16] Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, Austin Henley, Paul Denny, Michelle Craig, and Tovi Grossman. 2024. CodeAid: Evaluating a classroom deployment of an LLM-based programming assistant that balances student and educator needs. In *Proceedings of the 2024 chi conference on human factors in computing systems*. 1–20.

[17] Sam Lau and Philip Guo. 2023. From" Ban it till we understand it" to" Resistance is futile": How university programming instructors plan to adapt as more students use AI code generation and explanation tools such as ChatGPT and GitHub Copilot. In *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1*. 106–121.

[18] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 2024. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Article 52, 13 pages. https://doi.org/10.1145/3597503.3608128

[19] 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*. 1257–1268. https://doi.org/10.1145/3531146.3533182

[20] Mark Liffiton, Brad E Sheese, Jaromir Savelka, and Paul Denny. 2023. CodeHelp: Using large language models with guardrails for scalable support in programming classes. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*. 1–11.

[21] Francesca Lucchetti, Zixuan Wu, Arjun Guha, Molly Q Feldman, and Carolyn Jane Anderson. 2024. Substance Beats Style: Why Beginning Students Fail to Code with LLMs. *arXiv preprint arXiv:2410.19792* (2024).

[22] Microsoft Copilot. 2025. Copilot and AI agents. https://www.microsoft.com/en-us/microsoft-copilot/copilot-101/copilot-ai-agents. Accessed: 2025-07-25.

[23] Ismael Villegas Molina, Audria Montalvo, Benjamin Ochoa, Paul Denny, and Leo Porter. 2024. Leveraging LLM tutoring systems for non-native english speakers in introductory CS courses. *arXiv preprint arXiv:2411.02725* (2024).

[24] 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

[25] 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*. 2785–2799. https://doi.org/10.1145/3576915.3623157

[26] Leo Porter and Daniel Zingaro. 2024. *Learn AI-assisted Python programming: with GitHub Copilot and ChatGPT*. Simon and Schuster.

[27] James Prather, Juho Leinonen, Natalie Kiesler, Jamie Gorson Benario, Sam Lau, Stephen MacNeil, Narges Norouzi, Simone Opel, Vee Pettit, Leo Porter, et al. 2025. Beyond the hype: A comprehensive review of current trends in generative AI research, teaching practices, and tools. *2024 Working Group Reports on Innovation and Technology in Computer Science Education* (2025), 300–338.

[28] 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 (2023), 31 pages. https://doi.org/10.1145/3617367

[29] 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*. 469–486. https://doi.org/10.1145/3632620.3671116

[30] replit. 2025. replit - Build apps and sites with AI. https://replit.com/. Accessed: 2025-07-25.

[31] Ranjan Sapkota, Konstantinos I. Roumeliotis, and Manoj Karkee. 2025. Vibe Coding vs. Agentic Coding: Fundamentals and Practical Implications of Agentic AI. (2025). https://arxiv.org/abs/2505.19443

[32] Advait Sarkar and Ian Drosos. 2025. Vibe coding: programming through conversation with artificial intelligence. (2025). https://arxiv.org/abs/2506.23253

[33] 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*. 1050–1056. https://doi.org/10.1145/3641554.3701800

[34] 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/

[35] Md Istiak Hossain Shihab, Christopher Hundhausen, Ahsun Tariq, Summit Haque, Yunhan Qiao, and Brian Mulanda. 2025. The Effects of GitHub Copilot on Computing Students' Programming Effectiveness, Efficiency, and Processes in Brownfield Programming Tasks. *arXiv preprint arXiv:2506.10051* (2025).

[36] Lev Tankelevitch, Viktor Kewenig, Auste Simkute, Ava Elizabeth Scott, Advait Sarkar, Abigail Sellen, and Sean Rintel. 2024. The metacognitive demands and opportunities of generative AI. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–24.

[37] 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 conference on Innovation and Technology in Computer Science Education v. 1*. 297–303.

[38] 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*. Article 332, 7 pages. https://doi.org/10.1145/3491101.3519665

[39] 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*. 1475–1493. https://doi.org/10.1145/3630106.3658984

[40] Bingyang Wei. 2024. Requirements are All You Need: From Requirements to Code with LLMs. In *2024 IEEE 32nd International Requirements Engineering Conference (RE)*. 416–422. https://doi.org/10.1109/RE59067.2024.00049

[41] J.D. Zamfirescu-Pereira, Richmond Y. Wong, Bjoern Hartmann, and Qian Yang. 2023. Why Johnny Can't Prompt: How Non-AI Experts Try (and Fail) to Design LLM Prompts. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. Article 437, 21 pages. https://doi.org/10.1145/3544548.3581388