Understanding and Enhancing CS Students' Interaction Experience with AI Coding Assistant Tools

XIAO LONG, State Key Laboratory of Complex & Critical Software Environment (CCSE), School of Computer Science and Engineering, Beihang University, China
XIN TAN*, CCSE, School of Computer Science and Engineering, Beihang University, China
YINGHAO ZHU, School of Artificial Intelligence, Beihang University, China
JING JIANG, CCSE, School of Computer Science and Engineering, Beihang University, China
LI ZHANG, CCSE, School of Computer Science and Engineering, Beihang University, China

AI coding assistants (ACATs) are reshaping computer science (CS) education, yet students' perception and responses to ACATs' suggestions remains limited understood, especially regarding behavioral patterns, decision-making, and usability challenges. To address this gap, we conducted a study with 27 CS students, examining their interactions with three widely used ACATs across five key dimensions: interaction frequency and acceptance rate, self-perceived productivity, behavioral patterns, decision-making factors, and challenges and expectations. To support this investigation, we developed an experimental platform incorporating a VSCode extension for log data collection, screen recording and automatic generation of personalized interview and survey questions. Our findings reveal substantial variation in ACAT acceptance rates depending on task types, recommendation methods, and content. We propose a novel five-layer interaction behavior model that captures different stages of user interaction. Notable insights include the problem-solving value of rejected AI suggestions, the inefficiencies introduced by modifying existing code that often lead to backtracking, and the high stability of "slowly accepted" suggestions. Moreover, we identify 22 decision-making factors, 11 challenges, and 23 student expectations for future ACAT improvements—such as enhanced debugging accuracy and adaptive learning of individual coding styles. This study contributes actionable design implications for improving ACAT usability, informing student interaction strategies, and guiding future research in human-software interaction, ultimately aiming to better support CS education.

CCS Concepts: • **Software and its engineering → Software notations and tools**; • **Human-centered computing → Human computer interaction (HCI)**; **Empirical studies in HCI**; **User studies**; • **Social and professional topics → Computer science education**.

Additional Key Words and Phrases: AI, Artificial Intelligence, AI coding assistant tools, large language models, LLM, Copilot, novice programming, introductory programming, human-software interactions

*Corresponding Author

Authors' Contact Information: Xiao Long, longxiao@buaa.edu.cn, State Key Laboratory of Complex & Critical Software Environment (CCSE), School of Computer Science and Engineering, Beihang University, Haidian District, Beijing, China; Xin Tan, xintan@buaa.edu.cn, CCSE, School of Computer Science and Engineering, Beihang University, Haidian District, Beijing, China; Yinghao Zhu, zhuyinghao@buaa.edu.cn, School of Artificial Intelligence, Beihang University, Haidian District, Beijing, China; Jing Jiang, jiangjing@buaa.edu.cn, CCSE, School of Computer Science and Engineering, Beihang University, Haidian District, Beijing, China; Li Zhang, lily@buaa.edu.cn, CCSE, School of Computer Science and Engineering, Beihang University, Haidian District, Beijing, China.

ACM 1557-735X/2018/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Xiao Long, Xin Tan, Yinghao Zhu, Jing Jiang, and Li Zhang. 2018. Understanding and Enhancing CS Students' Interaction Experience with AI Coding Assistant Tools. *J. ACM* 37, 4, Article 111 (August 2018), 45 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

AI-coding assistant tools (ACATs), such as *GitHub Copilot*, *Tabnine*, and *CodeGeex*, have become indispensable in modern software development. Leveraging large language models (LLMs) trained on vast amounts of code, e.g., substantial repositories on GitHub, these tools provide context-aware and personalized code recommendation, significantly enhancing developer productivity [10, 63, 64]. By January 2024, *GitHub Copilot* alone had surpassed 1.3 million paid subscribers and was adopted by more than 50,000 companies [39], underscoring its widespread impact [36]. Given this rapid adoption, researchers have sought to understand how ACATs affect software development practices. While extensive research has examined ACATs' effectiveness and interaction patterns among professional developers [4, 38, 48], studies on computer science (CS) students — another critical user group — have only recently emerged and remain less fine-grained, leaving their distinct learning needs and challenges underexplored [1].

Unlike professional developers, who primarily use ACATs to improve productivity, students rely on these tools for programming knowledge acquisition [30, 43]. Although ACATs can facilitate learning through code suggestions, concept explanations, and error resolution [15, 25], over-reliance or misuse can hinder the development of fundamental programming skills and problem-solving abilities [5, 45]. Moreover, students' limited expertise makes them more vulnerable to unproductive interactions, as they may struggle to critically evaluate tool-generated suggestions [6, 24, 44]. These challenges highlight the need for a deeper understanding of how students interact with ACATs and how these interactions can be optimized to support both learning and productivity.

The growing body of work on student-ACAT interactions has begun to shed light on important aspects such as high-level outcomes learning impacts and meta-cognition effects [20, 24, 35, 44, 46, 47, 53], as well as broad interaction patterns [1, 20, 24]. However, granular insights into fine-grained behavioral patterns, decision-making processes, and the factors influencing students' interactions remain underexplored. Such insights are crucial for developing evidence-based guidelines (such as when and how to employ specific prompt strategies) and informing tool design improvements to better support students' learning and productivity.

In this paper, we address this gap by exploring student-ACAT interactions across five dimensions: interaction frequency and acceptance rate, self-perceived productivity, behavioral patterns, decision-making determinants, and students' challenges and expectations. Specifically, we propose the following research questions to guide our study.

— ***RQ1: What are the characteristics of student-ACAT interactions regarding usage frequency and acceptance rates across different scenarios?*** This question examines the frequency of tool usage in different scenarios (i.e., different task types, different tool recommendation methods, and different tool recommendation content), providing a comprehensive understanding of students' engagement with and adoption of ACATs.

— ***RQ2: How does interacting with ACATs affect students' self-perceived productivity?*** This question investigates the changes in students' perceptions of task difficulty, individual performance, and their frequency to seek external help before and after interacting with ACATs, aiming to exploring participants' programming experience and the emotional value they derive from using these tools.

— ***RQ3: What are the interaction patterns of students when using ACATs?*** This question explores the granular interaction patterns across different interaction modes and their corresponding frequency distributions, offering insights into how to optimize student interaction strategies.
— ***RQ4: What factors influence students' evaluations and decisions when interacting with ACATs?*** This question investigates the determinants of students' acceptance, rejection, or modification of code recommendations, revealing their needs and preferences.
— ***RQ5: What are the challenges and expectations when students interact with ACATs?*** This question identifies the challenges faced by students and their expectations, providing actionable insights for tool design and functionalities.

By addressing the above questions, our study aims to deepen the understanding of student-ACAT interactions, paving the way for more effective and supportive tools in CS education. To achieve this goal, we design a controlled human study involving 27 CS students, simulating real-world programming scenarios. Participants engage in three representative software development tasks (i.e., Algorithms and Data Structures, Management System Development, and Research Tool Development) with and without the assistance of three widely-used ACATs (i.e., *GitHub Copilot*, *Tabnine*, and *CodeGeeX*). To support this investigation, we develop an experimental platform equipped with a VSCode IDE extension for process data collection, alongside features for task description delivery, code evaluation, and dynamic generation of personalized interview and survey questions. Through a comprehensive analysis of participants' submitted code, behavioral data, and interview and survey results, we reveal several key findings. Overall, the novel contributions of this work are as the follows:

- **Multi-Perspective Evaluation of Student-ACAT Interactions:** We evaluate students' interactions with ACATs from multiple perspectives, including the acceptance rate of ACAT-recommended code across different task types, recommendation methods, and content categories, as well as the impact of ACAT usage on students' self-perceived productivity. This comprehensive evaluation provides a deep understanding of ACATs' usage dynamics and acceptance patterns from an interaction-centric perspective.
- **A Novel Five-Layer Interaction Behavior Model:** By analyzing students' fine-grained hierarchical interaction behaviors with ACATs during each recommendation cycle, we propose a five-layer interaction behavior model and compute state transition probabilities for each path within the model. By integrating interview data, we interpret the higher-level significance of these paths and identify notable interaction patterns.
- **Decision-Making Factors, Challenges, and Expectations:** Through timely and personalized post-experiment interviews reflecting participants' real-time interaction processes, we identify 22 factors influencing students' decisions to accept, reject, or modify ACAT-generated suggestions. We also uncover 11 challenges that hinder students' interaction experiences and 23 expectations for improving ACAT functionality and usability.
- **Practical Insights for Enhancing Student-ACAT Interactions:** Building on our findings, we provide actionable recommendations to enhance student-ACAT interactions from three perspectives: design improvements for ACATs, best practices for students to optimize tool usage, and implications for software engineering (SE) researchers.

The remainder of this paper is organized as follows. In Section 2, we present the related work. We introduce the methods that we use in Section 3. We report our results in Section 4. We discuss the implications in Section 5 and outline the threats to validity in Section 6. Finally, we conclude the paper in Section 7. More details about our experiment design, survey and interview templates, and collected data can be accessed in our replication package [33].

2 RELATED WORK

This section contextualizes our study within the existing literature, organized into three main streams: empirical evaluations of ACAT performance and perceptions in general contexts, the integration and impact of ACATs in programming education, and analyses of developers' interaction behaviors with these tools.

2.1 Performance, Impact, and Perceptions of ACATs

In this section, we review existing empirical studies that evaluate ACATs in general software development contexts. We focus on three primary dimensions: objective evaluations of ACATs' performance against human, ACATs' impact on development tasks, and subjective assessments of developer perceptions and requirements derived from surveys.

Some studies compare human programmers with ACATs [13, 40, 52]. Nascimento et al. [40] compare the performance of software engineers and *ChatGPT* on non-functional evaluation metrics, such as memory efficiency, which receive less attention. The results indicate that *ChatGPT* surpasses novice programmers in solving easy and medium-level problems, but lacks evidence to outperform experienced programmers. Dakhel et al. [13] compare *Copilot* with human programmers on solving fundamental algorithmic problems. They find that humans' solutions have a higher correct ratio, but *Copilot*'s buggy solutions are easier to fix. Siroš et al. [52] evaluate GitHub Copilot's code quality using a custom automated framework on the LeetCode problem set. They find that Copilot performs better in Java and C++ compared to Python3 and Rust, while generates more efficient code than an average human programmer. However, these studies treat ACATs as an alternative to programmers rather than as an assistant, which differs from real-world software development scenarios.

Other studies investigate how ACATs affect developers in completing certain programming tasks [55, 61]. Both studies find that the time difference in completing tasks brought by ACATs is not significant. Imai [22] conducts an experiment with 21 participants to compare the productivity and code quality of pair programming with *GitHub Copilot* versus human pair programming. They find that while programming with *Copilot* generates more lines of code, the code quality is lower. Asare et al. [2] conduct a study with 25 participants solving programming problems with and without GitHub Copilot assistance, focusing on tasks with potential security vulnerabilities. Their findings indicate that Copilot improves security in solutions for harder problems but has no significant effect on easier tasks. Kazemitabaar et al. [24] conducted a controlled experiment involving 69 learners of programming (aged 10-17) completing 45 Python code-authoring tasks. Their findings demonstrated that Codex usage significantly enhanced code-authoring performance, yielding a 1.15x increase in completion rate and 1.8x improvement in scores, without compromising performance on manual code-modification tasks. Lee et al. [27] found that while AI-powered code completion tools like GitHub Copilot can accelerate coding, they may also lead developers to choose more predictable identifiers, potentially influencing their programming decisions and reducing their sense of agency. However, these studies primarily emphasize outcome-based metrics (e.g., task completion time and success rate) while overlooking the in-depth analysis of interaction processes, particularly regarding interaction effectiveness.

Previous studies also collect users' challenges and expectations with ACATs based on surveys [11, 31, 56]. Ciniselli et al. [11] surveyed 80 developers on crucial characteristics of ACATs and constructed a taxonomy of 70 requirements that need to be considered when designing code recommendation systems. Wang et al. [56] surveyed 599 practitioners about their expectations on code completion and compared these with the existing research. They find that no paper evaluates the generated code via grammatical correctness and readability, which the practitioners value

most. Liang et al. [31] find that, among all the challenges developers encountering, intellectual property and ACATs' access to their code are the top concerns. However, while these surveys offer profound insights, they do not capture the user experience immediately after developers use the tools. This lack of immediate feedback limits our understanding of the genuine feelings and challenges developers encounter while interacting with ACATs in their day-to-day workflows.

2.2 ACATs in Programming and Software Engineering Education

The rise of ACATs has profound implications for both SE education and the broader discipline of foundational CS education, as programming proficiency is a cornerstone of both. Research in this area has largely focused on the impact of ACATs on the acquisition of core programming skills, which are essential prerequisites for advanced SE practices.

A primary line of inquiry has been evaluating the performance of code generation tools in introductory programming contexts [14, 16, 50]. In their early work, Finnie-Ansley et al. [16] examined Codex's performance on typical introductory programming problems. The model achieved an overall score of approximately 80% in both tests, placing it in the top quartile among students. Denny et al. [14] evaluated Copilot's performance on a public dataset of 166 programming problems, finding that it successfully solved approximately 50% on its first attempt and an additional 60% of the remainder through simple natural language reformulations.

Other research has focused on ACATs' capabilities in generating educational resources [29, 49, 58]. Sarsa et al. [49] employed Codex to synthesize novel programming exercises and code explanations suitable for introductory programming courses. Their findings revealed that while over 80% of generated exercises contained executable example solutions, only 30% of these solutions passed the test cases generated by Codex itself. Leinonen et al. [29] and Wermelinger et al. [58] analyzed LLMs' ability to interpret programming error messages, a notorious challenge for novice programmers. Both studies concluded that although LLMs could potentially enhance programming error messages in certain cases, neither Codex nor Copilot was sufficiently reliable for direct student use.

Furthermore, several researchers have investigated the use of LLMs for generating instructive code explanations [28, 34, 49]. Sarsa et al. [49] utilized Codex to generate natural language explanations for common code samples in introductory programming courses. Their results showed that while 90% of code segments received complete coverage, only 70% of individual lines were correctly explained. MacNeil et al. [34] integrated GPT-3-generated code explanations into an online textbook and found that students perceived these explanations as beneficial for learning. Leinonen et al. [28], comparing GPT-3 and student-generated code explanations, discovered that students found GPT-3's explanations more comprehensible and more accurate in summarizing the code.

Finally, several studies have explored the impact of ACATs on computer programming education [7, 20, 42]. Through surveys and interviews with students and faculty, Boguslawski et al. [7] investigated the impact of LLMs on motivation and learning in programming education, highlighting how LLMs enhance learner autonomy and competence while emphasizing the continued importance of social support for student motivation. Moreover, Park et al. [42] conducted a large-scale mixed-methods study comparing ChatGPT, Stack Overflow, and no-assistance approaches in programming learning, demonstrating ChatGPT's significant performance enhancement in big data analytics tasks and highlighting the potential of generative AI as a personalized learning support tool. These studies, while often set in CS contexts, provide crucial insights into how these tools are shaping the next generation of software engineers.

2.3 Developers' Interaction with ACATs

Several studies [1, 4, 23, 38, 44, 47, 48, 57, 65] have investigated the interaction patterns between developers and ACATs. Through observing 20 participants, Barke et al. [4] conducted the first grounded theory analysis of programmer-Copilot interactions, identifying two distinct interaction modes: "acceleration mode", where programmers use Copilot to expedite known solutions, and "exploration mode", where programmers leverage Copilot to explore potential approaches when uncertain.

Mozannar et al. [38] developed a Comprehensive Understanding of Programmer Status (CUPS) taxonomy and validated it through a study with 21 programmers. Their findings indicated that programmers spend 34.3% of their session time verifying and modifying Copilot suggestions, with over half of the total task duration devoted to Copilot-related activities. Wang et al. [57] conducted a comprehensive controlled experiment with 109 participants to evaluate the effectiveness of ChatGPT in software development tasks and observe the interactions between participants and ChatGPT. Their results revealed that while the AI assistant demonstrated proficiency in solving simple coding problems, its performance in supporting typical software development tasks was limited, and participants developed diverse interaction strategies when working with the generative AI tools. Ziegler et al. [65] discovered that code acceptance rate correlates more strongly with productivity than code persistence.

Prather et al. [44] conducted pioneering research on CS1 students using GitHub Copilot for introductory programming assignments. Their observations and interviews revealed novel interaction patterns termed "drifting" and "shepherding", while also highlighting students' cognitive and meta-cognitive challenges alongside their perceived benefits and limitations of the technology. Amoozadeh et al. [1] also examined student-ACAT collaboration patterns in CS1 courses, revealing that approximately one-third of students exclusively rely on generative AI tools like ChatGPT without initial independent problem-solving attempts, raising concerns about over-dependence and insufficient solution validation. Güner et al. [20] explored students' diverse interaction profiles with ChatGPT during programming learning, demonstrating how targeted instructional interventions can significantly influence AI tool usage and potentially reshape students' computational problem-solving strategies.

In summary, previous studies have analyzed student-ACAT interaction patterns at a coarse level, without fine-grained examination of students' behavioral patterns and their frequency distributions across interaction modalities. Additionally, limited attention has been paid to the factors that influence students' decision-making processes during ACAT interactions. Understanding these aspects is crucial for enhancing students' ACAT experience through both the identification of best interaction practices and the optimization of tool functionality, thereby potentially improving learning outcomes.

3 METHODOLOGY

Our research aims to (1) evaluate the frequency and success rate of student-ACAT interactions, (2) assess the students' self-perceived productivity, (3) conduct a fine-grained analysis of interaction behaviors, (4) explore factors influencing students' decision-making processes, and (5) identify challenges and expectations students encounter during these interactions. To achieve these objectives, we designed and conducted a controlled experiment, as illustrated in Fig. 1. We selected three widely-used ACATs and designed three typical types of tasks representative of SE education, each including two comparable sub-tasks (Task A and Task B). We recruited 27 CS students as participants. Each participant was assigned to a single experimental condition, consisting of one ACAT and one task type. This assignment resulted in a balanced distribution, with 9 participants

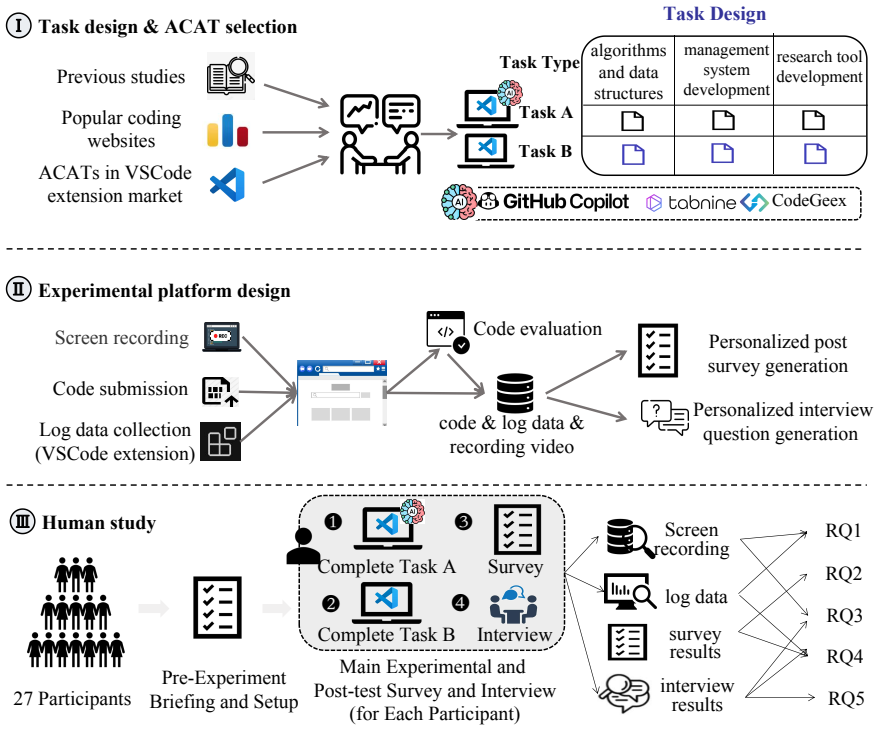


Fig. 1. Study Overview

for each ACAT and 9 for each task category, while each experimental condition (i.e., the specific ACAT-task combination) included 3 participants. Each participant was required to complete Task A with their assigned ACAT extension enabled and Task B with the ACAT disabled. In both conditions, participants could fully utilize VSCode's standard, syntax-based auto-completion and were permitted to use web browsers for external searches at any time. To support our study, we developed an experimental platform capable of collecting participants' interaction log data and screen recordings, evaluating code submissions, and automatically generating personalized interview questions based on participants' interaction processes. After completing the coding tasks, we conducted timely interviews and surveys with participants to gather additional qualitative feedback.

3.1 ACATs Selection

To ensure comprehensive analysis and data diversity, we selected three widely-used ACATs: *GitHub Copilot*, *Tabnine*, and *CodeGeeX*. These tools leverage advanced AI and machine learning techniques to provide functionalities such as code suggestions, auto-completion, code repair, and natural language to code conversion (NL2Code). The selection criteria were based on their widespread adoption in the developer community, comparable feature sets and user interfaces within VSCode extensions, and diverse pricing models (including both commercial and open-source options). Table 1 summarizes their key characteristics, including vendors, initial release dates, pricing strategies, and installation statistics as of March 2024.

The three ACATs selected for our study share a highly consistent set of core functionalities and user interfaces (UIs). As depicted in Fig. 2 (a)-(c), all tools implement code completion using inline ghost text (grayed-out suggestions). While multiple suggestions may be generated, only

Table 1. Basic Information of Studied ACATs

ACATs	Created-by	Release Time	Pricing	Downloads (–Mar. 2024)	Version Studied
GitHub Copilot	GitHub and OpenAI	2021	Paid	14M+	v1.144.0
Tabnine	Tabnine	2018	Paid	6.6M+	v3.51.0
CodeGeex	Zhipu AI	2022	Free	661K+	v2.2.6

one is displayed at a time, users navigate them sequentially via keyboard shortcuts. Acceptance mechanisms are also standardized: the "*Tab*" key accepts an entire suggestion by default, with options for partial, word-by-word acceptance. Beyond code completion, each tool offers a suite of auxiliary features through a sidebar chat interface (Figures 2 (d)-(f)), including command-driven actions (e.g., "*/debug*", "*/explain*") and code generation from natural language. Furthermore, it is worth noting that among the three tools, *CodeGeeX* natively supports a bilingual (Chinese and English) user interface. In contrast, *GitHub Copilot* and *Tabnine* rely on a VS Code Chinese language extension for interface localization. This allowed participants who preferred a localized interface to enable Chinese UI localization easily. Crucially, once the VS Code Chinese language extension was active, the user interface for the core code completion feature became virtually indistinguishable across all three tools.

This deliberate selection of tools with similar interaction paradigms serves an important methodological purpose. Our goal is not to compare ACATs, but to capture diverse, representative data on student interactions to derive generalizable insights. The consistent UI and core features help control for tool-specific differences, which mainly stem from their underlying models. This design allows us to focus on fundamental student–AI interaction phenomena across our research questions—feature use (RQ1), self-perceived productivity (RQ2), interaction patterns (RQ3), decision-making rationales (RQ4), and common challenges and expectations (RQ5)—ultimately revealing shared characteristics of how students engage with current ACATs.

3.2 Task Design

We designed three categories of Python programming tasks to reflect common coding scenarios in SE education, as detailed in Table 2. These tasks were carefully crafted to simulate authentic student software development activities while ensuring feasibility within the constraints of a controlled study. The details of these tasks, including complete task specifications, can be accessed in the replication package [33].

3.2.1 Task Categories. We design the following three task categories:

1) Algorithms and Data Structures (ADS): This type represents fundamental skills for CS and SE students, crucial in both university courses and job interviews. We choose two algorithm problems from Codeforces, a world-renowned online assessment system [12].

2) Management System Development (MSD): This category focuses on system-level software development tasks that encompass data persistence, input-driven data manipulation, formatted output generation, and error handling. Such tasks are fundamental components of object-oriented programming curricula and evaluate the SE skill of iteratively developing upon a pre-existing system, which represent common software development scenarios in industrial settings [8]. The tasks require participants to implement merchant and customer functionalities within a shop

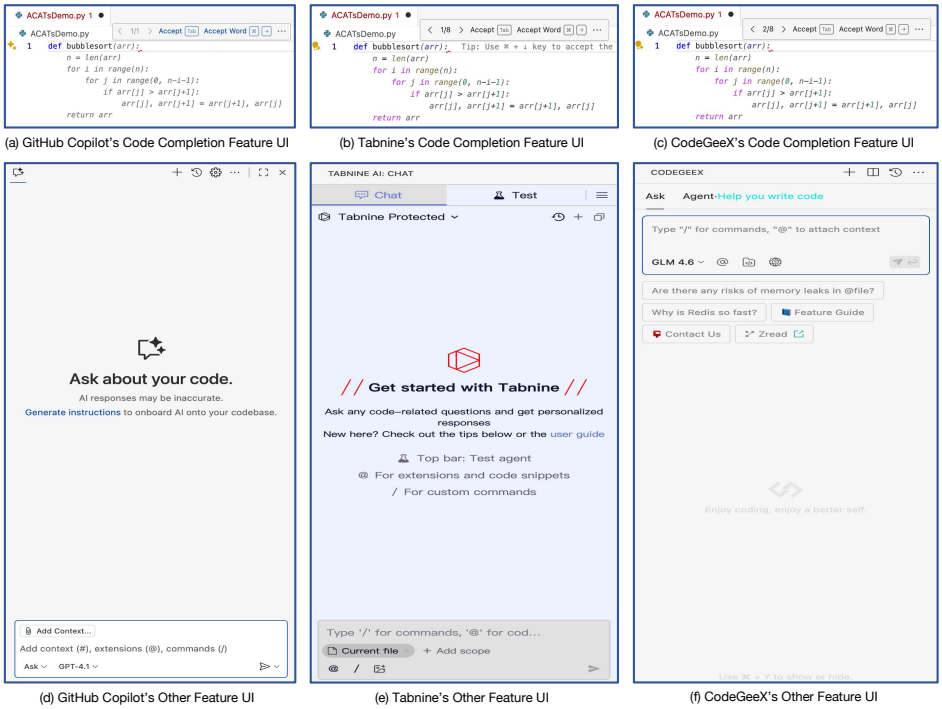


Fig. 2. Three ACATs' Code Completion Feature and Other Feature UI

Table 2. Three Types of Python Coding Tasks

Category	Task A (code with ACATs)	Task B (code without ACATs)
Algorithms and Data Structures (ADS)	**Lucky Number**: *judge if a number satisfies the given rule*	**Array Gracefulness**: *compute the best property of an array according to the given rule*
Management System Development (MSD)	**Merchant Class**: *implement merchant class of three relevant commands in a small product management system*	**Customer Class**: *implement customer class of three relevant commands in a small product management system*
Research Tool Development (RTD)	**Bounty Issue Statistics**: *analyze three characteristics of substantial bounty issues (i.e., open-source tasks that offer rewards for completion) from a JSON file*	**Top Contributor Statistics**: *analyze three characteristics of top contributors in an open-source community from a JSON file*

information management system. Specifically, Task A involves implementing three core operations: "*addCommodity*", "*showMyCommodity*", and "*removeCommodity*".

3) Research Tool Development (RTD): This category encompasses the development of analytical scripts for research purposes, a common requirement for SE students in academic research activities. This task emphasized the SE skill of developing scripts from scratch in response to novel and specific requirements. The tasks involve processing and analyzing large-scale JSON-formatted datasets from two open-source communities. Participants are required to perform data extraction, aggregation,

and statistical analysis. For instance, one task involves identifying developers who have maintained consistent contribution patterns for over a year and rank within the top 2% based on commit frequency.

3.2.2 Task Equivalence. Each type included two sub-tasks (Task A and Task B). To enable a controlled comparison between an ACAT-assisted workflow and a traditional one, participants completed Task A with an ACAT. For Task B, the ACAT was disabled, and participants relied on standard development resources, namely the IDE's built-in syntax completion. The sub-tasks were carefully calibrated to ensure comparable complexity and workload. We implemented counterbalancing measures to minimize potential sequence effects in the experimental design.

For the first type (ADS), we selected two problems from Codeforces with identical difficulty ratings (1,100, medium difficulty) and identical pass rates (46%), ensuring consistency in difficulty. While maintaining algorithmic complexity, we modified problem contexts and descriptions to prevent solution retrieval through search engines. The sub-tasks required different algorithmic approaches, thereby minimizing learning transfer effects between tasks.

For Management System Development (MSD), equivalence between sub-tasks was established by standardizing the number of required commands and their respective conditional branches. To mitigate learning effects, participants were provided with a partially implemented system template and utility functions for string handling before the experiment. These materials familiarized participants with the system architecture and common operations used across both sub-tasks.

For Research Tool Development (RTD), task complexity was balanced by equalizing the number of statistical computations and output requirements across sub-tasks. Prior to the experiment, participants were provided with reference materials covering essential Python operations (JSON parsing, sorting functions, and data manipulation). This preparation helped standardize participants' technical knowledge and minimize sequence-dependent performance variations.

3.2.3 Validation and Pilot Study. Before the experiment, we conducted validation checks. We input the descriptions of all tasks into three ACATs, ensuring that no correct solutions are generated. Thus, we confirm that none of the tasks are in the ACATs' training datasets. Additionally, we conducted a pilot study with six CS students (three graduate and three undergraduate) who were excluded from the final experiment. They were asked to complete each type of task without using ACATs. Our observations reveal that they are capable of completing both tasks within similar timeframes (the difference being less than five minutes), regardless of the order of task completion. The completion time for the three types of tasks ranges between one and two hours. Since we plan to limit the time for each sub-task to 1.5 hours, we consider the task difficulty appropriate.

Compared to previous studies [55, 61], our tasks are relatively challenging and more reflective of common coding scenarios in an upper-level CS education. This complexity level allows us to capture more diverse and nuanced student-ACAT interaction processes. After participation, they also offer valuable suggestions for enhancing the clarity of task descriptions and learning materials.

3.3 Participants Recruitment and Screening

We recruited participants from the pool of CS students at our university. To ensure eligibility, we conducted a pre-test survey to collect data on their self-reported Python proficiency, prior experience with ACATs, and familiarity with the three task categories. The survey instruments are available in the replication package [33]. We selected applicants with a Python proficiency level of 3 or higher on a 5-point Likert scale (ranging from 1, "very inexperienced", to 5, "very experienced"). This threshold was chosen to ensure participants had sufficient foundational knowledge to engage meaningfully with the tasks and ACATs, while still representing a range of skill levels typical of CS students.

A total of 27 participants were recruited, each of whom signed a consent form prior to participation. The consent form granted us permission to collect log data, screen recordings, and conduct interviews and surveys. Each participant received ¥250 as compensation for their time. Our study was reviewed and approved by the Ethics Committee of Beihang University.

3.3.1 Participants' Demographics. As shown in Table 3, our 27 participants consist of 6 females and 21 males, with 10 being senior undergraduate students and the remaining 17 graduate students. Despite majoring in CS, all of them lacked substantial professional software development experience. In terms of their Python experience, 18 participants possess 2 to 5 years of expertise, while 9 participants have over 5 years of experience.

Table 3. Demographics of Participants (#Exp of Program = Years of experience with programming; Exp of Python, ACATs Familiarity and Tasks Familiarity are participants' self-reported expertise levels with Python, the three ACATs and the three task categorie (from 1: very inexperienced to 5: very experienced).)

ID	Gender	Status	#Exp of Program	Exp of Python	ACATs Familiarity			Tasks Familiarity			Group
					GitHub Copilot	Tabnine	CodeGeex	ADS	MSD	RTD	
Stu1	Male	Graduate student	5-10	4	5	4	4	3	3	5	CG_RTD
Stu2	Male	Graduate student	5-10	3	3	3	3	3	4	3	TAB_MSD
Stu3	Female	Undergraduate student	2-3	3	2	1	3	2	4	2	CG_MSD
Stu4	Male	Undergraduate student	2-3	4	4	4	4	3	3	4	CG_RTD
Stu5	Male	Graduate student	3-5	4	2	1	3	3	4	3	CG_MSD
Stu6	Female	Graduate student	3-5	3	3	1	1	3	3	4	COP_RTD
Stu7	Male	Undergraduate student	2-3	3	4	3	4	5	4	3	CG_ADS
Stu8	Male	Graduate student	5-10	5	4	3	3	3	4	5	TAB_RTD
Stu9	Male	Undergraduate student	3-5	3	3	1	1	4	2	3	COP_ADS
Stu10	Male	Undergraduate student	2-3	4	4	1	1	3	2	5	COP_RTD
Stu11	Male	Graduate student	2-3	3	5	3	1	3	4	3	COP_MSD
Stu12	Male	Graduate student	3-5	4	2	3	2	4	3	4	TAB_ADS
Stu13	Female	Graduate student	3-5	3	4	1	1	3	4	4	COP_MSD
Stu14	Male	Graduate student	3-5	3	4	1	1	4	4	3	COP_ADS
Stu15	Male	Undergraduate student	5-10	4	5	1	4	4	3	5	COP_ADS
Stu16	Male	Graduate student	5-10	4	5	4	2	5	5	5	COP_MSD
Stu17	Male	Undergraduate student	2-3	3	2	3	1	4	3	2	TAB_ADS
Stu18	Male	Graduate student	5-10	5	4	1	1	5	5	5	CG_RTD
Stu19	Female	Undergraduate student	3-5	3	3	2	1	4	3	3	TAB_ADS
Stu20	Male	Undergraduate student	3-5	3	4	4	2	3	4	3	TAB_MSD
Stu21	Female	Graduate student	2-3	4	4	3	3	3	2	4	TAB_RTD
Stu22	Male	Graduate student	5-10	5	4	1	1	5	5	5	TAB_RTD
Stu23	Male	Graduate student	2-3	3	5	4	4	4	2	3	CG_ADS
Stu24	Female	Graduate student	3-5	3	2	1	3	4	2	2	CG_ADS
Stu25	Male	Undergraduate student	5-10	3	4	3	3	4	4	2	TAB_MSD
Stu26	Male	Graduate student	5-10	4	4	1	3	4	4	3	CG_MSD
Stu27	Male	Graduate student	3-5	3	4	1	1	3	2	4	COP_RTD

3.3.2 Task Assignment. To reduce the impact of participants' unfamiliarity with tasks and tools on the experiments, we assigned tasks based on their self-reported expertise levels with the three task categories and the three ACATs. Our goal was to assign each participant to a combination of tasks for which they reported high familiarity (≥ 4 on a 5-point Likert scale, from 1: very inexperienced to 5: very experienced) and to ACATs with which they demonstrated moderate to high proficiency (≥ 3 on the same scale). Additionally, this expertise-based assignment strategy further reduced the potential learning curve effects during the experiments.

This objective was met for the vast majority of our assignments. Specifically, all 27 participants reported high familiarity (≥ 4) with their assigned task category. For tool proficiency, 25 out of the 27 participants met our criterion of moderate to high proficiency (≥ 3). The two exceptions were participants Stu18 and Stu22 (assigned to `CG_RTD_3` and `TAB_RTD_3`, respectively), whose cases are explicitly acknowledged as a potential limitation in our validity discussion in Section 6

Each experimental condition (ACAT-task combination) was assigned to three distinct participants to ensure data diversity and reliability. For convenient data recording and organization, we use the format of "`toolName_taskName_index`" to denote the participants, such as `COP_ADS_1/2/3` for participants who use *GitHub Copilot* to complete the "Algorithms and Data Structures" task, `TAB_MSD_1/2/3` for those who use *Tabnine* for "Management System Development", and `CG_RTD_1/2/3` for those who use *CodeGeeX* for "Research Tool Development".

Once task assignments were finalized, participants were required to perform the tasks under designated experimental conditions while recording their screens. To prevent participant fatigue and ensure data quality, a 90-minute time limit was imposed for each sub-task. Participants who did not complete a sub-task within this timeframe were instructed to stop working and submit their log data and screen recordings.

It is important to note that in assessing the efficiency of student-ACAT interactions, we deliberately abstain from measuring metrics such as task completion rates and durations, as these are significantly influenced by the varying levels of expertise among participants. Furthermore, we intentionally omit cross-group comparisons of task completion efficiency, as our primary focus is on understanding the dynamic interaction process between students and ACATs, such as user behavior, decision-making processes, and cognitive states. This approach allows us to explore the intrinsic value of student-ACAT interactions without the noise introduced by performance-based metrics.

3.4 Experimental Platform Development

To facilitate our research, we designed and developed a dedicated experimental platform. As depicted in Fig. 1, the platform integrates a VSCode IDE extension for log data collection and offers a suite of comprehensive features, including task presentation, code submission and evaluation, time tracking, screen recording, log data collection, and automated generation of personalized interview and survey questions. The platform's interface and the full experimental workflow are demonstrated in a video included in our replication package [33]. Below, we provide a brief overview of some of the platform's key functionalities.

3.4.1 Code Submission and Evaluation. The platform facilitates automated code evaluation through predefined test cases. When participants submit their solutions, they receive instant feedback, including detailed error messages for any failed test cases. Multiple submission attempts are allowed, enabling participants to iteratively improve their solutions. This iterative cycle of assessment and feedback closely replicates the real-world programming workflow and learning processes experienced by students, providing an authentic and dynamic environment for skill development.

3.4.2 Data Collection during Coding Process. The platform captures both interaction logs and screen recordings. To collect interaction log data between participants and ACATs, we developed a VSCode extension named "*ccdc-plugin*" using TypeScript and two npm packages (*Yeoman* [62] and *VS Code Extension Generator* [18]). This extension has been published on the VSCode Extension Marketplace, allowing participants to download it directly within VSCode [32]. During the experiment, participants simply press a shortcut key to accept recommended code. The extension records the accepted code, pairs it with the current timestamp to form a {*code, timestamp*} entry, and automatically saves this data as a JSON file on the user's computer. Rejected or modified

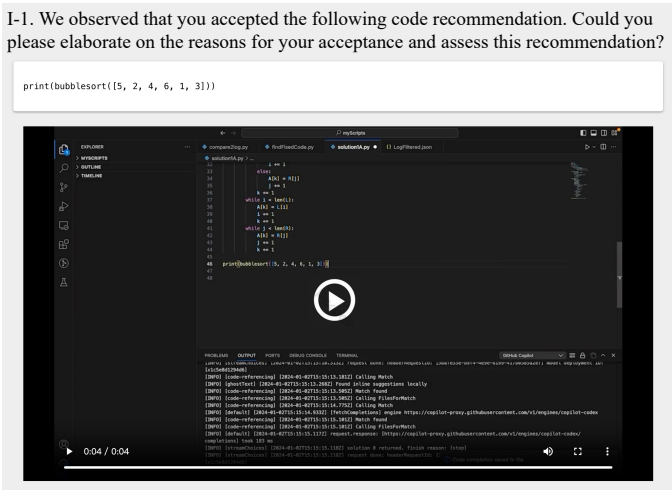


Fig. 3. Example of Personalized Interview Question

recommendations are identified by comparing the ACAT suggestions with the final implemented code.

In addition to log data, we also ask participants to record their screens from the moment they start coding until they complete Task A (or reach the 90-minute time limit), while the timer runs simultaneously. The recorded video, along with the timestamp information, provides data support for the subsequent analysis of interactions between participants and ACATs, as well as the automatic generation of personalized interview and survey questions.

3.4.3 Automatic Generation of Personalized Interview and Survey Questions. A key innovation of our platform lies in its ability to automatically generate personalized interview and survey questions by analyzing participants' interaction behavior. The system processes multiple data sources, including VSCode log files, submitted code files, screen recordings, and timestamp information. The analysis pipeline consists of three main steps: First, we examine the log files to categorize participants' interactions with code suggestions at different granularities (i.e., token-level, single-line level, and multi-line level), focusing on whether they *accepted*, *rejected*, or *modified* the suggestions. Second, we select representative cases from each interaction category and extract the corresponding video segments from the screen recordings. Finally, based on our predefined question templates (detailed in Section 3.5), we generate context-specific interview and survey questions. As illustrated in Fig. 3, this approach enables participants to review their actual coding process through video clips, facilitating more accurate recall of their decision-making processes and cognitive states during specific interactions, leading to more realistic and impressive responses.

3.5 Interview and Survey Design

To gain deeper insights into students' interactions with ACATs, we conducted post-study interviews and surveys. We first crafted a question template, which served as a foundation for automatically generating personalized questions tailored to each participant's coding process, as detailed in Section 3.4.3. The template's framework, as shown in Fig. 4, integrated eight evaluation dimensions. While two dimensions (i.e., challenges faced by students using ACATs and students' expectations for ACATs) were examined in both interviews and surveys, the questioning approaches differed

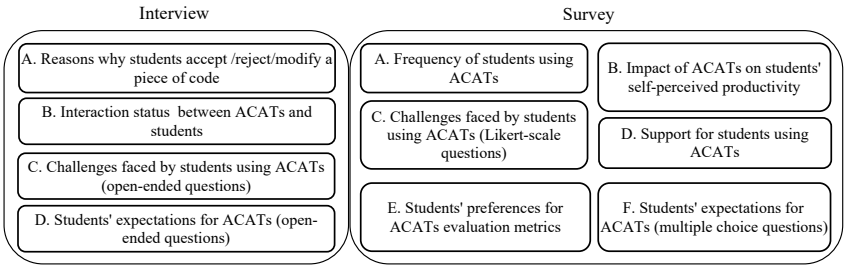


Fig. 4. Question Dimensions in Interview and Survey

substantially. Interviews were designed to elicit qualitative, open-ended responses, whereas surveys collected quantitative data through structured formats (e.g., Likert-scale questions). For each dimension, we developed specific question templates aligned with our research objectives. The complete interview protocol and survey instrument are available in our replication package [33].

3.6 Experimental Procedure

As described above, each participant was assigned to an experimental condition defined by a specific ACAT and a task category. The entire experiment was conducted remotely using our custom experimental platform. The procedure for each participant was divided into three main phases: a pre-experiment briefing and setup, the main experimental session, and a post-session data submission.

3.6.1 Phase 1: Pre-Experiment Briefing and Setup. Before the main experiment, we took several steps to prepare the participants:

- **Consent and Pre-test Survey:** Participants were first asked to complete an informed consent form, granting permission for data collection (including screen recordings), and a pre-experiment survey to gather their demographic and background information.
- **Standardized Training Session:** All consenting participants attended a unified online training session. This session covered: (1) a detailed walkthrough of the experimental workflow; (2) instructions for configuring and leveraging the experimental environment, including the assigned ACAT, our data collection plugin ("ccdc-plugin"), and the necessary Python environment; and (3) a general introduction to the three task categories.
- **Task Equivalence and Learning Effect Mitigation:** To familiarize participants with the upcoming tasks while mitigating learning effects (particularly for the self-perceived productivity measures in RQ2), we provided an introductory overview for each task category. This briefing covered the general domain and objectives of each task category, alongside an overview of the necessary technique stack, including relevant APIs and methods they were expected to utilize, but deliberately omitted the precise problem statements and functional requirements. As mentioned in Section 3.2.2, we also provided learning materials on methods and APIs that would be relevant for both sub-tasks in the "Management System Development" and "Research Tool Development" categories.

3.6.2 Phase 2: Main Experimental Session. The experimental session for each participant followed an identical structure, regardless of their assigned condition. The session was divided into two main parts: Task A, which was completed with ACAT support, and Task B, which was completed using a traditional workflow where the ACAT was disabled but the IDE's standard auto-completion remained available.

Task A (with ACAT Assistance). Participants chose a time to log into the experimental platform. The procedure was as follows:

(1) The participant initiated screen recording.
(2) On the "Experimental Introduction" page of the platform, they clicked "Start Task A" to reveal the problem description and begin the task.
(3) During development, they could use the "Code Evaluation" interface to check their solution against test cases.
(4) Upon completing Task A, or after the 90-minute time limit expired, they stopped the screen recording.
(5) They then uploaded the procedural data (screen recording and logs from "ccdc-plugin") to the "Interview" section of the platform and clicked "Generate Questionnaire and Interview Questions". This triggered our backend system to begin analyzing their interaction data to create personalized questions.

Task B (without ACAT Assistance). While the personalized questions were being generated in the background, participants immediately proceeded to Task B:

(1) To ensure a controlled environment, participants were instructed to first disable both their assigned ACAT extension and the "ccdc-plugin" within VS Code, and then restart the editor for the changes to take effect.
(2) They then clicked "Start Task B" on the platform to begin the second task.
(3) Upon completing Task B or reaching the 90-minute time limit, the programming part of the session ended.

3.6.3 Phase 3: Post-test Survey and Interview. After completing both sub-tasks, participants finalized their participation through two data submission steps:

(1) **Post-test Survey:** Participants navigated to the "Questionnaire" page on the platform, where they accessed and completed a survey via an embedded link.
(2) **Self-Administered Interview:** By this time, the personalized interview questions were available. Participants went to the "Interview" page, clicked "View My Interview Questions", initiated an audio recording software, and recorded their spoken answers to the displayed questions. Finally, they uploaded the resulting audio file to the platform.

3.7 Data Analysis

Our study employs a mixed-methods approach, encompassing both quantitative and qualitative data types. The quantitative data includes code interaction metrics (e.g., code recommendation acceptance counts) and survey responses (Likert-scale ratings), which provide measurable insights into user engagement and satisfaction. The qualitative data comprises interaction logs, screen recordings, interview transcripts, and open-ended survey responses, offering rich contextual information about participants' experiences and decision-making processes.

Regarding quantitative data, we employ descriptive statistical analysis [3, 51]. For qualitative data, the process is more comprehensive. We use general best practices of qualitative analysis [37, 59]. For the screen recordings (1,909 minutes), we conducted systematic content analysis using open coding techniques [21]. First, the first two authors reviewed all submitted videos to become familiar with the material and randomly selected recordings from three participants for preliminary annotation and discussion. This pilot phase aimed to develop and refine an initial coding framework for capturing students' interaction behaviors with ACATs. After the framework was established, both researchers independently applied the finalized coding scheme to annotate the recordings of all participants, ensuring comprehensive coverage and consistency.

To ensure the reliability of the coding process, we employed a "negotiated agreement" approach [9, 59]. While statistical metrics like inter-rater reliability are common, negotiated agreement is particularly suitable for continuous data (such as screen recordings) where defining precise segmentation units can be challenging [9]. After the independent annotation phase, the first two authors held a series of consensus meetings to compare their codes. In cases of disagreement regarding behavioral segmentation or categorization, they revisited the specific video segments to resolve discrepancies. For example, when differentiating between "**Adaption**" and "**Backtrack**" behaviors (as detailed in Section 4.3.1), the researchers clarified that "**Backtrack**" requires students to completely delete previously accepted ACAT-generated suggestions, restoring the code file to the same state it was in before the suggestion was generated. Any persistent disagreements were arbitrated by the forth author. This iterative process of comparison, discussion, and arbitration continued until 100% consensus was achieved. This rigorous analysis process required two months of manual annotation effort by two people.

For interview data, the audio recordings (225 minutes) were first transcribed verbatim for analysis. Then, the first two authors applied open coding techniques to the transcripts, following the same systematic approach used for screen recording analysis. This consistent analytical framework was extended to all qualitative data sources, including interaction logs, to ensure methodological rigor across the entire dataset.

Fig. 1 illustrates the mapping between different data sources and our research questions. Section 4 provides a detailed description of the specific data sources and analytical methods employed for addressing each research question.

4 RESULT

4.1 RQ1: What are the characteristics of student-ACAT interactions regarding usage frequency and acceptance rates across different scenarios?

In this question, we conducted granular quantitative analysis to evaluate the patterns of student-ACAT interactions. Our analysis departed from conventional approaches that primarily examine outcome-based metrics (e.g., task completion rates and duration), instead focusing on the dynamic interaction process between students and ACATs. Specifically, we measured interaction frequency (the number of ACAT-generated suggestions) and interaction success rate (the number of suggestions accepted by students) across various usage scenarios (i.e., in different task types, different tool recommendation methods and different recommendation content). Additionally, our assessment of interaction frequency is categorized into two categories: interactions related to code completion and those associated with other features offered by tools.

4.1.1 Interaction related to Code Completion Feature. We begin by presenting the overall acceptance rates for each ACAT across the various tasks. Subsequently, we conduct a more fine-grained analysis that breaks down these rates according to the different recommendation methods and content categories of the code suggestions.

1) The interaction frequency and interaction success rate of ACATs across task types. Using our extension "*ccdc-plugin*", we quantify the level of interaction by counting the total number of ACAT recommendations and the number of recommendations accepted by participants. We then compute the acceptance rates, as shown in Table 4. At first glance, the data suggests that the "Management System Development" task category achieves the highest overall acceptance rate (40%) among three task categories. However, a closer inspection reveals that this result is heavily influenced by the performance of *Tabnine*. Specifically, *Tabnine*'s acceptance rates for the "Algorithms and Data Structures" and "Research Tool Development" are notably low (17% and 19%, respectively), which significantly suppresses the overall averages for these categories.

Table 4. Code Completion Acceptance Rate

Task Category	ADS	MSD	RTD	Avg Acceptance
GitHub Copilot	75/219 (34%)	86/194 (44%)	139/353 (39%)	300/766 (39%)
Tabnine	67/392 (17%)	156/455 (34%)	95/493 (19%)	**318/1,340 (24%)**
CodeGeeX	117/230 (47%)	140/315 (44%)	122/230 (53%)	**379/755 (49%)**
Avg Acceptance	259/841 (31%)	**382/964 (40%)**	356/1,076 (33%)	

The denominator is the total number of code recommendations of an ACAT in one task type, and the numerator is the number of these recommendations accepted by participants (acceptance number). The numbers in parentheses represent acceptance rate.

If we consider the performance of *GitHub Copilot* and *CodeGeeX* independently, a different pattern emerges: the acceptance rates become consistently high and uniform across all three task types (ranging from 34% to 53%). For these two tools, there is no significant peak in the MSD category. This suggests that for more recent and powerful models, the type of programming task may have a less pronounced impact on acceptance rates than the aggregate data implies. Nevertheless, the structured and repetitive nature of MSD tasks, which involve common CRUD operations, likely contributes to its robust performance across all three ACATs, including *Tabnine*.

Among the three ACATs, *CodeGeeX* exhibits the highest acceptance rate at 49%, whereas *Tabnine* demonstrates the lowest at 24%. We further analyzed the content of the accepted suggestions to investigate the underlying factors contributing to the notably high acceptance rates observed for *CodeGeeX*. We find that a possible reason is that when predicting the next few lines after students input a "\n", *CodeGeeX* tends to suggests a single line of code, necessitating multiple triggers for a complete code block (e.g., a function body). As shown in Figure 6-W3, *CodeGeeX* generated a significantly higher volume of "next line prediction (single line)" suggestions (269 instances) compared to *Copilot* (154 instances) and *Tabnine* (149 instances), achieving a notable acceptance rate of 53.53% in this specific category. The impact of single-line generation on acceptance rate is potentially twofold. First, while the other two tools may generate a multi-line piece of correct code in one go, *CodeGeeX* is more likely to divide the code into multiple generations, leading to multiple instances of acceptance by the user. This inherently improves the acceptance rate. Second, generating smaller portions of code sequentially provides the model with a more comprehensive context for subsequent reasoning, thereby improving the accuracy of its recommendations.

Finding 1: While the overall acceptance rate of ACATs appears to peak at 40% for "Management System Development", this observation is largely an artifact of *Tabnine*'s disproportionately lower performance in the other task categories. For *GitHub Copilot* and *CodeGeeX*, acceptance rates are high and comparable across all tasks. Among the three ACATs, *CodeGeeX* leads with a 49% average acceptance rate, likely due to its preference for single-line code recommendations. Conversely, *Tabnine* has the lowest average acceptance rate (24%).

2) The interaction success rate of ACATs across recommended methods and code content. To better understand the students' engagement with and acceptance of ACATs and gain insights into their current capabilities, strengths, and limitations, we further investigate the acceptance rates of different types of code suggestions generated by these tools. Building upon previous studies on the classification of ACAT-recommended code snippets [56], we extend the type of code snippets and classify all code snippets accepted or rejected by participants. These code snippets are collected

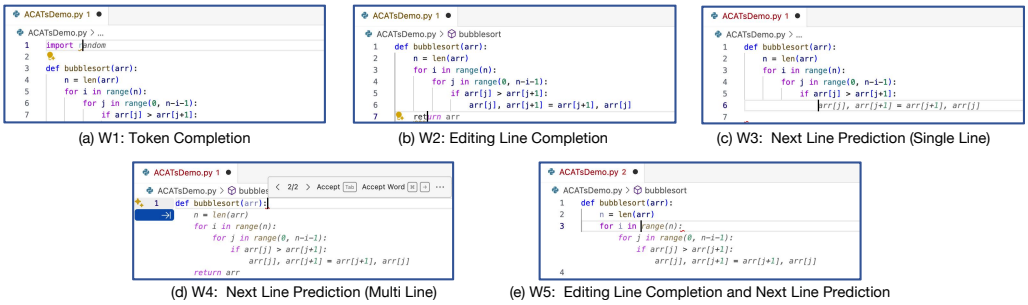


Fig. 5. Examples of Five Recommendation Method Types

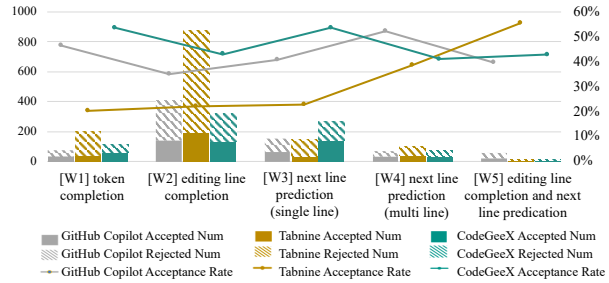


Fig. 6. Acceptance Rate of Different Recommended Ways

using our VSCode extension, i.e., "*ccdc-plugin*". The categorization process is based on two main criteria: 5 types for the recommended methods (as shown in Fig. 5) and 14 types for the code content. Specifically, Fig. 5 shows examples of the five recommendation method types: W1 (Token Completion) – completing a partial token within the current line; W2 (Editing Line Completion) – completing or editing the current line; W3 (Next Line Prediction, Single Line) – predicting one new line of code; W4 (Next Line Prediction, Multi Line) – generating multiple consecutive lines; and W5 (Editing Line + Next Line Prediction) – completing the current line while simultaneously suggesting subsequent lines. We then calculate the acceptance rates for specific types of code snippets. The results are shown in Fig. 6 and Fig. 7.

Fig. 6 illustrates the acceptance rates of different recommendation methods. Our analysis reveals that "*editing line completion*" (W2) is the most frequently generated recommendation type across all three ACATs, despite its relatively low acceptance rate. In contrast, recommendations involving multiple lines of code—namely "*next line prediction (multi-line)*" (W4) and "*editing line + next line prediction*" (W5)—occur less frequently but exhibit higher acceptance rates. This pattern indicates that students tend to prefer suggestions that offer more contextual relevance and higher-level task support (as in W4 and W5), rather than granular or partial edits (as in W1 and W2). Conversely, ACATs appear to overproduce fine-grained completions such as token or single-line edits, which may interrupt users' flow or provide limited utility. The discrepancy between the frequency of generated and accepted types thus highlights a misalignment between current recommendation strategies and user needs and preferences. These findings suggest that future ACATs should adapt their generation logic to favor context-aware and semantically meaningful completions over low-level token or line edits.

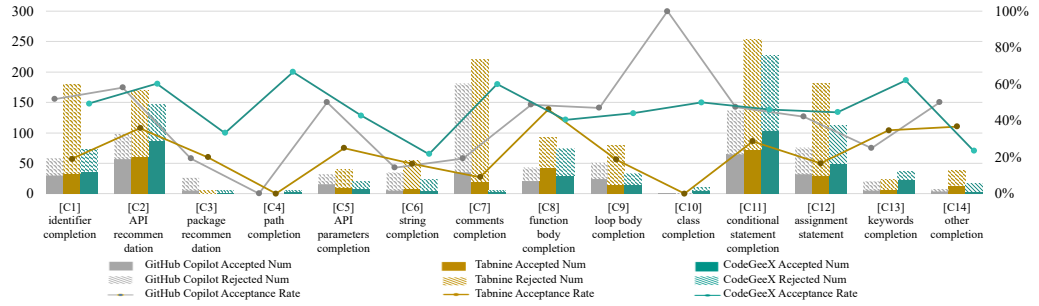


Fig. 7. Acceptance Rate of Different Recommended Code Content

Fig. 7 presents the acceptance rates of different code content types across the three ACATs. Note that "Other Completion" represents code suggestions that do not exclusively belong to any of the specific categories defined in C1 through C13. It primarily includes: (1) Hybrid Snippets: Completions that combine multiple syntactic constructs in a single suggestion (e.g., a statement containing both an API call and a complex arithmetic operation, such as "`return list.isEmpty() ? null : list.get(0);`"); and (2) Structural Fragments: Code fragments that serve structural purposes (e.g., closing parentheses followed by a colon "`):`" combined with new lines) or auxiliary syntax that lacks distinct semantic intent on its own.

Our analysis reveals that the most frequently recommended types are "*conditional statement completion*", "*API recommendation*", and "*identifier completion*". Certain recommendation types exhibit strong scenario-specific relevance. For instance, "*package recommendation*", "*path completion*", and "*class completion*" occur less frequently, likely due to their limited applicability in general development tasks. Among the more common types, "*comments completion*" and "*string completion*" have the lowest acceptance rates, indicating potential areas for tool improvement or limited practical value. These findings can guide students in reducing their reliance on such recommendations, thereby optimizing their workflow and saving time.

Moreover, each ACAT exhibits distinct characteristics in the types of code recommendations it generates. For instance, "*comments completion*" appears frequently in *GitHub Copilot* and *Tabnine* but is nearly absent in *CodeGeeX*. Similarly, *GitHub Copilot* generates "*package recommendation*" significantly more often than the other two tools. These differences in recommendation patterns highlight the unique strengths and weaknesses of each ACAT, providing valuable insights that could foster competitive advancements and innovation in the development of future coding assistance tools.

Finding 2: Regarding the recommendation methods, "*editing line completion*" is the most frequently generated type across all three ACATs, despite its relatively low acceptance rate. In contrast, recommendation methods involving multiple lines of code appear less frequently. In terms of code content, the two types with the lowest acceptance rates are "*comments completion*" and "*string completion*". Additionally, each ACAT exhibits distinct characteristics in its code recommendations. These findings offer valuable insights into the current limitations of code recommendations and highlight areas where further improvements can be made.

4.1.2 Utilization of Other Features. In addition to basic code-completion functionality, ACATs often provide advanced features, such as natural language-based chat capabilities [19]. During

interviews, we asked participants about their usage of these features. Our observations reveal that 12 out of 27 participants utilized these features: nine used the "natural language to code" feature, three employed the "debug" feature, one utilized "API usage inquiries," and one leveraged ACATs to "generate a test unit" for a function they had written. Among participants who did not use these features, we explored the reasons for non-adoption. The primary reason, cited by seven participants, was a lack of awareness that the features existed. Additionally, three participants reported that the features did not seem beneficial for completing their specific tasks.

Finding 3: Among 27 participants, 12 used advanced ACAT features: nine for "*natural language to code*", three for "*debugging*", one for "*API inquiries*", and one for "*test unit generation*". Non-users cited lack of awareness (7 participants) or perceived lack of benefit (3 participants) as reasons. These findings suggest a need for better user education to increase feature adoption and increasing the exposure of these features.

4.2 RQ2: How does interacting with ACATs affect students' self-perceived productivity?

Evaluating "self-perceived productivity" requires a clear conceptual grounding to avoid ambiguity. Therefore, before presenting the results, we first explain how we operationalize this construct. To ensure construct validity, we base our measurement on the **SPACE framework** [17], which defines productivity across five dimensions: Satisfaction and well-being, Performance, Activity, Communication and collaboration, and Efficiency and flow. Given the context of our study, we focus on three dimensions that can be reliably assessed through self-report, while excluding *Activity* (better captured through behavioral logs) and *Communication* (our tasks were individual rather than collaborative).

Specifically, we map our metrics to the SPACE dimensions as follows: (1) **Efficiency and flow**: measured through perceived *task difficulty* and the *frequency of seeking help*. Lower difficulty and fewer interruptions indicate smoother flow and higher perceived efficiency. (2) **Performance**: captured through *self-assessed individual performance*, which reflects participants' perceived output quality. (3) **Satisfaction and well-being**: measured through participants' *sentiment on reliance*, operationalized via the perceived *strenuousness* of programming without the tool. Higher strenuousness indicates that the tool reduces cognitive effort and contributes to well-being.

With this construct definition, we investigate how interacting with ACATs influences students' self-perceived productivity. Following established methodologies [1, 61], we administer a multi-dimensional post-task survey. The results are visualized across two complementary figures: Fig. 8 (capturing Efficiency and flow, and Performance) and Fig. 9 (capturing Satisfaction and well-being). Together, these figures represent the full spectrum of self-perceived productivity in our study.

Our findings highlight the significant role ACATs play in shaping students' self-perception. As depicted in Fig. 8, the perceived difficulty varied dramatically based on the availability of ACATs. When using an ACAT (Task A), nearly 30% of students rated the task as "Easy" or "Very Easy". Conversely, when deprived of ACATs for Task B, the perceived difficulty escalated, with over 50% of participants finding the same task "Difficult" or "Very Difficult".

This disparity in perceived difficulty directly translated to their self-assessed performance. For Task A (with ACAT), a majority of students (nearly 37%) rated their performance as "Good" or "Very Good". However, this confidence waned significantly in Task B (without ACAT), where approximately 55% of participants assessed their performance as "Bad" or "Very Bad". The reliance on external help (i.e., the number of instances where help was sought from resources external to the IDE, such as performing browser searches or consulting API documentation) mirrored this

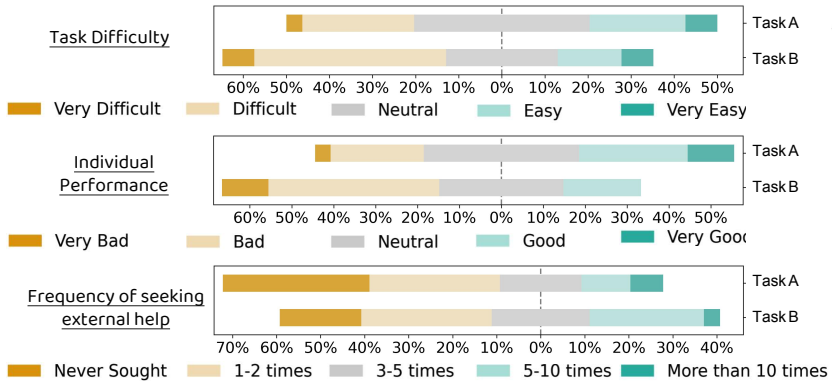


Fig. 8. ACATs' Impact on Participants' Self-perceived Productivity

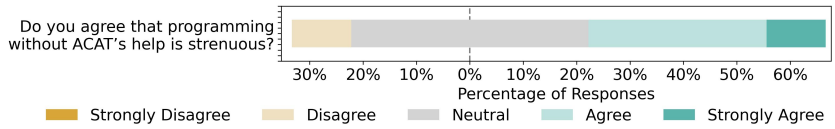


Fig. 9. Participants' Reliance on ACAT

trend, in Task A, over 60% of students "Never Sought" or only sought help "1-2 times". This changed substantially in Task B, where the absence of ACAT support compelled many to seek help more frequently, with a notable portion (nearly 30%) requiring assistance "5-10 times" or even "More than 10 times".

These task-specific findings are further corroborated by the participants' general sentiments on ACAT reliance, shown in Fig. 9. When asked if programming without ACATs is strenuous, a combined majority of participants (more than 55%) "Agreed" or "Strongly Agreed", aligning with their experience in Task B. These findings underscore the strong correlation between ACAT interaction and productivity perceptions, suggesting that ACATs have become indispensable cognitive scaffolds in students' programming workflows. They fundamentally reshape how students approach programming tasks and assess their own capabilities.

Finding 4: For the majority of participants, interacting with ACATs do reduce the perceived task difficulty, improve participants' self-performance, reduce the times of seeking external help, and generally improve the participants' programming experience. More than half (55%) of the participants feel programming more strenuous without ACAT's help.

4.3 RQ3: What are the interaction patterns of students when using ACATs?

To further analyze students' fine-grained interaction behaviors with ACATs and their frequency distribution, we conduct detailed data annotation and thematic analysis on screen recordings of participants interacting with ACATs. Specifically, the first two authors begin by reviewing all submitted recordings to familiarize themselves with the data. They then randomly select recordings from three participants (totaling 249 minutes) for preliminary annotation and discussion.

The purpose of this phase was to develop and refine a shared coding framework. Through the preliminary annotation, they propose an initial five-layer interaction behavior model, consisting of "**Prompt**", "**Understanding**", "**Decision**", "**Adaption**", and "**Backtrack**", along with an independent behavior category, "**Verification**". The model adopt the names for certain layers (e.g., "**Prompt**" and "**Backtrack**") from foundational work on developer-ACAT interactions [4, 38, 44] to keep the consistent terminology.

After finalizing the initial coding framework, the first two authors independently apply this model to code the remaining participants' recordings, refining the model based on observed interaction behaviors during participants' engagement with ACATs. We define a complete interaction as a sequence of behaviors centered around a single tool recommendation (e.g., prompting and deciding on the tool's suggestions). This segmentation method occasionally results in nested interactions (e.g., when the tool auto-completes comments while participants use comment-based prompts for assistance). Additionally, since video frames are annotated at 2 frames per second, some tool suggestions missed by participants due to rapid keystrokes are not captured in the annotation data. The entire data annotation and thematic analysis process follows the steps outlined in Section 3.7.

To capture a rich variety of interaction patterns while keeping the cost of manual analysis manageable, we adopted a stratified sampling strategy. From each of the nine experimental conditions (i.e., ACAT–task combinations), we selected the screen recording of the participant with the highest number of ACAT interactions. This ensured that our qualitative analysis covered the full range of tools and task types without being biased toward any single condition. These nine sessions, totaling 674 minutes, provided a rich dataset for analysis. After annotating the eighth video, no new interaction behaviors emerged, indicating data saturation. In total, we identified 1,257 instances of 47 distinct interaction behaviors, including 1,091 behaviors categorized under the five-layer interaction behavior model and 166 instances of "**Verification**" behaviors. The resulting five-layer hierarchical interaction behavior model is illustrated in Fig. 10.

4.3.1 The Five-layer Hierarchical Interaction Behavior Model. As shown in Fig. 10, a single interaction between a student and the ACAT consists of five layers of actions, with the last two being optional, and a behavior category of "**verification**" which is independent of the five-layer model. The interaction begins when the student initiates a "**prompt**" to engage with the ACAT. Once the ACAT provides a suggestion based on the context, the student proceeds to "**understand**" the suggestion before "**making a decision**" to accept or reject it. Afterward, the student may "**adapt**" the accepted suggestion or previous code to better fit their needs. Finally, the student may delete a previously accepted suggestion, and "**backtrack**" to an earlier stage of coding. Verification behaviors occur across multiple interaction stages; therefore, to maintain the parsimony of our interaction behavior model, we conceptualize "**verification**" as an external construct. Next, we will introduce the specific interaction behaviors observed at each layer during the experiment.

1) Prompt Method. We observe a total of four prompting methods (denoted as P1-P4). Fig. 11 demonstrates four prompting methods using an example of bubble sort algorithm. Among them, "*P1: comment prompt*" and "*P4: tool prompt*" are categorized as active prompts. "Active" indicates that participants deliberately interacted with ACATs, expecting the tool to provide suggestions that align with their needs. For example, in the "*P1: comment prompt*" approach, students provide additional information to ACATs by writing natural language comments, specifying their current needs or guiding the direction of the generated suggestions. Students typically use this prompting method when they have a solution idea for a problem or task but lack a concrete implementation. In the "*P4: tool prompt*" approach, students explicitly invoke functionalities offered by ACATs beyond standard code completion, such as bug fixing or querying the usage of a specific API. This typically

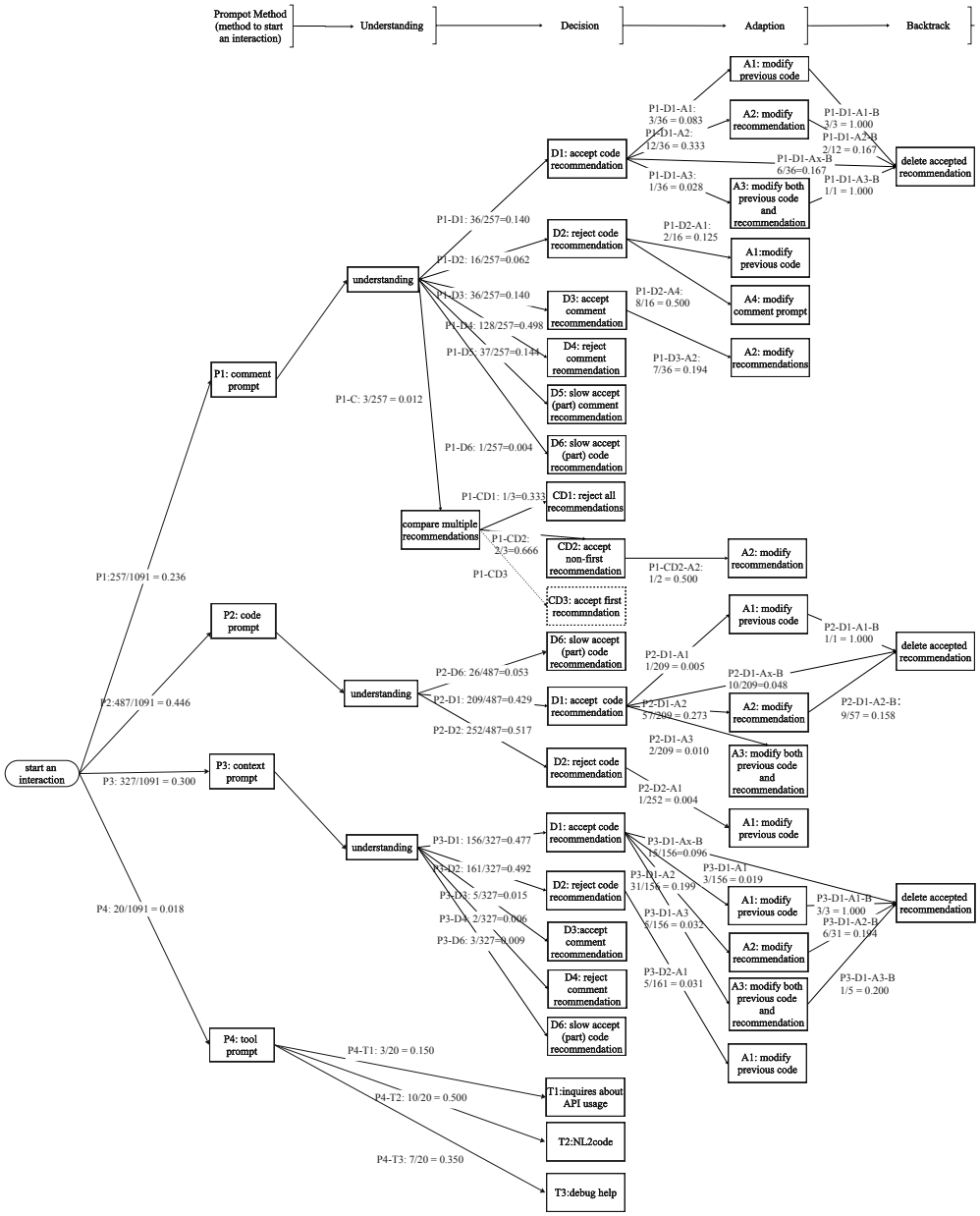
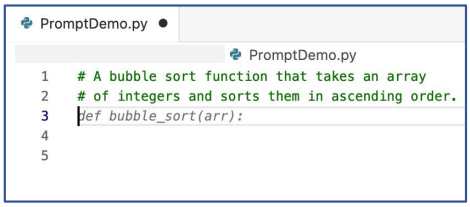
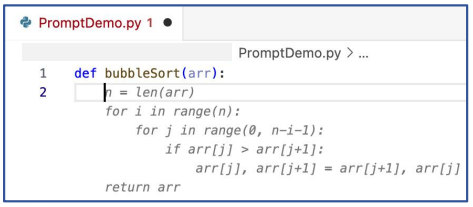


Fig. 10. The Five-layer Hierarchical Interaction Behavior Model of Comment Prompt

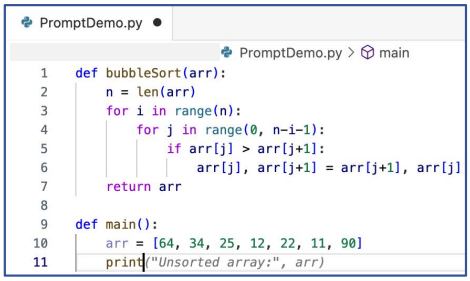
The numbers on the arrows represent the probability of state transitions.
The dashed lines indicate behaviors that are theoretically inferred from the existing results but were not experimentally observed.



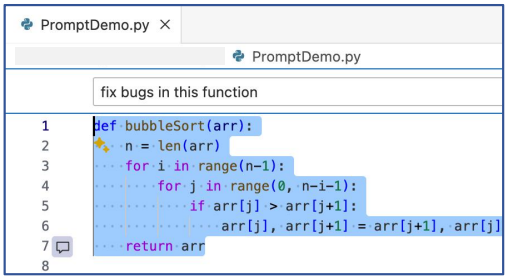
(a) "P1: comment prompt". Developers writing natural language comments to guide ACATs' recommendations.



(c) "P3: context prompt". It is triggered during pauses after students finish typing a code segment—for example, after writing a statement and pressing the Enter key.



(b) "P2: code prompt". It occurs when students are actively typing, with ACATs completing the content they are entering (e.g., filling in API parameters).



(d) "P4: tool prompt". Developers students explicitly invoke functionalities offered by ACATs beyond standard code completion, such as bug fixing.

Fig. 11. Examples of Four Prompting Methods Utilized by Participants

occurs when students face more challenging problems or when repeated interactions with ACATs fail to produce the desired results.

Passive prompts include "*P2: code prompt*" and "*P3: context prompt*". The key distinction between them lies in the timing of ACATs' suggestions. "*P2: code prompt*" occurs when students are actively typing, with ACATs completing the content they are entering (e.g., filling in API parameters). In such cases, ACATs enhance students' efficiency and reduce typing errors. In contrast, "*P3: context prompt*" is triggered during pauses after students finish typing a code segment—for example, after writing a statement and pressing the Enter key. This prompting method can also be activated when students delete a suggested segment, causing the cursor to return to the beginning of a line (i.e., backtracking). Here, ACATs predict subsequent content based on the surrounding context. While such predictions can accelerate the development of repetitive patterns and occasionally inspire students contemplating their next steps, they may also introduce inefficiencies. When predictions do not align with students' needs, reviewing lengthy code blocks can consume additional time and potentially mislead students who are uncertain about their approach.

2) Understanding. After ACAT generates a suggestion, students enter the "Understanding" phase, where they read and comprehend the tool's suggestion and evaluate its correctness and relevance to their needs. Since the "Understanding" process is inherently implicit and cannot be directly observed in the experiment, we define the time interval between the suggestion appearing and the student making a decision as the "Understanding" phase. Consequently, no specific coding is assigned to this behavior during the annotation process. However, in RQ4 (Section 4.4), we explore students' evaluation metrics and the factors influencing their decisions through timely, personalized interviews, incorporating both quantitative and qualitative analyses. Moreover, students may choose to accept the ACAT-generated suggestion first and understand and verify its correctness later. This "verification debt" aligns with findings observed in Mozannar et al.'s [38] research. We will elaborate on this behavior in the discussion of the "Verification" phase.

3) Decision. This layer encompasses nine distinct decision behaviors (denoted as D1-D9). Beyond categorizing decisions into "accept" and "reject", we further distinguish the content of the recommendations being evaluated, classifying them into "code recommendations" and "comment recommendations". This distinction is necessary because the underlying interaction intent differs significantly between these two types. For instance, when students "*accept code recommendations (D1)*", it indicates that the ACAT-generated code satisfies certain functional requirements of the participants. In contrast, when students "*accept comment recommendations (D3)*", it suggests that the ACAT-generated comments accelerate the process of guiding the tool to produce the correct code through natural language annotations, effectively speeding up the "prompt" phase.

Additionally, we also observe a "*slow accept*" behavior pattern, previously documented in Prather et al.'s research on student-ACAT interactions [44], which is a unique interaction behavior exhibited by students that distinguishes them from typical developers. This behavior occurs when students manually type out suggested code rather than immediately accepting it through keyboard shortcuts, despite the suggestion being potentially useful. However, our findings extend beyond Prather et al.'s observations by identifying that participants often partially adopted suggestions, typing only selected portions of the recommended code. And through qualitative analysis of interview transcripts and interaction recordings, we identify two primary intents underlying this behavior. First, participants employ partial slow acceptance when only specific segments of a suggestion aligned with their intended code, using the suggestion as a reference framework while maintaining control over the implementation details. Second, for brief suggestions, such as completing "`import numpy as`" with "`np`", participants often opted for manual input, perceiving the effort of manual typing as comparable to or more efficient than using acceptance shortcuts. Regardless of the underlying reason for this behavior, participants reported experiencing a sense of "validation" and "satisfaction" through "*slow accept*", where the ACAT suggestions aligned either partially or entirely with their subsequent thoughts, thereby affirming their problem-solving approach. As a participant articulated, "*The tool-generated suggestions matching the characters I was about to type substantiates the correctness of my subsequent reasoning, and the process of incrementally typing based on the tool's suggestions enhanced my confidence*".

The remaining three decision behaviors are all related to "*comparing multiple suggestions*", but we observe this phenomenon only three times in our experiment. Notably, we found that the behavior of "accepting the first recommendation after comparing multiple recommendations (CD3)" was not observed at all in our experiment. We include it in our model as a hypothetical path (represented by a dashed line in Fig. 10) to explicitly highlight its absence. This near-total lack of multi-suggestion comparison in our student participants contrasts sharply with Barke et al's study [4], where they found that a key characteristic of general developers when interacting with ACATs in "exploration mode" is their frequent use of the multi-suggestion panel to explore and compare multiple suggestions. This phenomenon reflects the difference between professional programmers and students in using ACATs. In RQ5, where we discuss the challenges that students face when interacting with ACATs (Section 4.5.1), we reveal a possible reason behind this difference.

4) Adaption. We observe four types of code adaptation behaviors (denoted as A1–A4). After accepting an ACAT-generated suggestion, participants may modify the accepted code to better fit their specific needs ("*A2: modify recommendations*"), such as changing the string in a "`print`" statement. Alternatively, they might adjust their previous code to maintain alignment with the newly accepted suggestion ("*A1: modify previous code*"), such as adding an "import" statement at the beginning of the file after accepting an ACAT-recommended API usage. In some cases, participants modified both the previous code and the accepted suggestion ("*A3: modify both previous code and recommendations*"). Additionally, under the "*P1: comment prompt*" approach, if the ACAT-generated

code based on the comment does not meet students' needs, they may reject the suggestion and then modify the natural language comment to refine their request ("*A4: modify comment prompt*").

5) Backtrack. "Backtrack" can occur for various reasons. For instance, a participant may accept a suggestion and later verify it, only to identify an issue, leading them to delete the entire suggestion. Additionally, as the task progresses and the code undergoes iterative modifications, previously accepted suggestions may become misaligned with current requirements, leading to backtracking. This often happens when students refactor their code or remove tool-generated "`print`" statements that were initially used for testing.

6) Verification. "Verification" encompasses the systematic processes through which students evaluate the correctness of AI-generated code suggestions. This critical behavior manifests across multiple interaction stages. During the "Understanding" stage, students frequently employ intuitive verification methods, such as identifying expected keywords within suggested code. In contrast, during the "Decision" and "Adaption" stages, they tend to implement more explicit verification strategies, including code execution to assess functionality. Our data revealed 166 instances of verification behaviors, categorized into two distinct types. The first category, "*verifying internal code logic*" (21 instances, 13%), resembles white-box testing methodology, where students examine the underlying logic of code segments by printing key variables or utilizing IDE debugging tools. The second category, "*verifying code output*" (145 instances, 87%), aligns with black-box testing, where students execute the code to validate results without scrutinizing internal mechanisms. This distribution demonstrates a significant preference for outcome-based verification over comprehensive logical analysis, suggesting students prioritize functional correctness rather than structural understanding when evaluating AI-generated code.

Finding 5: We propose a novel five-layer hierarchical interaction behavior model that systematically decomposes the student-ACAT interaction process into five distinct sequential stages: "Prompt", "Understanding", "Decision", "Adaption", and "Backtrack". Each stage is characterized by unique behavioral patterns that reflect students' cognitive processes while interacting with ACATs. Furthermore, our analysis reveals two distinct "Verification" strategies employed by students: "verifying internal code logic", which involves examining the structural and algorithmic components of generated code, and "verifying code output", which focuses on functional correctness through execution results. Notably, students strongly prefer output-based verification over internal logic analysis, highlighting potential implications for how they develop critical evaluation skills with ACAT-generated code.

4.3.2 Behavior Path Interpretation. We analyze the state transition probabilities of behaviors at each layer, further enriching our understanding with insights from participants' interview responses. In the following section, we *examine* the behavioral patterns observed within the five-layer interaction model. Table 5 presents interaction path information, encompassing path descriptions, global occurrence probabilities, and their respective classifications. We categorizes interaction paths into three distinct types based on participants' interview responses and our observational of screen recordings: **Efficient Success**, representing an optimal interaction pattern where AI-generated code is directly accepted without subsequent modifications or deletions; **Medium Value**, indicating an interaction path with positive potential but requiring further optimization; and **Negative Pattern**, signifying an interaction mode that produces adverse effects and urgently demands improvement. Next, we selectively interpret only the top 10 paths by global occurrence probability and several notable interaction paths of particular interest. Our investigation reveals profound insights that illuminate the intricate cognitive processes of student-ACAT collaborative coding.

Table 5. Behavior Path Transition Probabilities (Probability represents the likelihood of a specific interaction path occurring within the set of all possible interaction paths.).

No.	Path Code	Probability	Path Description	Classification
1	P2 → D2 → END	0.2297	ACAT's code completion is rejected (Main rejection path).	Negative Pattern
2	P3 → D2 → END	0.1432	Context-predicted code suggestion is rejected.	Negative Pattern
3	P1 → D4 → END	0.1175	Comment-generated comment suggestion is rejected.	Negative Pattern
4	P2 → D1 → END	0.1273	ACAT's code completion is directly used (Main success path).	Efficient Success
5	P3 → D1 → END	0.0936	Context-predicted code suggestion is directly used.	Efficient Success
6	P2 → D1 → A2 → END	0.0440	ACAT's code completion is accepted and adjusted.	Medium Value
7	P1 → D5 → END	0.0340	Comment-generated code suggestion is slowly (and selectively) accepted.	Medium Value
8	P1 → D3 → END	0.0266	Comment-generated comment suggestion is accepted.	Efficient Success
9	P2 → D6 → END	0.0236	ACAT's code completion is slowly (and selectively) accepted.	Medium Value
10	P3 → D1 → A2 → END	0.0230	Context-predicted code suggestion is accepted and adjusted.	Medium Value
11	P3 → D1 → B	0.0137	Context-predicted code suggestion is accepted then deleted.	Negative Pattern
12	P1 → D1 → END	0.0128	Comment-generated code suggestion is accepted.	Efficient Success
13	P2 → D1 → B	0.0092	ACAT's code completion is accepted then deleted.	Negative Pattern
14	P1 → D1 → A2 → END	0.0092	Comment-generated code suggestion is accepted and adjusted.	Medium Value
15	P4 → T2 → END	0.0090	Use ACAT's natural language to code feature.	Medium Value
16	P2 → D1 → A2 → B	0.0083	ACAT's code completion is accepted, then deleted after being modifying.	Negative Pattern
17	P1 → D2 → A4 → END	0.0073	Comment-generated code suggestion is rejected thus prompting student to modify the comment prompt.	Medium Value
18	P1 → D3 → A2 → END	0.0064	Comment-generated comment suggestion is accepted and adjusted.	Medium Value
19	P4 → T3 → END	0.0063	Use ACAT's code debugging feature.	Medium Value
20	P1 → D2 → END	0.0055	Comment-generated code suggestion is rejected.	Negative Pattern
21	P3 → D1 → A2 → B	0.0055	Context-predicted code suggestion is accepted, then deleted after being modifying.	Negative Pattern
22	P1 → D1 → B	0.0055	Comment-generated code suggestion is accepted then deleted.	Negative Pattern
23	P3 → D2 → A1 → END	0.0046	Context-predicted code suggestion is rejected thus prompting student to modify existing code.	Medium Value
24	P3 → D3 → END	0.0045	Context-predicted comment suggestion is accepted.	Efficient Success
25	P3 → D1 → A3 → END	0.0037	Context-predicted code suggestion is accepted. Then student simultaneously modifies suggestion and existing code to ensure compatibility.	Medium Value
26	P1 → D1 → A1 → B	0.0027	Comment-generated code suggestion is accepted then deleted after student modifying existing code.	Negative Pattern
27	P3 → D1 → A1 → B	0.0027	Context-predicted code suggestion is accepted, then deleted after student modifying existing code.	Negative Pattern
28	P3 → D6 → END	0.0027	Context-predicted code suggestions is slowly (and selectively) accepted.	Efficient Success
29	P4 → T1 → END	0.0027	Use ACAT's API inquiry feature.	Medium Value
30	P2 → D1 → A3 → END	0.0019	ACAT's code completion is accepted. Then student simultaneously modifies suggestion and existing code to ensure compatibility.	Medium Value
31	P1 → D2 → A1 → END	0.0018	Comment-generated code suggestion is rejected, thus prompting student to modify existing code.	Medium Value
32	P1 → D1 → A2 → B	0.0018	Comment-generated code suggestion is accepted then deleted after being modifying.	Negative Pattern
33	P3 → D4 → END	0.0018	Context-predicted comment suggestion is rejected.	Negative Pattern
34	P2 → D1 → A1 → B	0.0010	ACAT's code completion is accepted, then deleted after student modifying existing code.	Negative Pattern
35	P1 → D6 → END	0.0009	Comment-generated suggestion is slowly (and selectively) accepted.	Medium Value
36	P1 → D1 → A3 → B	0.0009	Comment-generated comment suggestion is accepted, then deleted after student simultaneously modifying suggestion and existing code.	Negative Pattern
37	P1 → CD1 → END	0.0009	No suggestion is adopted after comparing multiple comments.	Negative Pattern
38	P1 → CD2 → END	0.0009	Select the second-best suggestion after comparing multiple comments.	Medium Value
39	P1 → CD2 → A2 → END	0.0009	Select the second-best suggestion and adjust it.	Medium Value
40	P2 → D2 → A1 → END	0.0009	ACAT's code completion is rejected thus prompting student to modify existing code.	Medium Value
41	P3 → D1 → A3 → B	0.0009	Context-predicted code suggestion is accepted, then deleted after student simultaneously modifying suggestion and existing code.	Negative Pattern
42	P1 → CD3 → END	0.0009	Select the first suggestion after comparing multiple comments.	Medium Value

Sorted in descending order based on the global probability of path.

1) Among the top 10 interaction paths by occurrence probability, all paths except the sixth and tenth terminate at the third layer ("Decision"). This observation suggests that students' interaction behaviors are relatively straightforward, with few instances of suggestion modifications and backtracking. These interaction paths primarily focus on accepting or rejecting suggestions generated by three prompt types: "*P1: comment prompt*", "*P2: code prompt*", and "*P3: context prompt*". Given the results from RQ1, which revealed that the acceptance rates for these ACATs suggestions do not exceed 50%, the top three interaction paths consistently involve rejecting suggestions (and terminating the interaction). Among these, "*P2-D2*" represents the primary rejection path. This indicates that ACATs need to improve the quality of their generated recommendations.

In RQ1, we conducted a detailed analysis of the probabilities of different suggestion types being accepted across various scenarios for different tools, identifying the recommendation types where current tools demonstrate limitations. This analysis provides valuable insights from an ACAT perspective, offering potential strategies to enhance suggestion quality and, consequently, improve interaction success rates.

2) The acceptance rate of suggestions in the active prompting method is lower than in passive prompting methods. We analyze the suggestion acceptance rates across the three prompting methods and find that the acceptance rate for the active prompting method "*P1: comment prompt*" (43.2%, with 257 generated suggestions) is lower than those for the two passive prompting methods, "*P2: code prompt*" (48.2%, with 487 generated suggestions) and "*P3: context prompt*" (50.1%, with 327 generated suggestions). One major reason for this discrepancy is the high frequency of the "*D4: reject comment recommendation*" behavior following "*P1: comment prompt*"—occurred in 49.8% of the 257 generated suggestions under this prompting method. This indicates that the tool frequently generates additional comment type suggestions in response to a comment prompt, which students often reject.

Through comprehensive analysis of interviews and screen recordings, we identify several reasons for this rejection: a) Weaker natural language modeling – The underlying model is trained more extensively on code than on natural language, making it less effective at predicting user intent when processing natural language inputs; b) Interruptions to the user's thought process – When participants enter natural language prompts, they are often actively engaged in formulating their approach, typing with a coherent flow. Compared to other cases, they are less willing to accept suggestions that might interrupt their train of thought; c) Mismatch between intent and generated suggestions – Participants use comments primarily to prompt the tool for code suggestions, but the tool sometimes continues generating additional comments instead. In extreme cases, such as with Tabnine, we observe instances where typing "#" led to the tool generating a suggestion like "#########", a clear bad case of unwanted comment generation.

3) "*Slow accept*" interaction path is effective. Beyond directly accepting all tool suggestions via keyboard shortcuts, "*slow accept*" represents another prevalent interaction pattern for partially accepting tool recommendations. The interaction paths "*P1→D5*" (prompting via natural language comments → slow, partial acceptance of comment suggestions) and "*P2→D6*" (code prompt → slow, partial acceptance of code suggestions) are also among the top 10 most frequent interaction modes, ranking 7th and 9th respectively.

Interestingly, we find that **no "*backtrack*" behavior was observed following any "*slow accept*" behavior, regardless of the participant's prompt strategy (0/67)**. This might be closely tied to the contextual background of such behavior. First, compared to directly accepting entire suggestions via shortcut keys (i.e., "*D1: accept code recommendation*" and "*accept comment recommendations (D3)*"), participants seemed to develop a deeper understanding of the suggestion when employing "*slow accept*" (thus potentially triggering partial acceptance behaviors). Second, during "*slow accept*", the entire process remained in a state of comprehension and evaluation,

with extended assessment and reading time further deepening understanding and ensuring that only the code segment truly meeting requirements was accepted. Finally, the alignment between participants' own thoughts and ACAT suggestions under "*slow accept*" not only reinforced students' confidence but also enhanced their trust in the suggestions.

These observations are supported by our qualitative coding, which showed that slow-accept episodes consistently co-occurred with evaluation-related behaviors (e.g., intent checking, meaning verification) that were largely absent in direct-accept cases, providing analytic evidence that slow accept reflects a deliberate learning strategy rather than reactive adoption.

4) In the "Adaptation" layer, following the acceptance of code suggestions, students predominantly opt to directly modify the accepted code ("*A2: modify code recommendations*"), regardless of the prompting method used. Notably, among the top 10 most frequent interaction paths, only two involve this adaptation behavior - ranking 6th ("*P2→D1→A2*") and 10th ("*P3→D1→A2*") respectively. Moreover, across different prompting approaches, "*A2: modify code recommendations*" remains the predominant modification strategy that participants employed after receiving code suggestions ("*D1: accept code recommendation*"). Under "*P1: comment prompt*", out of 16 adaptation behaviors following "*D1: accept code recommendation*", 12 (75%) involved "*A2: modify code recommendations*." Under "*P2: code prompt*", 57 (95%) involved "*A2: modify code recommendations*" out of 60 adaptation behaviors and 31 (79%) out of 39 adaptation behaviors under "*P3: context prompt*." Additionally, we find that **whenever students engaged in "*A1: modify previous code*" after accepting a code suggestion, it was almost always followed by a "*backtrack*" behavior (7/7 cases)** (e.g., path "*P1→D1→A1→B*", "*P3→D1→A1→B*", and "*P2→D1→A1→B*"). Although the sample size is small, this high proportion suggests that **modifying previously written code to accommodate accepted suggestions may lead to wasted time and effort**.

These observations, supported by our qualitative coding, suggest that A1 behavior reflects inefficient effort rather than productive refinement: episodes were consistently associated with codes indicating rework, and these codes rarely appeared in A2 cases. The convergence of these coded patterns with consistent backtracking provides analytic evidence that modifying previous code often leads to redundant or wasted effort.

This underscore **the critical importance of strategic code integration when working with ACATs**. The high correlation between extensive code modifications and subsequent backtracking implies that CS students should critically evaluate whether an AI suggestion truly aligns with their existing code structure and programming intentions before undertaking significant code adjustments.

5) "*P4: tool prompt*"-related interaction paths exhibit low probability, suggesting a need to enhance feature visibility. The most frequent "*P4: tool prompt*" interaction paths ranked only 15th. Furthermore, our RQ1 results revealed that among the 27 experiment participants, merely 12 (less than 50%) utilized the tools' additional functionalities. Participants who did not use these features primarily cited "unclear feature guidance" or "lack of awareness (uncertainty about feature existence)" as significant barriers. These findings imply that ACAT designers should prioritize increasing the exposure of these features. Potential strategies include developing more user-friendly and transparent documentation or implementing intelligent proactive guidance that triggers when student developers encounter difficulties—such as API usage errors or persistent debugging challenges that remain unresolved after multiple attempts.

6) The same interaction behavior may arise from multiple underlying intents. This phenomenon manifests in two key ways. First, a single behavior within a given interaction path can have multiple interpretations. For example, as discussed earlier, the behavior "backtrack" can occur for various reasons. Second, the same behavior may carry different intents depending on the interaction path in which it appears. For instance, in the "Adaptation" layer, the behavior

"*A1: modify previous code*" may have different meanings depending on the preceding decision. When it follows "*D1: accept code recommendation*" in the "Decision" layer, the intent is typically to align earlier code with the newly accepted suggestion. However, when it follows "*D2: reject code recommendation*", the intent may differ significantly—students might not accept the suggestion but instead use it as inspiration to revise their prior code. This revision could aim to correct errors or adjust the context in which ACATs generate code, potentially improving future recommendations. This observation underscores the importance of analyzing students' interaction intents within the specific context of their interaction paths, rather than relying on generalized assumptions.

7) A rejected suggestion does not necessarily indicate an unproductive interaction. Even when a suggestion is declined, it may still provide participants with insights into potential code improvements, whether by directly addressing existing issues or refining the prompt to guide future suggestions more effectively. This is particularly evident in the "*P1: comment prompt*" strategy, where we observed multiple instances of the "*P1-D2-A*" (i.e., the process of "*using comments to guide suggestion generation → rejecting the code suggestion → modifying the code*") interaction pattern. This iterative process suggests that participants continuously refine their inputs to achieve their intended outcomes, demonstrating the value of rejected suggestions in the broader problem-solving workflow.

Finding 6: Our analysis of CS students' interaction with ACATs reveals several significant patterns.

- Students' interaction behaviors are relatively straightforward, with the top 10 interaction paths except the sixth and tenth terminating at the third layer ("Decision").
- Active prompting methods (comment-based) demonstrate lower acceptance rates (43.2%) compared to passive approaches (code-prompt: 48.2%, context-prompt: 50.1%).
- When students carefully evaluate suggestions before accepting ("slow accept"), these are not backtracked, representing an optimally efficient ACAT interaction pattern.
- In adaptation behaviors following accepted suggestions, students predominantly modify the accepted code (75-95% across prompting methods) rather than altering their previous code.
- When students do modify previous code after accepting suggestions (which occurred rarely), this almost invariably led to "Backtrack" behaviors (7/7 cases), suggesting that retrofitting existing code to accommodate new suggestions may result in inefficient workflows.
- "*P4: tool prompt*"-related interaction paths exhibit low probability, suggesting a need to enhance feature visibility.
- Identical interaction behaviors may stem from diverse underlying intents, with interpretations varying based on both immediate context and preceding interaction paths—highlighting the necessity of contextual analysis when evaluating user intent.
- Rejected AI suggestions still contribute significantly to problem-solving processes, often providing valuable insights that inform subsequent code modifications or prompt refinements.

These findings collectively underscore the complex, path-dependent nature of student-ACAT interactions and highlight opportunities for improving ACATs' alignment with users' cognitive processes.

4.4 RQ4: What factors influence students' evaluations and decisions when interacting with ACATs?

To comprehensively investigate factors influence participants' decision-making during interactions, we employ a mixed-methods approach combining quantitative surveys of evaluation metrics with qualitative interviews exploring participants' decision-making rationales for accepting, rejecting, or modifying recommended code. As stated in Section 3.4.3, the interview questions are automatically generated by the experimental platform we designed, based on the participant's actual interaction process with ACATs. Each interview question is contextualized with specific code recommendations generated during the experiment, accompanied by corresponding interaction video recordings (refer to Fig. 3). This stimulated recall approach, where participants reviewed their actual interaction sessions, facilitated more accurate reconstruction of their cognitive processes and decision-making contexts. The integration of quantitative metrics with contextually-grounded qualitative insights provides a nuanced understanding of participants' interaction patterns and reasoning processes with ACATs.

4.4.1 Evaluation Metrics during Interactions. The evaluation metrics used in this study were primarily derived from prior literature on code generation and recommendation systems [11, 31, 56]. These works commonly emphasize dimensions such as correctness, readability, and maintainability when evaluating AI-assisted coding support. Building on these studies, we selected a set of metrics that capture both the intrinsic quality of generated code and the perceived usability of the tool in an educational setting. Specifically, we categorized the metrics into two dimensions: (a) *code-quality-related metrics*, which assess the technical soundness and clarity of generated code (e.g., correct syntax, similarity to correct code, readability, consistent code style, inclusion of comments); and (b) *usability-related metrics*, which concern the tool's interaction experience and practicality (e.g., ranking of recommendations, support for multiple scenarios, code length).

Our analysis of participants' perspectives on these metrics revealed several key priorities (Fig. 12). Two metrics emerged as paramount: "*correct syntax*" and "*similarity to correct code*", with 97% of participants rating them as (extremely) important. This emphasis reflects students' practical needs: correct syntax ensures error-free code, preventing frustrating debugging efforts, while similarity to correct code ensures that the generated code aligns with the intended functionality and logic of the program. "*Readability*" emerges as the third most critical factor, with 88% of participants rating it as (very) important, highlighting its role in facilitating code comprehension and long-term maintenance efficiency.

Three additional metrics garnered significant attention, each rated as important by over 70% of participants: "*consistent code style*", "*support for multiple scenarios*" (e.g., identifiers, keywords, and multi-line completions), and "*inclusion of code comments*". While 70% of participants considered "*ranking of recommendations*" (very) important, 30% expressed neutrality, indicating they primarily focus on the first suggestion rather than examining the entire recommendation list. Notably, "*code length*" elicited diverse responses, showing no clear consensus among participants.

Finding 7: Participants offer insights on evaluation metrics during interactions, highlighting factors such as "correct syntax" and "similarity to correct code". "Readability" and other considerations like "support multi-scenarios" are also important. Opinions vary on factors like "ranking of recommendation" (with some participants focusing solely on the top suggestion) and "code length".

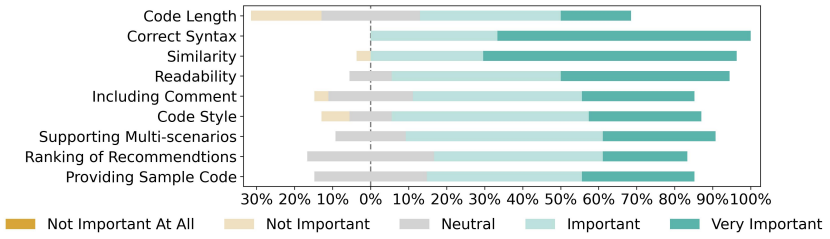


Fig. 12. Perspectives on the Evaluation of ACATs

4.4.2 Reasons Affecting Participants' Decisions. Table 6 presents a comprehensive analysis of factors influencing participants' decisions regarding ACAT code recommendations, encompassing 22 distinct reasons for acceptance, rejection, or modification. While most reasons are self-evident, several key factors warrant detailed discussion. Notably, "*fulfilling functional requirements*" emerges as the universal criterion for acceptance, cited by all participants. This encompasses fundamental code completion categories including *"If" logic*, *identifiers*, *API recommendation*, *class definitions*, and *standard input/output operations*, demonstrating ACATs' competence in these foundational programming aspects. Additionally, "*reducing keystroke numbers*" and "*offering a framework, structure, or idea for development*" are frequently cited as primary motivators for recommendation acceptance, highlighting ACATs' role in enhancing coding efficiency and providing architectural guidance.

Analysis of recommendation rejection patterns reveals several critical limitations in current ACAT implementations. The most frequently cited concern is the generation of unsuitable code, manifesting as "*irrelevant code, logical errors, detailed errors, or inconsistent coding styles*". A significant technical constraint emerges in the form of "*missing completions due to fast keystrokes*", highlighting timing synchronization issues between user input and ACAT response. Notably, one participant reported an instance where the "*ACAT misinterpreted the intended completion context*", attempting to complete comments rather than implementing the functionality described within those comments. These findings underscore the need for ACATs to evolve beyond surface-level code generation towards more sophisticated context awareness and intention understanding capabilities.

Our analysis identifies eight distinct patterns in code recommendation modifications. Interestingly, some frequently mentioned reasons overlap with those for rejecting recommendations, including "*adjusting input/output format*", "*enhancing logic details*", "*correcting errors*", and "*standardizing code style*". Beyond these common factors, participants revealed insightful perspectives on modification necessity. A particularly noteworthy observation emerges from three participants who highlighted the propagation effect of contextual errors: when the surrounding code contains imperfections, the context-dependent nature of ACAT recommendations necessitates corresponding modifications to maintain code consistency and correctness.

Finding 8: We identify 22 reasons affecting participants' decisions. They accept recommendations to fulfill functional needs, reduce keystrokes, and gain development ideas. However, they reject recommendations due to irrelevant or erroneous code and misunderstandings. Reasons for modifying recommendations include error correction and contextual adjustments. These findings underscore the need for improvements in ACATs, emphasizing the importance of code relevance, accuracy, adaptability, and a deeper understanding of developers' intentions and contexts.

Table 6. Reasons Affecting Participants' Decisions

Decision	Reason
Accept	1. Fulfilling functional requirements (27)
	2. Reducing the number of keystrokes required (6)
	3. Offering a framework, structure, or idea for development (5)
	4. Generating simple and repetitive content efficiently (4)
	5. Aligning with intuitive coding practices (3)
	6. Providing comprehensively completed code (1)
Reject	1. Generating irrelevant code (4)
	2. Missing completion due to too fast keystrokes (4)
	3. Containing logic errors in the generated code (3)
	4. Exhibiting detailed errors, such as output format mismatch (3)
	5. Demonstrating varying code styles (3)
	6. Being difficult to comprehend (1)
	7. Producing overly long code (1)
	8. Misinterpreting the developer's intention (1)
Modify	1. Adjusting the input/output format to align with requirements (8)
	2. Enhancing the logical details of the tool's recommended code (8)
	3. Correcting errors in the generated code (5)
	4. Optimizing the generated code to achieve the desired outcome (4)
	5. Standardizing the code style (3)
	6. Fixing issues in the contextual code to optimize the generated code (3)
	7. Adjusting the complex logic of the generated code (1)
	8. Removing code completions accepted during debugging (1)

Numbers in parentheses indicate #participants mentioning this item.

4.5 RQ5: What are the challenges and expectations when students interacting with ACATs?

Analysis of the interview data uncovers diverse challenges and expectations experienced by CS students during their ACAT interactions. While previous studies have investigated this topic across various populations including developers and students [1, 11, 31, 56], our approach of employing personalized, process-driven interview questions enabled more nuanced insights into student-ACAT interactions (as detailed in Section 3.4.3). In subsequent discussions, we also compare our results with existing research.

4.5.1 Challenges. Table 7 details participants' 11 challenges during the interaction with ACATs, divided into functional and non-functional categories. The bold items represent new challenges identified in our experiments, compared to previous studies [1, 11, 31, 56].

1) Functional Challenges. This category revolves around code completion, especially ACATs' "*poor performance on tasks with complex logic*" (mentioned by 44% of participants). Currently, ACATs may encounter difficulties in grasping the nuances and subtleties of the code, necessitating the pursuit of developing more sophisticated algorithms and leveraging larger datasets for training.

The second mostly mentioned challenge is ACATs' "*poor natural language understanding capability*". Natural language understanding demands linguistic knowledge coupled with a profound comprehension of relevant domains or context. ACATs may fall short in this regard. As participant

Table 7. Challenges When Participants Using ACATs

Category	Challenge
Functional Challenge	1. Poor performance on tasks with complex logic (12)
	2. Poor natural language understanding capability (12)
	3. Insufficient support for multi-type completion (10)
	4. Too long/short completion (3)
	5. Poor performance on debug (2)
Non-functional Challenge	1. Unreasonable way of displaying and switching candidate recommendations (19)
	2. Slow response time (10)
	3. **Other UI-related issue (8)**
	4. **Obscure documentation (3)**
	5. Data leakage (1)
	6. Difficulty in configuration (1)

1. Numbers in parentheses indicate #participants mentioning this item.
2. The bold items represent new challenges identified in this study compared to previous studies [31, 56].

TAB_ADS_1 said, "*After completing the function declaration, I attempted to describe to ACAT what I intended to achieve with the function using natural language comments, but it did not fully comprehend and returned code that did not match my description*".

Ten participants also mentioned the challenge of "*insufficient support for multi-type completion*". For instance, two participants suggested that beyond basic code completion, ACATs should automatically recommend relevant packages along with API suggestions. The "*length of recommended code*" is also a challenge. Participants expressed diverse opinions. Some complained that brief code recommendations may not fully meet their needs, while others found longer ones overwhelming. Hence, a pivotal question emerges: What's the ideal length of code recommendations, and how does it correlate with the context?

2) Non-functional Challenges. Participants also mentioned six non-functional challenges. The most frequently mentioned issue is the "*unreasonable way of displaying and switching candidate recommendations*". During the experiment, only 3 out of 27 participants viewed or selected other candidate suggestions provided by ACATs. When asked about this in our interviews, fourteen participants stated that "they did not notice that the tool offered other candidates". As participant COP_MSD_2 said, "*I've been using Copilot for a long time, but I only recently realized that it provides more than one code recommendation at a time, and the candidate suggestions are so inconspicuous!*" Five participants acknowledged the presence of multiple recommendation codes but pointed out that "*the inconvenient way of switching between different candidates affects the willingness to switch. Moreover, the fact that only one candidate can be displayed at a time makes it difficult to compare different options*".

The second mostly mentioned non-functional challenge is "*slow response time*", which is mainly related to NL2Code feature. The next most mentioned challenge relates "*other issues of UI*", including "*low visibility of auxiliary features (e.g., NL2Code and code repair)*", and "*disappearance of completion suggestions after switching applications*".

Finding 9: Participants face challenges when interacting with ACATs, including ACATs' poor performance in handling complex logic, limited natural language understanding, insufficient support for multi-type completion, and varying preferences regarding the length of code recommendations. There are also non-functional issues, such as unreasonable ways of displaying and switching code recommendations, slow response times, and various UI problems.

4.5.2 Expectations. Participants totally expressed 23 expectations to improve interaction experience with ACATs, belonging to functional and non-functional categories, as shown in Table 8. Compared to previous studies [1, 11, 31, 56], our experiments newly identify 16 additional challenges.

1) Functional Expectations. Participants expressed ten expectations about enhancements to current features. Over half of participants emphasized the desire to "*enhance natural language understanding and interaction capabilities*". It is worth noting that, in our study, *CodeGeeX* offers native bilingual (Chinese and English) support, whereas *GitHub Copilot* and *Tabnine* achieve UI localization via a standard VS Code extension. While this meant the visual interface for code completion was largely consistent across tools for participants who preferred Chinese, their feedback was not aimed at the UI itself. Rather, it targeted the quality of AI-generated content when prompts or queries were made in Chinese for features like NL2Code or code explanation. This clarifies that the primary expectation is for deeper semantic understanding and more accurate generation in multilingual contexts, directly informing the top-ranked functional expectation. Although it is not directly related to code completion, participants' concerns highlight the unmet need for the improvement of NL2Code functionality. The second expectation centers on "*optimized ranking of code suggestions*", with three participants also expressing a desire to *refine the UI and selection of multiple suggestions*. Sixty-seven percent of participants indicated that they are willing to review the top three code suggestions, emphasizing the importance of "*optimized ranking*" in enabling student developers to swiftly identify and select the most pertinent and precise code snippets, thereby streamlining the interaction process. Seven participants further hope to enhance the "*accuracy of debug functionality*". This underscores the evolving perception of ACATs, from mere code completion tools to comprehensive assistants that can facilitate various coding aspects, ultimately facilitating students' acquisition and development of comprehensive skills across the entire software development lifecycle. Participants also mentioned expectations like "*avoiding lengthy code suggestion*" and"*adaptive learning of personal coding style*".

Participants further mentioned six new features that they consider necessary additions to ACATs. Five participants expressed the hope that ACATs can *directly generate frameworks based on natural language*, particularly during the initial stages of development. This is due to the convenience and efficiency it offers, enabling students to quickly set up the foundation of their projects using natural, intuitive language rather than spending time manually configuring frameworks. Another similar expectation is "*capability to help understand project-level code*". Some expectations are novel and interesting. For example, participants hope to *enhance the multi-modal/visualization capabilities* of ACATs and to *incorporate support for verbal chat*, which suggests a desire for a more interactive and intuitive coding experience.

2) Non-functional Expectations. Participants expressed three key non-functional expectations: *accessibility*, *response speed*, and *system/UI design*. Regarding accessibility, they hope that ACATs can *support local connections*, minimizing the impact of network quality on their usage. Cost is another factor, with participants favoring *more affordable solutions*. In terms of response speed, they desire faster code completion and NL2Code responses. Lastly, for system/UI design, participants

Table 8. Expectations When Participants Using ACATs

Category	Sub-category	Expectation
Functional Expectation	Enhancements to existing features	1. Enhanced natural language understanding and interaction capabilities (15) 2. Optimized ranking of code suggestions (13) 3. Improved accuracy of code auto-completion (7) **4. Improved accuracy of debug functionality (7)** **5. Extended length of recommended code blocks (4)** **6. Clear display, selection, and configuration of multiple suggestions for code completion (3)** **7. Avoidance of lengthy code suggestions (2)** 8. Enhanced understanding of user Intent (2) 9. Adaptive learning of personal coding style (2) **10. Enhanced support for Chinese language (2)**
	Adding new features	**1. Direct framework generation from natural language requirements at initial stages (5)** **2. Capability to help understand project-level code (2)** **3. Programming language tutoring for beginners (2)** **4. Addition of multi-modal/visualization capabilities (2)** **5. Expansion of application scenarios, including web architecture suggestions (1)** **6. Addition of verbal chat functionality (1)**
Non-functional Expectation	Accessibility	1. Local connectivity capability (5) **2. More affordable costs (2)**
	Response speed	1. Faster completion response (6) **2. Faster NL2Code response (2)**
	System/UI Design	**1. User-friendly documentation with easy visibility (2)** **2. More prominent other features or text guidance for user convenience (1)** **3. Personalized features for different user groups (1)**

1. Numbers in parentheses indicate #participants mentioning this item.
2. The bold items represent new challenges identified in our experiments, compared to previous studies [31, 56].

want *user support and other features (e.g., code comprehension, code repair, and NL2Code)* to be easily visible and accessible.

Finding 10: Participants express 23 expectations for ACATs, including enhanced natural language understanding, optimized code suggestion ranking, improved debug accuracy, and adaptive learning of coding style. They also desire features like direct framework generation, project-level code understanding, and improved multi-modal capabilities. Non-functionally, participants want local connection support, affordability, faster response times, and user-friendly design.

5 IMPLICATIONS

Our study highlights several implications for ACAT designers, CS students, and SE researchers.

5.1 Implications for ACAT Designers

By understanding the differences between various ACATs and their performance in different application scenarios, designers can implement targeted optimizations to better address students' learning needs. RQ1 examines student-ACAT interactions through multiple dimensions, including the usage frequency and acceptance rates of AI-recommended code across task types, recommendation approaches and content categories. This systematic evaluation offers insights into students' engagement with and acceptance of ACATs, providing empirically-grounded implications for design improvements.

1) Our comparative analysis of three popular ACATs highlights specific features that impact students' learning processes. For instance, while *Tabnine*'s rapid suggestion generation demonstrates technical responsiveness, it may inadvertently hinder students' cognitive processing and learning opportunities. Students often miss potentially valuable suggestions due to the constant stream of recommendations. This suggests the need for adaptive suggestion timing that aligns with students' individual coding rhythms and cognitive load, allowing them to meaningfully engage with and learn from AI recommendations while maintaining their programming flow.

2) ACAT designers could develop targeted features that scaffold students' learning in complex programming domains, addressing the current limitations in support for higher-order programming skills. The task-specific performance analysis reveals that ACATs currently provide stronger support for structured programming tasks, particularly in Management System Development, while demonstrating notable limitations in more complex problem-solving scenarios. This finding exposes a critical gap in supporting students' comprehensive programming learning experiences. While existing tools offer adequate assistance with routine coding tasks, they fall short in supporting crucial cognitive skills such as sophisticated API integration. The research underscores the necessity of designing adaptive assistance mechanisms that can effectively bridge the current support deficit across diverse programming contexts.

3) ACAT designers can focus on improving the relevance and quality of specific types of recommendations. RQ1 also reveals the acceptance rates of AI-recommended code across content categories. Specifically, we observe that "comments completion" and "string completion" have the lowest acceptance rates among multiple types of recommendations. For example, several participants mention that "*I know that the tool supports generating corresponding code based on the comments I write, and I have tried to use it this way. However, sometimes the tool just continues to complete my comments instead of generating the corresponding code*". By analyzing why these suggestions are often rejected, designers can refine the algorithms to provide more contextually appropriate completions.

Additionally, **ACAT designers can improve suggestion generation by tracking and analyzing the student's current state to accurately infer their intent**. The findings from RQ3 reveal that the same interaction behavior may stem from multiple underlying intents, which are closely related to the student's current interaction path. By precisely monitoring the student developer's current interaction state, tools can optimize suggestion relevance and minimize cognitive interruption. For example, if a student is identified as being in the "comment prompt" state, the tool can reduce the frequency of generating comment suggestions, minimizing resource waste while reducing interruptions for student developers.

Designers can optimize existing tools based on identified reasons for code modifications/rejections, as well as the challenges and expectations highlighted in the study. RQ4 and RQ5 provide detailed insights into these aspects, offering valuable implications for ACAT designers. Here, we discuss some interesting findings.

1) Enhance Learning Through Natural Language Interaction. Chatbots based on LLMs (e.g., ChatGPT) have significantly impacted SE [26, 54, 60]. Our study reveals that over half of the students desire stronger natural language interaction capabilities in their learning process. This suggests that purely code-focused tools may not adequately support students' learning needs. Incorporating more sophisticated natural language understanding could help students better articulate their programming concepts, receive clearer explanations, and bridge the gap between conceptual understanding and code implementation. This enhancement could particularly benefit novice programmers who are still developing their technical vocabulary.

2) Adapt Code Suggestions Ranking to Learning Progress. Nearly half of the students raised concerns regarding the relevance of the ranked code suggestions. This indicates the need for a learning-oriented approach to code recommendation ranking that considers students' current knowledge level, learning objectives, and previous interactions with the tool. Such adaptive suggestion systems could provide progressively challenging recommendations that align with students' learning trajectories.

3) Support Personalized Response of ACATs. Our study highlights the importance of personalization in ACATs. Participants express distinct preferences for aspects such as recommended code length, the number of suggestions provided, and coding style. This diversity emphasizes the importance of personalized learning experiences. Tools should accommodate different learning styles and paces, allowing students to customize their learning environment while maintaining pedagogical effectiveness. This flexibility could help students develop their unique programming approaches while meeting educational objectives.

4) Improve the UI Design of ACATs. Our findings underscore the critical role of UI design in ACATs. Despite the crucial role of code recommendation accuracy, many participants mention that UI issues negatively impact their interaction experience. For example, 19 participants reported the challenge of "*unreasonable way of displaying and switching candidate recommendations*" and 8 participants reported other UI issues. To enhance user satisfaction, designers should focus on improving the usability of ACAT interfaces. Improving interface usability is crucial for maintaining students' focus on learning programming concepts rather than struggling with tool operation. Enhanced interfaces should facilitate seamless access to learning resources, code examples, and explanations, supporting a more integrated learning experience.

5) Ensure Consistent Learning Environment. Five students emphasized the need for reliable tool access, noting how network dependencies can disrupt their learning process. Enabling offline functionality would ensure continuous learning opportunities regardless of network conditions. This is particularly important for maintaining consistent learning experiences and allowing students to practice programming in various environments. Additionally, offline capability addresses data privacy concerns, which can be particularly relevant in educational settings.

5.2 Implications for Students

Proactively contextualizing ACATs rather than passively awaiting recommendations. In RQ3, we observe some effective students' interactions with ACATs. Rather than waiting passively for ACAT to provide code completions, these participants continuously supplement contextual information by writing code comments, which is the basis for ACAT's recommendations. For example, before writing a function, users can use code comments to write what the function does, the parameters the function accepts (and their types), and the function's return value. In this way, the code recommended by ACAT has a higher probability of meeting students' needs and is more likely to be accepted. This active interaction with ACAT is worth learning by other ACAT users.

We observed a strong correlation between extensive code modifications and subsequent backtracking. This implies that students should develop a meta-cognitive awareness of

when to pursue integration of AI suggestions versus when to reject them early in the process, potentially saving considerable development time and cognitive effort. Furthermore, findings related to the "Adaptation" layer in RQ3 indicate that when students modified their previous code after accepting a code suggestion (to better integrate the suggestion with existing code), this behavior was almost invariably followed by backtracking. This reveals a key implication: If students make substantial modifications to their own code to accommodate AI-generated suggestions, they are highly likely to eventually discard these suggestions after multiple revision attempts. Consequently, CS students should develop a nuanced evaluation mechanism that carefully assesses the alignment between AI suggestions and their existing code architecture and programmatic intent before implementing substantial code transformations.

Additionally, our findings in RQ2 reveal that "slow accept" is a sophisticated learning strategy rather than a purely reactive adoption of AI suggestions. By manually typing suggested code segments, students demonstrate an engagement that goes beyond passive acceptance. This behavior reveals critical learning mechanisms. First, "slow accept" facilitates *deep cognitive processing*, enabling students to critically evaluate and selectively integrate recommendation components. This deliberate interaction promotes active learning, where students maintain agency over their coding process while leveraging AI-generated guidance as a cognitive scaffolding mechanism. Second, the behavior serves as a *meta-cognitive validation strategy*. By incrementally typing suggestions that align with their emerging problem-solving approach, students experience enhanced confidence and epistemological trust in their reasoning. Through these mechanisms, students transform AI suggestions from mere code recommendations into interactive learning artifacts. These insights imply that learners should view ACATs not as replacement tools, but as collaborative learning platforms. Educators and students alike should cultivate such interaction patterns that prioritize cognitive agency, critical evaluation, and active learning, thereby maximizing the pedagogical potential of AI-assisted coding environments.

5.3 Implications for SE Researchers

Our investigation illuminates the critical need for examining novel collaborative paradigms between students and ACATs in educational contexts. This "Student-ACAT" interaction model introduces multifaceted considerations including ACATs features, task complexity, and learner expertise levels that significantly influence learning outcomes. SE researchers should systematically explore these dimensions to develop comprehensive frameworks explaining how students and ACATs collaborate during programming skill acquisition. Such investigations could yield valuable insights for designing pedagogically effective collaboration practices that enhance coding education.

Our findings underscore **the importance of investigating how different combinations of interaction behaviors influence task completion and learning outcomes**. Future SE research should examine which interaction sequences (e.g., requesting explanations before accepting code, iteratively refining prompts, or combining verification with modification behaviors) lead to optimal knowledge construction and skill development. Understanding these interaction patterns could enable the design of adaptive systems that guide students toward more productive engagement strategies with AI tools, ultimately enhancing both immediate task performance and long-term learning benefits.

The data collected in our study—including interaction logs and screen recordings of CS students engaging with ACATs—constitutes a valuable resource for further research and replication efforts. **SE researchers can analyze code characteristics by comparing accepted versus rejected AI recommendations to improve algorithmic accuracy in educational contexts.** Additionally, these data enable detailed examination of students' interaction behavior patterns with ACATs, revealing how learners engage with AI suggestions and how these interactions shape their learning

trajectories. **Such insights can inform the development of more learner-centered ACATs specifically tailored to support programming skill development in educational settings.** For example, our focus on high-interaction users opens up new research questions regarding the full spectrum of student engagement. Future work could leverage our dataset to investigate the behaviors of students who interact minimally with ACATs. Understanding the reasons behind low engagement—whether it stems from distrust, frustration with irrelevant suggestions, or a conscious pedagogical strategy—is crucial for building more inclusive and universally effective tools. Such studies could uncover significant barriers to adoption and inform the design of scaffolding mechanisms that can better support students who are initially hesitant or skeptical of AI assistance, ensuring that the benefits of ACATs are accessible to all learners.

6 LIMITATIONS

6.1 Internal Validity

The tasks A&B under each task type might affect results of ACAT's impact on participants' self-perceived productivity due to differences in difficulty and workload. For ADS-tasks, we choose two Codeforces problems with similar difficulty and pass rates. For MSD and RTD tasks, we ensure consistency in difficulty and workload by requiring the same number of similar functions to be implemented.

The sequence of completing tasks A&B may also affect the results of participants' self-perceived productivity. To mitigate this, we provide participants with preparatory materials like half-completed code, JSON file tutorials, and pre-defined string handling functions for tasks A&B.

Participants' familiarity with ACATs may introduce bias. To mitigate this, tasks were allocated based on participants' self-reported familiarity levels from the pre-test survey. Although two participants (Stu18 in the `CG_RTD` group and Stu22 in the `TAB_RTD` group) did not meet our criterion of moderate proficiency (≥ 3), we argue that their limited familiarity had minimal impact for three reasons. First, as discussed in Section 3.1, the three tools share nearly identical core functionalities and user interfaces—particularly for inline code completion via ghost text accepted with the Tab key. Both participants reported high familiarity (≥ 4) with *GitHub Copilot*, which uses the same interaction paradigm, making their prior experience readily transferable. Second, all participants completed a structured onboarding session covering the assigned tool's features (Section 3.6), ensuring a consistent baseline of operational competence. Finally, interaction data show no evidence of performance issues related to unfamiliarity. For instance, in the `TAB_RTD` group, Stu22's suggestion acceptance rate (17.9%) is comparable to that of other participants (14.8–31.4%), while in the `CG_RTD` group, Stu18 achieved a higher rate (62.9%) than peers (40–48.7%). These values fall within a plausible range of user strategies rather than indicating operational difficulty. Thus, we believe that initial familiarity differences did not materially affect the validity of our findings.

Participants may misinterpret the phrasing of certain survey questions. To mitigate this, we conduct a pilot survey, emphasizing the clarity of the survey questions, and subsequently revise the survey based on their feedback.

Memory bias may threaten the internal validity, as survey questions require participants to recall their interactions with ACATs. To address this, we provide visual cues to assist participants in recalling their past experiences.

The risk of data leakage during our pre-study verification. We tested the experimental tasks with the ACATs to ensure they were not already part of the tools' training data and thus solvable via simple memorization. While this process could theoretically introduce the tasks into the models' training data, we consider the risk negligible for two reasons. First, the underlying LLMs are not updated in real time; incorporating new data into a foundational model involves lengthy cycles of

curation, training, and deployment. Second, the formal participant studies were conducted shortly after verification, making it highly unlikely that the task data could have been integrated into a publicly deployed model. Thus, the tasks remained novel to the ACATs during the study, ensuring that our results reflect genuine human-AI interaction rather than memorization.

6.2 External Validity

Only 27 participants took part in our experiments, which may limit the generalizability of the findings. Despite our efforts to include a diverse group, this sample size is comparable to related studies [1, 2, 4, 38]. These participants produced 31.8 hours of screen recording data and 225 minutes of audio recording data for interviews. The total annotation process took two people two months, which requires a significant amount of work and considerable manual effort. Future studies could benefit from an even larger sample size to further improve the robustness of the study.

We recognize the typical challenges to reliability and generalizability when analyzing limited qualitatively data [41]. We mitigate this threat by involving both authors in the coding process and including survey responses in our replication package [33].

Given the survey was conducted in January 2024, respondents' feedback reflects their ACAT interactions at that time point. Hence, some elements might not apply to subsequent tool iterations with varying functionalities and performances.

Our qualitative behavior model is derived from high-interaction users. The development of our five-layer interaction model was based on data from the most interactive participant within each experimental group. This strategy was chosen to ensure a rich dataset for identifying a comprehensive range of behaviors within the constraints of a qualitative analysis. However, we acknowledge this as a limitation, as the model may not fully generalize to students with low levels of ACAT interaction. The behavioral patterns of less engaged students, which might be influenced by distinct factors such as a lack of trust, difficulty in eliciting useful suggestions, or a preference for independent problem-solving, represent an important area for future investigation.

Our findings are primarily generalizable to ACATs with an inline completion paradigm. The three tools selected for this study, while highly popular and representative of the current generation of AI assistants [10, 36, 63, 64], all share a primary interaction model: inline code completion supplemented by conversational chat. Consequently, our five-layer interaction model is most accurately understood as describing the cognitive and behavioral processes involved in this specific, human-in-the-loop collaborative paradigm. We acknowledge that our model may not directly apply to other, emerging forms of AI-driven software development tools. For example, the interaction patterns with fully autonomous AI agents, which operate with minimal human intervention, would likely be fundamentally different. Therefore, the generalizability of our conclusions is intentionally scoped to the class of collaborative, completion-centric ACATs that currently dominate the software development landscape.

6.3 Construct Validity

The study's observational nature of screen recordings introduces a potential observer effect. The awareness of being recorded might have caused participants to behave more cautiously or deliberately than they would in a natural setting. We took several steps to mitigate this inherent threat. First, our methodology aligns with established practices in HCI and SE research, where screen recording is a standard and widely used method for studying developer-tool interactions, including those with ACATs [1, 4, 38, 57]. Second, we chose remote screen recording as a less intrusive alternative to in-person, over-the-shoulder observation, while being indispensable for capturing the fine-grained behavioral data (e.g., code modifications, backtracking, pauses) that are central to our research questions. Third, to minimize performance anxiety, we explicitly instructed

participants before the study that their performance would not be judged in any way and that their compensation was guaranteed regardless of task completion.

Finally and crucially, we did not analyze observational data in isolation. Our findings are grounded in a triangulation of screen recordings with post-task interview data, which allowed us to validate our interpretations of behaviors by understanding the participants' underlying intents and cognitive processes. This approach, as demonstrated in our analysis of the interaction model and behavioral paths (Section 4.3.1 and Section 4.3.2), strengthens the construct validity of our findings by confirming that observed patterns represent genuine user strategies rather than artifacts of the data collection method. While the observer effect can never be fully eliminated, our mixed-methods design provides robust evidence for our conclusions.

7 CONCLUSION

This paper takes an important step toward understanding how students interact with ACATs. It presents a comprehensive analysis of interaction frequency, behavioral patterns, decision-making factors, and users' challenges and expectations from students' perspective. Through a controlled study involving 27 CS students, we find that the acceptance rate of ACAT suggestions varies by task type, recommendation method, and recommendation content. We also find that ACATs enhance users' self-perceived productivity. To better capture the interaction process, we propose a five-layer hierarchical interaction behavior model that outlines distinct stages of user engagement. Our findings further reveal many intricate behavioral connections across different interaction stages. For example, actively prompted suggestions have lower acceptance rates than passively generated ones; "slowly accepted" suggestions exhibit minimal likelihood of revision or backtracking, and code modifications to accommodate suggestions often incur unnecessary effort. In addition, we identify 22 key factors that influence users' decisions and evaluations, as well as 11 challenges and 23 expectations regarding ACAT usage. Overall, this study offers important implications for ACAT designers, student users, and SE researchers, and advances the development of more effective and user-centered ACATs.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant 62572030), the Fundamental Research Funds for the Central Universities, the Young Elite Scientists Sponsorship Program of the Beijing High Innovation Plan, and the Exploratory Elective Projects of the State Key Laboratory of Complex and Critical Software Environments.

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*. 1–13.

[2] Owura Asare, Meiyappan Nagappan, and N. Asokan. 2024. A User-centered Security Evaluation of Copilot. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 158, 11 pages. doi:10.1145/3597503.3639154

[3] AC Bajpai, Irene M Calus, and JA Fairley. 1992. Descriptive statistical techniques. In *Methods of environmental data analysis*. Springer, 1–35.

[4] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.

[5] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming is hard-or at least it used to be: Educational opportunities and challenges of AI code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 500–506.

[6] A.F. Blackwell. 2002. First steps in programming: a rationale for attention investment models. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*. 2–10. doi:10.1109/HCC.2002.1046334

[7] Samuel Boguslawski, Rowan Deer, and Mark G Dawson. 2025. Programming education and learner motivation in the age of generative AI: student and educator perspectives. *Information and Learning Sciences* 126, 1/2 (2025), 91–109.

[8] Eugenio Brusa, Ambra Calà, Davide Ferretto, et al. 2018. *Systems engineering and its application to industrial product development*. Springer.

[9] John L. Campbell, Charles Quincy, Jordan Osserman, and Ove K. Pedersen. 2013. Coding In-depth Semistructured Interviews: Problems of Unitization and Intercoder Reliability and Agreement. *Sociological Methods & Research* 42, 3 (2013), 294–320. doi:10.1177/0049124113500475 arXiv:https://doi.org/10.1177/0049124113500475

[10] Ruijia Cheng, Ruotong Wang, Thomas Zimmermann, and Denae Ford. 2022. " It would work for me too": How Online Communities Shape Software Developers' Trust in AI-Powered Code Generation Tools. *arXiv preprint arXiv:2212.03491* (2022).

[11] Matteo Ciniselli, Luca Pascarella, Emad Aghajani, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. 2023. Source code recommender systems: The practitioners' perspective. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2161–2172.

[12] Codeforces. 2025. Codeforces. https://codeforces.com/. Accessed: Sep. 21, 2025.

[13] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. 2023. GitHub Copilot AI pair programmer: Asset or liability? *Journal of Systems and Software* 203 (2023), 111734.

[14] 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.

[15] 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.

[16] James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. 2022. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference*. 10–19.

[17] Nicole Forsgren, Margaret-Anne Storey, Chandra Madsen, Thomas Lippert, Iain Toft, and Utsav Vost. 2021. The SPACE of developer productivity: There's more to it than you think. *Queue* 19, 1 (2021), 20–48.

[18] Generator-code. [n. d.]. Generator-code. https://www.npmjs.com/package/generator-code. Accessed: Mar. 22, 2025.

[19] GitHub-Copilot-Chat. [n. d.]. GitHub-Copilot-Chat. https://docs.github.com/en/copilot/github-copilot-chat. Accessed: Mar. 22, 2025.

[20] Hacer Güner and Erkan Er. 2025. AI in the classroom: Exploring students' interaction with ChatGPT in programming learning. *Education and Information Technologies* (2025), 1–27.

[21] Judith A Holton. 2007. The coding process and its challenges. *The Sage handbook of grounded theory* 3 (2007), 265–289.

[22] Saki Imai. 2022. Is Github Copilot a substitute for human pair-programming? an empirical study. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 319–321.

[23] Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. 2022. Discovering the syntax and strategies of natural language programming with generative language models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–19.

[24] 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.

[25] Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara Jane Ericson, David Weintrop, and Tovi Grossman. 2023. 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*. 1–12.

[26] Jan Kocoń, Igor Cichecki, Oliwier Kaszyca, Mateusz Kochanek, Dominika Szydło, Joanna Baran, Julita Bielaniewicz, Marcin Gruza, Arkadiusz Janz, Kamil Kanclerz, et al. 2023. ChatGPT: Jack of all trades, master of none. *Information Fusion* 99 (2023), 101861.

[27] M Lee, A Blackwell, and A Sarkar. 2024. Predictability of identifier naming with Copilot: A case study for mixed-initiative programming tools. In *Proceedings of the 35th Annual Conference of the Psychology of Programming Interest Group (PPIG 2024)*.

[28] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. 2023. Comparing code explanations created by students and large language models. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. 124–130.

[29] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A Becker. 2023. Using large language models to enhance programming error messages. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 563–569.

[30] X Li, A Blackwell, and H Ge. 2024. Harnessing the Power of Artificial Intelligence in Mathematics Education: The Potential of Probabilistic Programming Languages in the Teaching and Learning of Bayesian Statistics. In *EDULEARN24 Proceedings*. IATED, 8074–8083.

[31] 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 *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 605–617.

[32] Xiao Long, Xin Tan, Yinghao Zhu, Jing Jiang, and Li Zhang. 2025. ccdc-plugin. https://marketplace.visualstudio.com/items?itemName=longlong-ccdc-plugin.ccdc-plugin. Accessed: Sep. 21, 2025.

[33] Xiao Long, Xin Tan, Yinghao Zhu, Jing Jiang, and Li Zhang. 2025. Replication Package. https://figshare.com/s/996a1d41bf886f955810 Accessed: Sep. 21, 2025.

[34] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. 2023. Experiences from using code explanations generated by large language models in a web software development e-book. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 931–937.

[35] Lauren E Margulieux, James Prather, Brent N Reeves, Brett A Becker, Gozde Cetin Uzun, Dastyni Loksa, Juho Leinonen, and Paul Denny. 2024. Self-Regulation, Self-Efficacy, and Fear of Failure Interactions with How Novices Use LLMs to Solve Programming Problems. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*. 276–282.

[36] Andrew M McNutt, Chenglong Wang, Robert A Deline, and Steven M Drucker. 2023. On the design of AI-powered code assistants for notebooks. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–16.

[37] Sharan B Merriam and Robin S Grenier. 2019. *Qualitative research in practice: Examples for discussion and analysis*. John Wiley & Sons.

[38] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2024. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–16.

[39] Satya Nadella. 2024. Microsoft Fiscal Year 2024 Second Quarter Earnings Conference Call. https://www.microsoft.com/en-us/investor/events/fy-2024/earnings-fy-2024-q2.aspx Accessed: Sep. 21, 2025.

[40] Nathalia Nascimento, Paulo Alencar, and Donald Cowan. 2023. Comparing software developers with ChatGPT: An empirical investigation. *arXiv preprint arXiv:2305.11837* (2023).

[41] Lorelli S Nowell, Jill M Norris, Deborah E White, and Nancy J Moules. 2017. Thematic analysis: Striving to meet the trustworthiness criteria. *International journal of qualitative methods* 16, 1 (2017), 1609406917733847.

[42] Arum Park and Taekyung Kim. 2025. Code suggestions and explanations in programming learning: Use of ChatGPT and performance. *The International Journal of Management Education* 23, 2 (2025), 101119.

[43] James Prather, Paul Denny, Juho Leinonen, Brett A Becker, Ibrahim Albluwi, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, Andrew Luxton-Reilly, et al. 2023. The robots are here: Navigating the generative AI revolution in computing education. In *Proceedings of the 2023 working group reports on innovation and technology in computer science education*. 108–159.

[44] 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 Transactions on Computer-Human Interaction* 31, 1 (2023), 1–31.

[45] 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.

[46] Ben Puryear and Gina Sprint. 2022. GitHub Copilot in the classroom: learning to code with AI assistance. *Journal of Computing Sciences in Colleges* 38, 1 (2022), 37–47.

[47] Christian Rahe and Walid Maalej. 2025. How Do Programming Students Use Generative AI? *arXiv preprint arXiv:2501.10091* (2025).

[48] Steven I Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D Weisz. 2023. The programmer's assistant: Conversational interaction with a large language model for software development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*. 491–514.

[49] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*. 27–43.

[50] Jaromir Savelka, Arav Agarwal, Christopher Bogart, Yifan Song, and Majd Sakr. 2023. Can generative pre-trained transformers (GPT) pass assessments in higher education programming courses?. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. 117–123.

[51] Forrest Shull, Janice Singer, and Dag IK Sjøberg. 2007. *Guide to advanced empirical software engineering*. Springer.

[52] Ilja Siroš, Dave Singelée, and Bart Preneel. 2024. GitHub Copilot: the perfect Code compLeeter? *arXiv preprint arXiv:2406.11326* (2024).

[53] Xin Tan, Xiao Long, Yinghao Zhu, Lin Shi, Xiaoli Lian, and Li Zhang. 2025. Revolutionizing Newcomers' Onboarding Process in OSS Communities: The Future AI Mentor. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE050 (June 2025), 23 pages. doi:10.1145/3715767

[54] Minaoar Hossain Tanzil, Junaed Younus Khan, and Gias Uddin. 2024. ChatGPT Incorrectness Detection in Software Reviews. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 964–964.

[55] 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 *Chi conference on human factors in computing systems extended abstracts*. 1–7.

[56] Chaozheng Wang, Junhao Hu, Cuiyun Gao, Yu Jin, Tao Xie, Hailiang Huang, Zhenyu Lei, and Yuetang Deng. 2023. How Practitioners Expect Code Completion?. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1294–1306.

[57] Wei Wang, Huilong Ning, Gaowei Zhang, Libo Liu, and Yi Wang. 2024. Rocks coding, not development: A human-centric, experimental evaluation of LLM-supported SE tasks. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 699–721.

[58] Michel Wermelinger. 2023. Using GitHub Copilot to solve simple programming problems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 172–178.

[59] David Wicks. 2017. The coding manual for qualitative researchers. *Qualitative research in organizations and management: an international journal* 12, 2 (2017), 169–170.

[60] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.

[61] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-IDE Code Generation from Natural Language: Promise and Challenges. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 29 (mar 2022), 47 pages. doi:10.1145/3487569

[62] Yeoman. [n. d.]. Yeoman. https://yeoman.io/. Accessed: Mar. 22, 2025.

[63] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the quality of GitHub copilot's code generation. In *Proceedings of the 18th international conference on predictive models and data analytics in software engineering*. 62–71.

[64] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large language models meet NL2Code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 7443–7464.

[65] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 21–29.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009