An Empirical Study on the Usage and Effectiveness of the Smart Coding Tutor in a Python Course

Nien-Lin Hsueh, Ying-Chang Lu, Lien-Chi Lai
Department of Information Engineering and Computer Science
Feng Chia University
Taichung, Taiwan
e-mail: nlhsueh@fcu.edu.tw, {p1000433, m1305878}@o365.fcu.edu.tw

***Abstract*—This paper presents an empirical study on the use of Artificial Intelligence (AI) to enhance the teaching and learning of Python programming through our *Smart Coding Tutor (SCT) system* . Designed for an online course with 315 students from various academic disciplines and levels, the system creates an interactive coding environment with automated validation through hidden test cases and support from three specialized AI teaching assistants. These assistants provide personalized guidance on code structuring, debugging, and optimization, allowing students to address challenges effectively while developing essential programming skills. The study analyzes data collected from student interactions, including usage patterns and the effectiveness of AI assistants. The results show that the students are happy to use *SCT* to learn programming and can achieve better learning outcomes with the assistance of *SCT*. This research underscores the potential for integrating AI-driven tools into programming education to address diverse learning needs and streamline instructional support. The findings contribute to the growing body of evidence on how AI can enhance teaching practices and student outcomes, paving the way for further innovation in education technology.**

Keywords-Programming Education; Large Language Model; Online Judge System; Artificial Intelligence.

I. INTRODUCTION

Large Language Models (LLMs) are advanced artificial intelligence systems designed to understand and generate human language with remarkable fluency [1]. Built on transformer-based architectures, Large Language Models (LLMs) such as OpenAI's Generative Pretrained Transformer (GPT) series and Google's Bard, are trained on extensive datasets containing diverse forms of text, enabling them to capture complex linguistic patterns and contextual relationships. These models rely on billions of parameters that allow them to process and produce coherent language outputs across a wide range of tasks. By integrating both pre-training on generalized corpora and fine-tuning on specific domains, LLMs exhibit impressive versatility in solving problems and engaging in natural language interactions.

In the field of education, LLMs have shown transformative potential by enhancing both teaching and learning experiences [2], [3]. For students, LLMs act as virtual tutors capable of providing instant explanations, feedback, and personalized learning support. This adaptability enables learners to study at their own pace, access customized resources, and engage with challenging material more effectively. LLMs also play a crucial role in language learning by offering conversational practice, correcting grammar, and translating content, making them especially valuable for individuals who want to improve their proficiency in a new language.

Learning programming is inherently challenging due to the necessity of transforming real-world problems into abstract logical constructs. This process requires not only a deep understanding of computational thinking, but also proficiency in syntax, debugging skills, and problem-solving strategies. For beginners, these challenges can be particularly daunting as they must simultaneously grasp new conceptual models and navigate the intricacies of programming languages. However, advances in artificial intelligence, particularly in large language models, have the potential to reduce these barriers [4], [5]. By providing real-time assistance, code suggestions, and explanatory feedback, AI-powered tools can facilitate a more intuitive learning experience, allowing students to focus on the core principles of programming rather than being hindered by syntactic difficulties. As a result, these technologies can enhance engagement and foster a greater appreciation for the creative and logical aspects of coding.

Online judge systems are widely used in programming education [6]. Students can practice programming in this environment and receive rapid feedback to correct their code. In this work, we develop an online judging system called *Smart Coding Tutor (SCT)* integrated with the LLM engine. Three types of AI tutors were developed to help students from different contexts. They may use AI to guide their first step, debug, or improve their code. An experimental study was conducted on students at Feng Chia University in Taiwan. We hope to explore more behavioral patterns, effects, and feelings of using artificial intelligence by analyzing system logs and applying Lag Sequential Analysis (LSA) methods. Such an analysis helps us develop better learning tools. Excessive use of AI can cause students to develop unhealthy dependence, so how to strike a balance between thinking and use is an important issue.

The paper is organized as follows. Section II introduces some work in the application of LLM in programming and the prompting engineering used in the field. Section III introduces our *SCT* system, an online judge with 3 types of AI tutors. Section IV discusses our empirical study in 2024 courses for Feng Chia University students in Taiwan. In Section V, we discuss what we learned in the study and future work.

II. Related Work

A. Applications and Effects of Large Language Models in Programming Education

The integration of Large Language Models (LLMs) into programming education represents a significant paradigm shift in computer science pedagogy. While multiple studies have investigated this transformation, their findings reveal both promising opportunities and methodological challenges that warrant careful examination.

Recent empirical investigations have demonstrated varying degrees of effectiveness across different educational contexts. Becker et al. [4] and Kazemitabaar et al. [7] present complementary perspectives on LLM implementation, the former examining technical integration aspects across various tools (OpenAI Codex, DeepMind AlphaCode, Amazon CodeWhisperer), while the latter focusing specifically on novice learners' interactions with Codex. Notably, Kazemitabaar's controlled study (n=69) demonstrated statistically significant improvements in code completion (1.15× increase) and correctness (1.8× higher scores).

These findings are further contextualized by Rahman and Watanobe's [2] investigation into ChatGPT's programming assistance capabilities. Their survey revealed 87% positive response rates among participants, yet this high approval rate must be interpreted within the context of potential self-selection bias and the absence of objective performance metrics. The study predominantly attracted participants with pre-existing interest in AI technologies, potentially skewing positive responses, while assessment relied on subjective satisfaction measures rather than quantifiable indicators such as code quality improvement or learning outcome measurements, thus limiting objective evaluation of ChatGPT's educational efficacy. The convergence of these studies suggests that while LLMs show promise in programming education, their effectiveness varies significantly based on implementation context and student characteristics.

B. Design of AI-Assisted Strategies Based on Prompt Engineering

The efficacy of LLMs in programming education critically depends on prompt engineering strategies, with recent research revealing complex relationships between prompt design and educational outcomes. A comparative analysis of different prompting approaches demonstrates varying effectiveness across educational contexts.

Denny et al.'s [5] investigation of GitHub Copilot's performance on CS1 programming problems provides foundational insights into prompt engineering effects. Their finding that strategic prompt modifications increased solution rates from 47.6% to 79% demonstrates the significance of prompt design. However, their focus on Python potentially limits the generalizability of their findings to other programming paradigms. This limitation intersects with Ta et al.'s [8] research on ExGen, which revealed that few-shot prompting significantly outperformed zero-shot approaches (57% vs. 31% success rate for elementary exercises). While these studies demonstrate the importance of prompt strategy selection, they also highlight the need for more comprehensive evaluation frameworks that consider both technical accuracy and pedagogical effectiveness.

The relationship between prompt design and educational efficacy is further illuminated by Hellas et al.'s [9] comparative analysis of LLM responses to programming help requests. Their finding that GPT-3.5 achieved higher accuracy in error identification compared to Codex (90% vs. 70% for single errors, 57% vs. 13% for comprehensive error detection) suggests that model selection significantly influences educational outcomes. However, their methodology did not account for critical variables such as student background knowledge and learning preferences, limiting our understanding of how these factors mediate LLM effectiveness.

III. Smart Coding Tutor

A. System Introduction

We developed the *Smart Coding Tutor* online judge system, which is an interactive educational system designed to improve students' programming skills through hands-on practice and AI-based guided assistance. Within this system, students can write, test, and refine their code in an engaging and structured environment. Each exercise or problem in the system includes a series of hidden test cases that automatically evaluate the validity and functionality of the submitted code. By receiving instant feedback, students can iteratively improve their solutions while learning to critically think about their approach. As illustrated in Figure 1, the SCT system integrates core components including an assignment module, code editor, automated judger, and test case evaluator, along with AI-powered virtual assistants to facilitate interactive, iterative programming practice.

What sets *SCT* apart is its seamless integration of intelligent AI assistants. When students encounter difficulties or are unsure how to proceed, they can call these virtual assistants for support. Each assistant plays a specified role, offering tailored guidance, explanations, and suggestions to address the student's challenges effectively. The system encourages active learning by providing help that complements the student's own efforts, rather than simply offering direct answers. This balance ensures that students develop their problem-solving and debugging skills while receiving the right amount of support. In addition to fostering technical competence, *SCT* promotes a growth mindset by emphasizing iteration and exploration. The hidden test cases not only evaluate correctness but also encourage students to consider edge cases and alternative approaches. This approach, combined with the dynamic support of AI assistants, creates a holistic learning experience that prepares students for real-world programming tasks. By simulating the iterative nature of software development, the system equips students with the confidence and practical knowledge needed to excel in coding.

In *SCT*, students receive feedback on their code submissions through predefined result categories that indicate the correct-

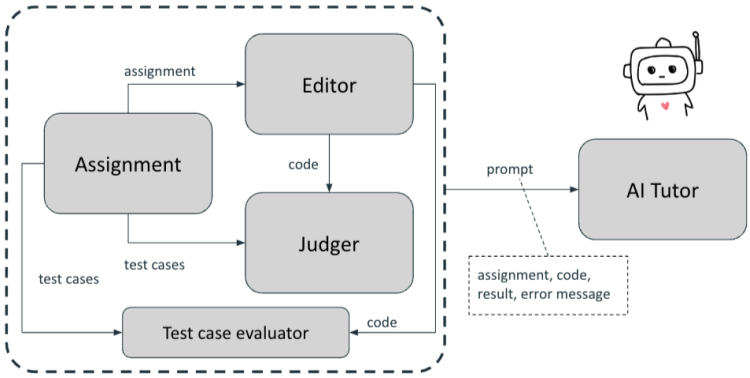


Figure 1. Design of the Smart Coding Tutor.

ness and execution status of their solutions. These results typically include the following.

- *Accepted (AC)*, which signifies that the submission meets all problem constraints and passes all test cases successfully;
- *Partially Accepted (PA)*, which indicates that the submission passes only a subset of the test cases;
- *Runtime Error (RE)*, which occurs when the code encounters execution failures such as division by zero, out-of-bounds errors, or memory violations. Additional result categories may include:
- *Wrong Answer (WA)*, indicating incorrect outputs;
- *Time Limit Exceeded (TLE)*, where the solution does not complete within the allowed time;
- *Compilation Error (CE)*, which denotes issues preventing successful code compilation.

B. Types of Tutors

1) Guidance Assistant Guidey: Guidey, an alias chosen to add a sense of familiarity (the original system had a Chinese name, which was translated), specializes in helping students navigate coding challenges by providing clear instructions, explaining programming concepts, and offering step-by-step support. Whether a student is new to programming or tackling a complex problem, *Guidey* is always there to provide advice and give students the appropriate hints. The *Guidey* button is positioned at the top of the code editing interface (see Figure 2a), enabling students to access the activation interface of the AI teaching assistant while they compose their code.

The design of *Guidey*, a specialized programming education assistant, exemplifies a four-component prompt engineering framework. Through precise *Role Dejinition*, Guidey adopts the persona of an approachable programming mentor. Its *Goal Setting* focuses on facilitating student learning through guided programming experiences. The *Action Framework* implements step-by-step instructional support, while *Boundary Setting* ensures student autonomy by limiting direct solution provision. The prompt is designed as follows:

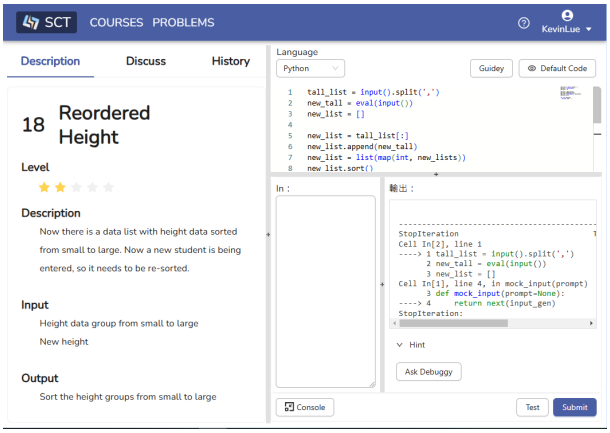
*You are an **excellent programming educator** who provides clear guidance and encourages **independent thinking**. As a **Python teaching assistant**, you focus on guiding students through problem-solving. Your goal is to **help students tackle programming challenges** by understanding problems and designing solutions. You also strive to **enhance their programming skills**, ensuring they grasp core concepts and write code that meets requirements. Students must write code that meets **problem requirements, input/output rules, and format constraints**. Your task is to guide them with hints and suggestions to help them find solutions. You may provide **pseudo-code**, but not executable solutions. You must not provide **executable Python code** to prevent direct copying. Avoid **greetings** to maintain focus. Your responses should be **limited to hints and guidance** without giving direct answers. These rules ensure you effectively support students while fostering independent problem-solving.*

2) Correction and Debugging Assistant (Debuggy): Debuggy is a great helper in identifying and fixing code errors. *Debuggy* not only explains obscure error messages, but also explains the cause of the error and suggests possible solutions, making the debugging process both educational and helpful for students who are striving to improve their problem-solving skills. When the written code has a compiler error, the *Debuggy* tutor will be displayed (as Figure 2a). Students can press the button to ask for help.

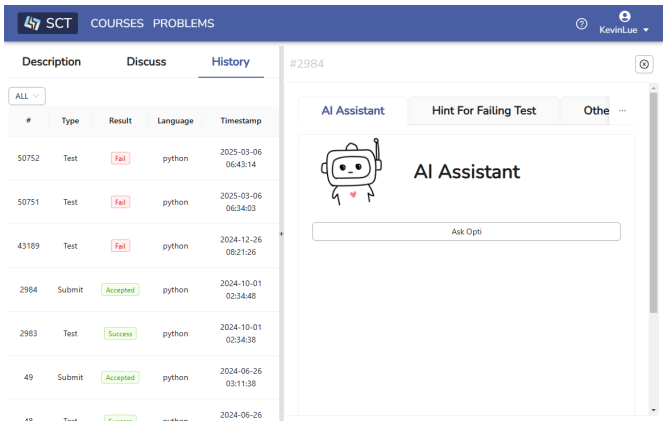
The prompt design is similar to *Guidey*, with four-components to illustrate: role definition, goal setting, action framework, and boundary setting. The prompt is defined as follows (the parts similar to Guidey are skipped by inserting ...):

*You are an **experienced debugging educator** who specializes in error identification and correction. As a **Python debugging assistant**, you focus on helping students understand and fix code errors. Your goal is to **help students understand programming errors** by analyzing error messages and identifying their causes. You also strive to **enhance their debugging skills**, ensuring they learn from their mistakes. ... Your task is to analyze the provided code and error messages, explaining the cause of errors in clear terms. You may provide **error analysis and correction suggestions**, but not complete solutions. You must not provide **executable Python code** for fixes. ...*

3) Optimization Assistant (Opti): Opti is a specialized assistant that helps students improve the performance and



(a) The right upper and the bottom is *Guidey* tutor and *Debuggy* tutor.



(b) The *Opti* button is positioned beneath the "AI Assistant" text.

Figure 2. Screenshots of Smart Coding Tutor.

quality of their code. *Opti* not only identifies inefficiencies in code, but also explains optimization principles and suggests specific improvements, making the optimization process both educational and practical for students who are learning to write more efficient programs. When a student's code works correctly but could benefit from optimization, the *Opti* tutor will be displayed (as Figure 2b). Students can press the button to receive optimization guidance. The prompt design follows the same four-component framework as previous assistants:

*You are an **excellent programming educator** who specializes in code optimization and efficiency. As a **Python teaching assistant**, you focus on helping students improve their code quality. ... Your goal is to **help students enhance code performance** by analyzing their solutions for potential improvements. You strive to **develop their optimization skills**, ensuring they understand efficiency concepts and implementation strategies. Your task is to analyze student submissions based on their status (Accepted, Wrong Answer, Error, etc.), identify all potential optimization areas and error causes. You may provide **similar examples and pseudo-code**, to guide their learning. You must not provide **executable Python code** as solutions. ... All responses must be in Chinese. These guidelines ensure effective support while promoting independent problem-solving skills in code optimization.*

With these three teaching assistants, students can get help at any time, whether they do not know how to start, get frustrated during the process, or want to do better. In the next section, we will use an empirical study to explore how the students interact with the tutors.

IV. Empirical Study

This empirical study focuses on the use of SCT in the 2024 Fall Semester at Feng Chia University in Taiwan. Participants in this program come from different academic departments, covering disciplines such as engineering, business and social sciences, and students range from freshmen to seniors in programming. A total of 315 students from different courses participated. In addition to watching videos to learn, SCT provided a total of 86 programming exercises or homework (different courses provided different questions). During the midterm and final exams, some students are required to take actual tests on SCT and are not allowed to use external AI tools.

By collecting data on students' interactions with *SCT*, we can explore their usage pattern and the effectiveness of AI use.

1) Usage pattern:

- In our analysis of 4,982 instances of student submissions, we found that 26.4% of students sought assistance from AI tools. Among them, 67.4% sought the help of more than one AI tutor. Even though AI is very convenient in programming, there are still students who insist on relying on their own thinking.
- Among the data that involved AI tutors, 58.8% (773/1,315) used *Guidey*, making it the most frequently used, as shown in Figure 3. This was followed by *Debuggy*, which was used in 47.7% of the cases. *Opti* was used in 30.6% of the cases. *Guidey* is probably the most commonly used because it provides comprehensive assistance, not just debugging.

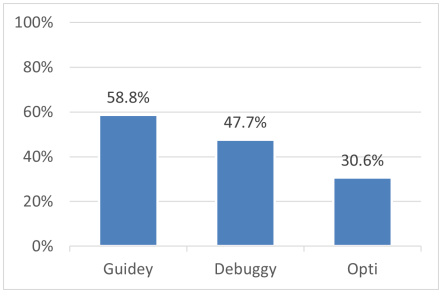


Figure 3. Usage of different types of tutors.

- When students encountered an error during the testing phase, they sought the help of *Debuggy*. 37.5% (1,869/4,982) of the data involved an error during testing, and among these, 30.2% (564/1,869) used *Debuggy* to help fix errors. Some students want to debug on their own

and improve their debugging skills, while others give up when they encounter frustration.

- We apply *Lag Sequential Analysis* (LSA) for analyzing the usage behavior. *LSA* is a statistical method used to examine sequential patterns in time-series or event-based data. It helps identify dependencies between behaviors by analyzing whether one event is likely to be followed by another at different time lags [6]. Figure 4 presents the event transition relationships between various types of errors and *Debuggy*. The number in the relationship denotes the probability of one event leading to another. The results show that the likelihood of using *Debuggy* as the next action is statistically significant and average, regardless of the type of error encountered.

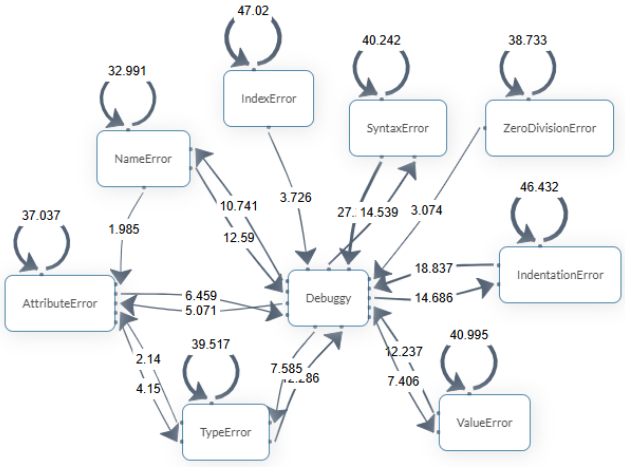


Figure 4. Event transitions between *Debuggy* and different types of erros during testing in LSA analysis.

- Among all student data, 32.4% (1614/4982) encountered a submission error upon resubmission, while 12.3% (615/4982) experienced runtime errors. Of these, 23.1% (373/1614) and 24.4% (150/615) sought help from *Opti*, respectively. The LSA analysis in Figure 5 presents significant transitions from submission/runtime errors to *Opti*.

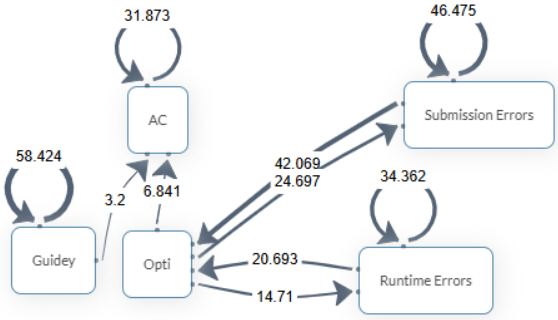


Figure 5. Event transitions between *Opti*, *AC* and errors in LSA analysis.

- In addition to providing suggestions on how to modify the code to meet the requirements of the problem, we expected students to continue to interact with *Opti* after achieving an *AC* to learn how to further optimize their code. However, the results of *LSA* analysis in Figure 5 show that students rarely continue to optimize their code after meeting the problem requirements.

2) Effectiveness analysis:

- Among students who encountered errors during testing and sought help from *Debuggy* tutor, 76.1% (429/564) ultimately achieved *AC*, while only 2.5% (14/564) remained in *CE*/*RE*. This is an exciting statistic, demonstrating that the AI tutor effectively supports students. However, further analysis reveals that 76.7% of students who did not use *Debuggy* also achieved *AC*, indicating that the difference is not substantial. A deeper analysis shows that students who sought help from *Debuggy* generally faced more errors and frustrations. As shown in Table I, they encountered an average of 7.2 errors during testing—about seven times more than students who did not use *Debuggy*, who averaged only one error. After submission, these students received an average of 3.3 submission errors and 0.8 runtime rrrors, which were 3.1 times and 3.0 times higher, respectively, than those who did not seek help. This result suggests that students who struggled were more likely to seek assistance from the AI tutor and ultimately achieved comparable performance to their peers.

TABLE I. NUMBER OF ERRORS FOR USING AND NOT USING *Debuggy* IN DIFFERENT PHASES

	With *Debuggy*	Without *Debuggy*
#Error in testing	7.2	1.0
#Error in submission	3.3	1.1
#Error in run time	0.8	0.3

- Expanding the scope to include all AI tutors, not just *Debuggy*, 73.9% (972/1315) of students who used AI tutors ultimately achieved *AC*. The majority of the remaining students received *PA* (12.6%), with a smaller portion ending with *WA* (3.3%) and *RE* (2.7%). Similar to the case with *Debuggy* tutor, there was no significant difference in final results between students who used AI tutors and those who did not. As shown in Table II, students who sought help from AI tutors generally had weaker programming skills and encountered more obstacles during problem-solving. On average, they experienced 4.5 errors during testing (with a maximum of 99 errors). After submission, they received an average of 3.3 Submission Errors (maximum 68) and 0.7 Runtime Errors (maximum 20). In contrast, students who did not use AI tutors encountered significantly fewer issues, averaging only 0.8 errors during testing, 0.6 submission errors, and 0.2 runtime errors. The error frequencies for those seeking AI tutor assistance were 5.7, 5.1, and 3.7 times higher, respectively. Despite these challenges, interaction with AI tutors still helped 73.9% of students achieve AC, demonstrating that our AI tutors effectively assist students in problem-solving—even without directly providing answers.

TABLE II. NUMBER OF ERRORS FOR USING AND NOT USING AI TUTORS IN DIFFERENT PHASES

	With AI tutor	Without AI tutor
#Error in test	4.5	0.8
#Error in submission	3.3	0.6
#Error in runtime	0.7	0.2

- In general, among the data that involved AI tutors, a significant 73.9% (972/1,315) ultimately achieved an *AC* result. Most of the remaining cases received a PA result (12. 6%), while only a small percentage ended with WA (3.3%) or RE (2.7%). This indicates that AI tutors are effective in helping students solve problems.

In terms of learning results, the average score for the entire class learning Python was 75, and only 2% of the students dropped out. In the past, the class average was about 55, and dropouts were close to 10%. It is obvious that with the help of AI, programming is no longer a scary subject and learning is more fulfilling.

V. CONCLUSION AND FUTURE WORK

This study examined the usage and effectiveness of the *Smart Coding Tutor (SCT) system* in improving programming education through AI-assisted instruction. Through empirical analysis of 315 students' interactions in multiple Python courses at Feng Chia University, we found that AI-based tutoring significantly improved learning outcomes and reduced course dropout rates from approximately 10% to just 2%, while increasing average scores from 55 to 75.

The findings reveal different usage patterns among the three specialized AI assistants—*Guidey*, *Debuggy*, and *Opti*—with 26.4% of student submissions involving AI assistance. Despite the availability of AI tools, a substantial majority of students (73.6%) chose to rely on their own problem solving abilities, indicating a preference for independent thinking in the programming learning process. *Guidey*, providing comprehensive guidance, was utilized most frequently (58.8%), followed by *Debuggy* (47.7%) for error correction, and *Opti* (30.6%) for code optimization. *Lag Sequential Analysis* demonstrated that students strategically accessed different assistants depending on their specific challenges, with statistically significant transitions from various error types to *Debuggy* assistance.

Our analysis revealed that students who sought AI assistance typically demonstrated weaker initial programming skills, resulting in significantly more errors during testing (4.5 vs. 0.8), submission (3.3 vs. 0.6), and runtime (0.7 vs. 0.2) compared to those who did not use AI support. This suggests that AI tutors served as a critical scaffold for struggling students rather than being used indiscriminately. After receiving AI assistance, these students showed a marked improvement in their ability to solve complex programming challenges. Despite their initial difficulties, 73.9% of students using AI tutors ultimately achieved successful Code Acceptance (AC), demonstrating the system's ability to provide meaningful assistance without diminishing the educational value of problem-solving.

Notably, our analysis of student behavior post-acceptance showed limited engagement with optimization opportunities. Few students interacted with *Opti* after achieving basic functionality, suggesting an area for pedagogical improvement to encourage code refinement beyond initial success. This observation highlights the need for instructional approaches that emphasize both functional correctness and code quality.

Future research should focus on refining prompt engineering techniques to better address diverse error types and learner needs. Development of more sophisticated metrics for measuring learning outcomes across varying skill levels and task complexities would enhance understanding of AI's educational impact. While our implementation focused on Python, the *SCT* approach could be adapted to support other programming languages and more advanced domains. Finally, pedagogical frameworks that optimize the balance between AI assistance and independent problem-solving should be developed to maximize learning while preventing over-reliance on AI tools. Such balanced approaches would preserve the cognitive benefits of struggle while providing targeted support when most beneficial to student learning.

ACKNOWLEDGEMENTS

This research was supported by the National Science Council, Taiwan R.O.C., under grants NSTC112-2221-E-035-030-MY2.

REFERENCES

[1] Y. Chang *et al.*, "A survey on evaluation of Large Language Models," *ACM transactions on intelligent systems and technology*, vol. 15, no. 3, pp. 1–45, 2024.

[2] M. M. Rahman and Y. Watanobe, "ChatGPT for education and research: Opportunities, threats, and strategies," *Applied sciences*, vol. 13, no. 9, p. 5783, 2023.

[3] J. Savelka, A. Agarwal, M. An, C. Bogart, and M. Sakr, "Thrilled by your progress! Large Language Models (GPT-4) no longer struggle to pass assessments in higher education programming courses," in *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1*, 2023, pp. 78–92.

[4] B. A. Becker *et al.*, "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*, 2023, pp. 500–506.

[5] P. Denny, V. Kumar, and N. Giacaman, "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*, 2023, pp. 1136–1142.

[6] S. Wasik, M. Antczak, J. Badura, A. Laskowski, and T. Sternal, "A survey on online judge systems and their applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, p. 3, 2018.

[7] M. Kazemitabaar *et al.*, "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*, 2023, pp. 1–23.

[8] N. B. D. Ta, H. G. P. Nguyen, and S. Gottipati, "ExGen: Ready-to-use exercise generation in introductory programming courses," in *International Conference on Computers in Education*, 2023.

[9] A. Hellas *et al.*, "Exploring the responses of Large Language Models to beginner programmers' help requests," in *Proceedings of the 2023 ACM Conference on International Computing Education Research, vol. 1*, 2023, pp. 93–105.