Designing for Novice Debuggers: A Pilot Study on an AI-Assisted Debugging Tool

Oka Kurniawan
oka_kurniawan@sutd.edu.sg
Singapore University of Technology and Design
Singapore, Singapore

Erick Chandra
erick_chandra@sutd.edu.sg
Singapore University of Technology and Design
Singapore, Singapore

Christopher M. Poskitt
cposkitt@smu.edu.sg
Singapore Management University
Singapore, Singapore

Yannic Noller
yannic.noller@acm.org
Ruhr University Bochum
Bochum, Germany

Kenny Tsu Wei Choo
kennytwchoo@gmail.com
Singapore University of Technology and Design
Singapore, Singapore

Cyrille Jegourel
cyrille_jegourel@sutd.edu.sg
Singapore University of Technology and Design
Singapore, Singapore

Abstract

Debugging is a fundamental skill that novice programmers must develop. Numerous tools have been created to assist novice programmers in this process. Recently, large language models (LLMs) have been integrated with automated program repair techniques to generate fixes for students' buggy code. However, many of these tools foster an over-reliance on AI and do not actively engage students in the debugging process. In this work, we aim to design an intuitive debugging assistant, CodeHinter, that combines traditional debugging tools with LLM-based techniques to help novice debuggers fix semantic errors while promoting active engagement in the debugging process. We present findings from our second design iteration, which we tested with a group of undergraduate students. Our results indicate that the students found the tool highly effective in resolving semantic errors and significantly easier to use than the first version. Consistent with our previous study, error localization was the most valuable feature. Finally, we conclude that any AI-assisted debugging approach should be personalized based on user profiles to optimize their interactions with the tool.

CCS Concepts

• **Applied computing** → *Education*; • **Software and its engineering** → **Software testing and debugging**; • **Social and professional topics** → **Software engineering education**.

Keywords

Assisted debugging, programming education, intelligent tutoring systems, large language models, interactive debugging, novice programmers, AI assistants, AI tutoring, design guidelines.

ACM Reference Format:
Oka Kurniawan, Erick Chandra, Christopher M. Poskitt, Yannic Noller, Kenny Tsu Wei Choo, and Cyrille Jegourel. 2025. Designing for Novice Debuggers: A Pilot Study on an AI-Assisted Debugging Tool. In *25th Koli Calling International Conference on Computing Education Research (Koli Calling '25), November 11–16, 2025, Koli, Finland.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3769994.3769997

ACM ISBN 979-8-4007-1599-0/25/11
https://doi.org/10.1145/3769994.3769997

1 Introduction

The first hurdle novice programmers encounter when writing code is ensuring that it is free of syntax errors [24]. Although most compilers and interpreters generate error messages that provide information about these errors, many novice programmers struggle to interpret and fix them [6]. To address these challenges, numerous tools and approaches have been developed to help novice programmers read, interpret, and resolve error messages effectively [3, 16]. However, many of these tools primarily focus on improving error messages rather than guiding students through the debugging process [1].

Recent advances in AI-assisted programming have introduced large language model (LLM) tools as potential debugging assistants, capable of automatically identifying and fixing errors. LLMs have already shown strong accuracy in correcting syntax errors, opening up new possibilities for automated debugging support [10, 29]. Moreover, LLM-based tools such as ChatGPT and GitHub Copilot can provide explanations of syntax errors along with code examples that help learners understand and correct their mistakes [8, 19].

While LLM-based tools provide valuable assistance in fixing syntax errors, their ability to help novice programmers identify, understand, and resolve semantic errors remains limited. First, studies have shown that while LLM-based tools can resolve some semantic errors, their accuracy varies depending on the complexity of the problem [29]. Second, the generated solutions may differ significantly from students' original buggy code, making it difficult for novice programmers to understand how to modify their code accordingly [26]. Third, LLM-based tools may also introduce new errors, which novice programmers often struggle to recognize or correct, leading to a negative learning experience.

Finally, and most importantly, many LLM-based tools generate complete solutions to semantic errors rather than guiding users through the step-by-step debugging process that is essential for learning. As a result, novice programmers may develop an over-reliance on AI tools, which can hinder their ability to debug independently [5]. This dependence not only affects the accuracy of solutions but also impairs the ability of students to develop critical problem-solving skills. Without structured guidance, students may adopt AI-generated solutions without fully understanding how to troubleshoot and resolve issues on their own. Proficiency in testing, debugging, and fixing code is a fundamental skill for computer science (CS) graduates, and its importance is even greater in the era of AI-assisted programming, where AI tools enhance productivity but cannot substitute for a programmer's ability to independently reason through errors.

Our work introduces CodeHinter, an interactive debugging tool that goes beyond automatic error correction to actively guide students through diagnosing and fixing their own code. By integrating fault localization techniques with LLM-powered hints directly into the IDE through a simple one-click interface, CodeHinter promotes step-by-step problem solving, deepens students' understanding of errors, and reduces reliance on fully automated solutions. We evaluate its usability through a user study, highlighting how structured, interactive guidance can strengthen debugging skills and improve programming education.

2 Related Work

In this section, we review related work on how LLMs assist novice programmers in fixing errors. We examine various tools designed to help them resolve both syntax and semantic errors. Given the extensive literature on this topic, we focus on a selection of representative studies most relevant to our work.

Leinonen et al. utilized Codex, a model based on GPT-3, to improve the clarity of programming error messages [16]. Programming error messages are often difficult for novice programmers to interpret [22]. Their work aimed to enhance these messages by providing explanations of the errors along with suggested fixes. They found that LLMs can generate helpful, novice-friendly explanations, improving students' comprehension and their ability to correct errors. While the explanations generated by Codex were generally comprehensible, the suggested fixes were correct only 33% of the time. To improve the accuracy of generated fixes, Phung et al. introduced a novel runtime validation mechanism to assess whether the feedback provided by the LLM in the initial stage was suitable for students [21]. Their approach involved iteratively querying the LLM to generate fixes for the buggy program while leveraging feedback from previous LLM iterations. If the number of syntactically correct programs exceeded a predefined threshold, the feedback was deemed acceptable. This method enabled them to achieve high precision in identifying and correcting syntax errors. However, their study was limited to syntax errors, and its effectiveness in addressing semantic errors has not been explored.

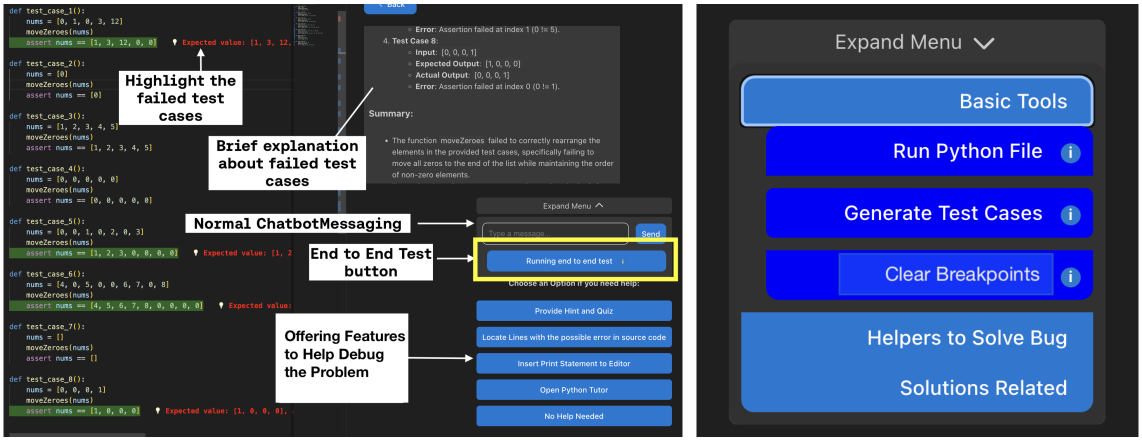
A widely explored approach for addressing semantic errors is automated program repair (APR) [27]. LLMs have been applied to APR through fine-tuning, few-shot learning and zero-shot learning [28]. One notable example is PyDex, which employs few-shot learning to repair both syntactic and semantic bugs in introductory Python assignments [26]. PyDex utilizes the structure of a student's buggy program as input to the LLM, resulting in repairs that require fewer edits. When compared to other APR tools, PyDex achieved a repair rate of 96.5%. Additionally, the average token edit distance for PyDex-generated patches was lower than that of competing tools. However, PyDex provides fixes without student intervention, which may hinder the development of debugging skills, as students receive corrections without actively engaging in the debugging process. CodeAid, on the other hand, provides helpful and technically accurate responses without revealing full code solutions to students [12]. This tool allows students to ask general programming questions, seek explanation for their code, request help in fixing their code, and receive help with writing new code. CodeAid employs an LLM in a two-step process to assist students in fixing their code. First, it generates a corrected version of the code based on the provided description and buggy implementation. Second, it explains the modifications using bullet points, detailing what was changed and why. However, the student interface only displays the bullet-point explanations rather than the corrected code itself, ensuring that solutions are not directly provided.

Although CodeAid does not directly provide solutions, it offers step-by-step guidance on necessary changes, potentially reducing the cognitive effort required for debugging. Carver and Risinger developed a traditional framework for training students in debugging [4]. The process begins with testing a program, followed by answering a sequence of diagnostic questions, including "what is the problem?" and "what type of bug could cause the problem?". Katz and Anderson observed that programmers often employ backward reasoning when debugging their own code, typically starting by examining the output [11]. Additionally, research indicates that the skills required for fixing errors are not necessarily related to the methods used to identify and locate them. Studies show that for most students, the main difficulty in debugging lies not in repairing errors, but in earlier troubleshooting stages, such as understanding the system, testing, and locating errors within it [17]. In fact, a multi-institutional study of novice debuggers by Fitzgerald et al. found that once students successfully identify and locate bugs, they are generally able to fix them [7].

3 Design of CodeHinter

Our tool, CodeHinter, shown in Figure 1, is designed to help students debug and fix their code. It evolved from an earlier version of the tool called SID (for Simulated Interactive Debugging) [18], which allowed users to run a test file for their Python programs. If any of the tests failed, SID automatically inserted breakpoints in the IDE. To achieve this functionality, the tool was integrated as an extension in Visual Studio Code (VS Code). An initial study on SID provided insights that informed further design improvements.

This paper presents the current version of CodeHinter, which continues to be developed as an extension within Visual Studio Code, following positive feedback from students. The key improvement over SID is a closer alignment with the debugging process to actively engage students in critical thinking. In particular, CodeHinter includes the following new features: (1) a single 'End-to-End Test' button that allows students to run and test their code; (2) an LLM-powered system that generates hints and quizzes to prompt users to analyze errors and explore possible fixes; (3) line highlighting and a code difference interface to help students identify and modify relevant code sections; (4) a spectrum-based fault localization tool to pinpoint errors; and (5) an LLM-generated pseudo-code feature to help students understand the given problem.



```
def test_case_1():
    nums = [0, 1, 0, 3, 12]
    moveZeroes(nums)
    assert nums == [1, 3, 12, 0, 0]    Expected value: [1, 3, 12,

def test_case_2():
    nums = [0]
    moveZeroes(nums)
    assert nums == [0]

def test_case_3():
    nums = [1, 2, 3, 4, 5]
    moveZeroes(nums)
    assert nums == [1, 2, 3, 4, 5]

def test_case_4():
    nums = [0, 0, 0, 0, 0]
    moveZeroes(nums)
    assert nums == [0, 0, 0, 0, 0]

def test_case_5():
    nums = [0, 0, 1, 0, 2, 0, 3]
    moveZeroes(nums)
    assert nums == [1, 2, 3, 0, 0, 0, 0]    Expected value: [1, 2

def test_case_6():
    nums = [4, 0, 5, 0, 0, 6, 7, 0, 8]
    moveZeroes(nums)
    assert nums == [4, 5, 6, 7, 8, 0, 0, 0, 0]    Expected value:

def test_case_7():
    nums = []
    moveZeroes(nums)
    assert nums == []

def test_case_8():
    nums = [0, 0, 0, 1]
    moveZeroes(nums)
    assert nums == [1, 0, 0, 0]    Expected value: [1, 0, 0, 0],
```

- Error: Assertion failed at index 1 (0 != 5).

4. **Test Case 8**:
   - **Input**: [0, 0, 0, 1]
   - **Expected Output**: [1, 0, 0, 0]
   - **Actual Output**: [0, 0, 0, 1]
   - **Error**: Assertion failed at index 0 (0 != 1).

Summary:

- The function moveZeroes failed to correctly rearrange the elements in the provided test cases, specifically failing to move all zeros to the end of the list while maintaining the order of non-zero elements.

Figure 1: *(Left)* 'End-to-End Test' feature, the main feature of CodeHinter. The screenshot shows the state when users encounter failed test cases (corresponding to * in Figure 2), highlighting the expected value and actual output in the text editor while providing a brief explanation in the chatbot. *(Right)* CodeHinter 'Expand Menu', where users can access other features.

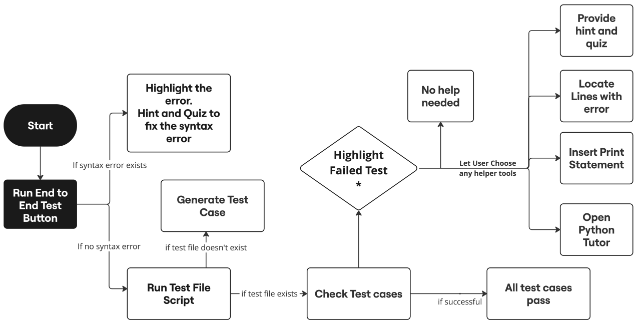


Figure 2: Flow of user interactions with our tool. The process begins with users pressing the 'End-to-End Test' button, which checks for syntax and semantic errors. Users are then guided to complete the missing debugging steps. If test cases fail, the tool provides options to help users resolve the bugs. The asterisk indicates the state of the screenshot in Figure 1.

Our tool currently supports Python, as it is one of the most widely used languages in introductory programming courses. Additionally, Python has several mature libraries for testing and fault localization that can be easily integrated. Our tool utilizes FauxPy [23] for spectrum-based fault localization, which is also part of the PyTest testing framework. The LLM powering our tool is OpenAI's GPT-4o, which offers high accuracy and significantly improved speed and cost-effectiveness compared to reasoning models like o1-mini and o1, which are better suited for more complex problems. GPT-4o is particularly effective for debugging and addressing novice programming issues [20].

One of the key features of the extension is the 'End-to-End Test' button. This button is designed to align with the standard debugging framework, incorporating backward reasoning. Figure 2 shows how the user is engaged at various debugging stages.

Note that we did not implement SID's 'insert breakpoints' feature in CodeHinter, as the new version of the tool utilizes print statements for debugging instead, which are hypothesized to be simpler for novices. In the pilot study, we ask participants to test both CodeHinter and SID to compare these alternative strategies.

The helper tools in the end-to-end tests can also be manually selected by using the main menu item 'Helpers to Solve Bugs'. Four tools are currently provided: identifying lines with errors; providing hints and quizzes; inserting print statements; and opening Python Tutor [9]. These features are designed to assist users in debugging their code without directly providing solutions from an LLM. They can be triggered automatically when test cases fail, or manually by expanding the menu and selecting the desired option (Figure 1). We expand on the helper tools below.

Locate Lines With Errors. When users select this option, the tool runs Python's spectrum-based fault localization method, FauxPy [23]. FauxPy generates probability scores for source code line numbers, indicating the most likely sources of failed test cases. The algorithm identifies and ranks multiple lines based on their likelihood of containing faults. From this ranked output, we extract up to three lines with the highest probabilities.

Next, we leverage an LLM to provide explanations for why these specific lines are the most probable sources of error. This approach enhances the objectivity of the debugging process by ensuring that LLM-generated insights are grounded in the output of fault localization rather than speculative reasoning. By doing so, we mitigate the risk of hallucination when identifying potential errors.

The tool then highlights the identified lines and provides explanations to help users fix the error. It is important to note that fault localization tools like FauxPy identify lines where variables are not updated correctly, but these may not always be the exact source of the error. For instance, if a bug originates from an incorrect condition in an if-else statement, leading to a wrong assignment, FauxPy can detect that the values are assigned incorrectly but will not highlight that the issue stems from the condition itself. Although this approach may seem unintuitive, identifying lines where the expected value differs from the actual executed value aligns with how programmers typically debug their code using backward reasoning. Once they identify discrepancies in values, they hypothesize potential causes and trace the issue back to its source.

Provide Hint and Quiz. This feature is activated either when a syntax error occurs or when a user selects this option after encountering a semantic error. Instead of directly providing an explanation, the LLM generates three possible solutions, with only one being correct. These options suggest possible ways to correct the error, whether syntactical or semantic. Once the user selects an answer, the system immediately indicates whether the choice was correct and provides an explanation. By incorporating interactive quizzes, this feature encourages users to think critically while still receiving structured support.

Insert Print Statement. Research has shown that novice programmers often rely on print statements rather than using debugger mode [7]. To support this behavior, we leverage LLMs to suggest up to three key variables for users to observe by printing their values to the standard output. Once the LLM identifies the locations for the print statements, the tool provides a brief explanation in the chat about why these variables are relevant. A new tab is then opened, displaying the printed output with green-highlighted lines to help users visualize changes. Additionally, users have the option to paste the modified code into the text editor if they wish to incorporate the suggested changes.

One intentional design choice is that this process does not directly modify the source code, encouraging users to analyze the suggested change before making edits. The code is only modified when the user chooses to paste the suggested changes into the text editor. This approach allows users to manually insert print statements, selecting only the most relevant lines rather than applying all the suggested modifications.

Open Python Tutor. Given that novices rarely use debugger mode, to bridge this gap, the final helper tool is a button that redirects users to Python Tutor, allowing them to visualize code execution in an interactive environment. This approach provides a more intuitive and accessible debugging experience without requiring users to step into the complexity of debugger mode. We see this as a stepping stone for users, helping them gradually transition to advanced debugging techniques commonly used by more experienced programmers.

4 Methodology

We conducted a study with ten participants, divided into two groups of five. Though this number seems small, it is considered enough for usability testing according to [25]. Each participant was given 50 minutes to complete two tasks: (1) debugging a buggy program using CodeHinter; and (2) debugging another buggy program using the 'insert breakpoints' feature of SID. The only difference between the two groups was the sequence in which they used the tools. One group used CodeHinter first, followed by SID, while the other group used SID first, followed by CodeHinter. Each buggy question contained a maximum of two incorrect lines of code, and corresponding test cases were provided for both debugging tasks.

Before the session began, participants completed a profiling questionnaire to assess their programming background. We also provided a demonstration of the tools. During the debugging tasks, users had the flexibility to choose which helper tools within CodeHinter to use based on their preferences. However, we encouraged them to avoid revealing the provided code solution unless they were unable to solve the problem after 15 minutes. The participants' chat sessions were stored in a MongoDB NoSQL database to track their interactions and usage patterns. This allowed us to monitor which tools they accessed during the study.

The two test questions were sourced from LeetCode [14, 15] and cover different algorithmic challenges. The first question, *Move Zeroes* (#283), requires participants to move all zeroes in an array to the end while maintaining the relative order of the non-zero elements. The second question, *Summary Ranges* (#228), asks participants to summarize a sorted array of numbers into concise ranges of consecutive elements.

After completing the tasks, participants were required to submit a post-task survey and a standardized Brooke's system usability questionnaire [2]. The post-task survey consisted of common questions applicable to both CodeHinter and SID, followed by tool-specific questions. The common section assessed general usability, effectiveness, and user preferences, while the tool-specific sections focused on unique features, user confidence in debugging, and areas for improvement. Additionally, participants provided qualitative feedback on what they liked, disliked, and suggested features for future enhancements. As a token of appreciation, participants received the equivalent of USD 20 upon completion of the study.

5 Results

5.1 Profile of Participants

We recruited eight first-year and two second-year undergraduate students from the Singapore University of Technology and Design (SUTD). The first-year students had completed an introductory programming course but had not yet chosen their major, while the second-year students may have taken additional programming courses. We found that seven out of ten participants had experience writing over 500 lines of code for unique projects, suggesting they were not complete beginners. However, their confidence levels in debugging varied. While 50% felt confident in debugging simpler problems, only 40% were comfortable handling more complex issues. Regarding debugging preferences, 60% preferred guidance that included explanations of key actions needed to identify the bug. Notably, only two students regularly used debugging tools as part of their programming habits, while five students had never heard of any debugging tools.

5.2 Post-Task Survey Results

We collected survey responses from participants after they completed all the assigned tasks. Due to limited space, we only highlight results for CodeHinter and not SID. Figure 3 illustrates the features that participants found most useful in CodeHinter. The results indicate that the most helpful tools were 'End-to-End Test' and 'Locate Lines With Errors', which aligns with our expectations.

Additionally, we retrieved data from our database to track chatbot sessions and identify the most frequently used tools. The data

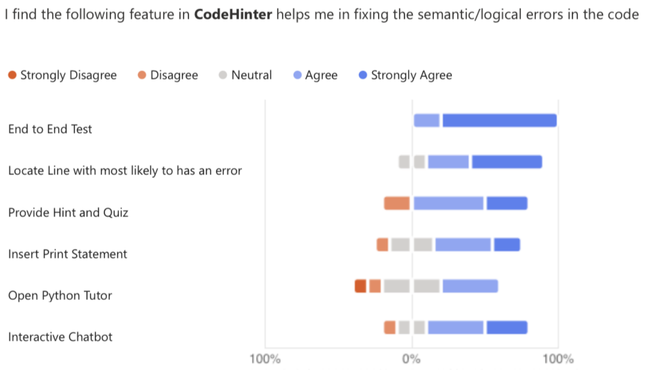


Figure 3: User perspectives on the usefulness of CodeHinter's features for fixing semantic errors.

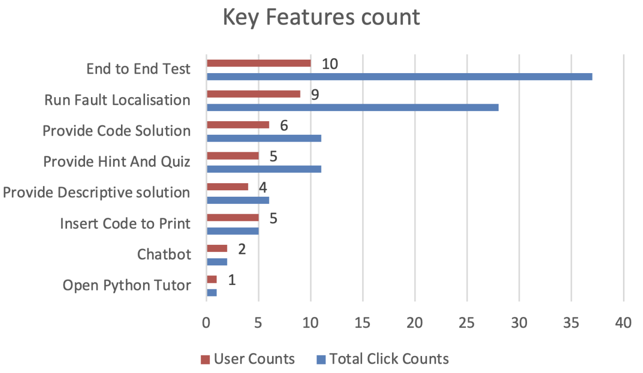


Figure 4: Frequency of access and utilization of features.

shown in Figure 4 only counts clicks on individual menu options, excluding instances where features were used as part of the 'End-to-End Test' button.

As expected, 'End-to-End Test' was used by all participants and ranked as the most frequently used feature. Among the four available helper tools, 'Locate Lines With Errors' was the most popular, with most participants using it multiple times, except for one user. The 'Provide Hint and Quiz' and 'Insert Print Statements' features were each used by half of the participants. However, participants tended to use 'Provide Hint and Quiz' multiple times, whereas 'Insert Print Statements' was typically used only once per user. Additionally, only one user utilized the 'Open Python Tutor in Web Browser' feature, while two users interacted with the general chatbot. Notably, six out of ten participants clicked the 'Provide Code Solution' button, with some clicking it multiple times, possibly to check and compare their answers.

5.3 Usability and Satisfaction

Using the standardized Brooke's System Usability Scale (SUS) [2], we calculated an average score of 75/100 for CodeHinter, compared to 65/100 for SID, as reported in our first study [18]. According to Brooke's methodology, SUS scores are most meaningful when compared across systems, allowing us to conclude that CodeHinter provides a significantly improved user experience over SID. This finding is further reinforced by the post-task survey, where participants gave an overall satisfaction rating of 89/100 for CodeHinter, compared to 60/100 for SID. The higher rating for CodeHinter suggests that users found it more effective and user-friendly in assisting with debugging tasks, indicating a strong improvement in our second design iteration.

Participants highlighted the strengths of CodeHinter, particularly in debugging semantic errors and its ease of use. Selected user quotes from the survey include:

> "It has a lot of features to help us debug semantic errors instead of simply copying and pasting solutions from other LLM AI models."
> "It was user-friendly and intuitive to use, providing accurate information."
> "I do not dislike it because of its high quality in helping beginners become more familiar with coding."

5.3.1 *End to End Test.*

Users expressed the highest level of satisfaction with this feature. All participants found it helpful, with two selecting 'Agree' and eight selecting 'Strongly Agree' in response to the statement that it significantly aided in problem-solving.

> "It helps me to know which logic errors I am still facing and also display the failed test cases that I need more time to debug the semantic errors."
> "It was very useful to have a tool that could provide me with descriptions of what happened during the testing of the test cases."

5.3.2 *Locate Line With Error.*

Among the four helper tools, 'Locate Lines With Errors' was the most frequently used and highly favored by users. In the survey measuring how often participants would use this feature, it received an average rating of 4.5/5. Interestingly, our database retrieval showed that one participant did not use this tool at all, as this participant was able to solve the problem just by utilizing 'End to End Test'.

> "This feature allowed me to more easily find possible logical flaws in my code which could've taken me a lot more time to spot without such tools."
> "This is by far the best feature which helps us seamlessly navigate where the bug is most likely to be."
> "I don't have to search and debug/print a million lines to find the root cause of the error."

5.3.3 *Hint and Quiz.*

This feature was also well-received by users. The majority of the participants appreciated its interactive approach, as it encourages thinking when fixing the bugs. However, one comment suggested that this feature may not be their preferred option if given a limited time to solve the problems in.

> "The hints provided make me think about where the error is in my code by myself."
> "Effective. It is interactive and helps the user to understand the bug."
> "I am unlikely to use this tool unless I am very desperate to solve a problem within a time duration, and

> in that situation I wouldn't want to be quizzed with possible solutions."

5.3.4 Insert Print Statement. The results for 'Insert Print Statement' are similar to the 'Provide Hint and Quiz' feature, with half of the participants attempting to use it. However, each user only used this feature once, unlike 'Provide Hint and Quiz', which participants interacted with multiple times. This difference may be because locations and variables of the print statements tend to be the same depending on the structure of the code. On the other hand, the 'Provide Hint and Quiz' feature generates different options as the code changes. This could happen if the users apply a wrong fix. Users also mentioned that 'Insert Print Statement' helps them debug faster by allowing them to quickly spot errors, making it especially beneficial for beginners.

> "I think this tool can be useful for simple testing and spotting errors in the code, especially for beginners who are less familiar with the debugging tool."
> "Can do the print statements for me faster, which normally takes a while to do."

6 Discussion & Future Work

Overall, participants provided positive feedback on CodeHinter. As expected, users found the 'End-to-End Test' button highly useful, as indicated by both its frequent usage among all participants and their comments about the tool. Among the individual tools designed to assist with debugging, users found the 'Locate Lines With Errors' feature the most helpful. This aligns with previous studies showing that one of the most challenging aspects of debugging for novice programmers is locating errors [7, 17]. Additionally, research on the use of APR has similarly found that error location messages are the most helpful information for novice programmers when fixing their code [13].

Our study results also showed positive feedback on the use of the 'Hints and Quiz' helper tool to engage the participants in the debugging process. Instead of directly specifying what to replace, as most previous tools do, incorporating quizzes encourages learners to think critically and formulate hypotheses about the possible causes of errors and potential solutions. Participants' comments after using the tool aligned with our intended approach, confirming that novice debuggers can be actively involved in aspects of the debugging process, such as hypothesizing the cause of errors and identifying possible fixes.

Regarding the 'Insert Print Statement' feature, we were initially surprised that each user used it only once for the problem they worked on with CodeHinter. However, their comments were largely positive, indicating that they found the feature useful. Therefore, we hypothesize that, unlike 'Provide Hints and Quiz', this feature serves a one-time function in helping users identify bugs. Once the print statements are inserted, users do not need to reinsert them; instead, they simply run the code to observe the output. Additionally, this feature was introduced as part of our design exploration, serving as an alternative to automatically inserting breakpoints, as implemented in the first iteration of our tool, SID. Our initial hypothesis was that novice programmers find using print statements easier than working with an IDE's debug mode. However, we discovered that some intermediate programmers prefer using breakpoints and the debug mode instead. This suggests that future iterations of the tool should offer both options, allowing users to choose the method that best suits their debugging preferences.

This leads us to conclude that personalizing the tool based on the user's profile is essential. With advancements in AI, it is increasingly feasible to profile users based on their interactions with debugging tools. By doing so, novice programmers can be prompted with features better suited to their skill level, while more experienced users can be provided with advanced debugging tools tailored to their needs. Additionally, the tool can be adapted based on the specific problem being debugged, ensuring a more effective and user-centric debugging experience.

Furthermore, the participants also provided valuable suggestions for future improvements. For example, participants suggested incorporating real-time assistance not only for writing code but also for debugging, enabling dynamic support as they work through identifying and fixing bugs. Additionally, they expressed interest in integrating code quality analysis during debugging to provide more tailored feedback.

This work is limited by its small participant pool, as it represents a pilot study focused on the tool's design and usability. Further research should examine its impact on learning outcomes, particularly how it influences the development of debugging skills among novice programmers.

7 Conclusion

In conclusion, this study demonstrates the potential of AI-assisted tools like CodeHinter to enhance the debugging experience for novice programmers by fostering active engagement rather than passive reliance on generating complete solutions. Through a user-centric design that integrates spectrum-based fault localization, interactive hints and quizzes, and print statement suggestions, all within a familiar IDE environment, CodeHinter effectively supports the debugging process. The tool's usability and effectiveness were validated in a pilot study, with participants favoring features that guided them through the problem-solving process while reinforcing critical thinking. Our findings also underscore the importance of tailoring AI support to individual user profiles and problem contexts. Future debugging tools should focus on providing personalized assistance to optimize learning outcomes and foster independent debugging skills.

Acknowledgments

This work was supported by the Ministry of Education, Singapore, under the Tertiary Research Fund (MOE-TRF) Grant No. MOE2023-TRF-034. We gratefully acknowledge this support, which made this research possible. We would also like to acknowledge the use of OpenAI's ChatGPT for assisting in the refinement of English and improvement of paragraph clarity throughout the writing process. This study was reviewed and approved by the Institutional Review Board (IRB) of the Singapore University of Technology and Design (SUTD) under IRB protocol number S-24-669. All participants provided informed consent prior to their involvement in the study.

References

[1] Brett A Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael

Osera, Janice L Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (New York, NY, USA). Association for Computing Machinery, 177–210.

[2] John Brooke. 1996. SUS: A quick and dirty usability scale. *Usability Evaluation in Industry* (1996).

[3] Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. 2014. Syntax errors just aren't natural: improving error reporting with language models. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (New York, NY, USA). Association for Computing Machinery, 252–261.

[4] Sharon McCoy Carver and Sally Clarke Risinger. 1987. *Improving children's debugging skills.* Ablex Publishing Corp., 147–171.

[5] Jonathan E Collins. 2023. Policy Solutions: Policy questions for ChatGPT and artificial intelligence. *Phi Delta Kappan* 104 (2023), 60–61. Issue 7.

[6] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2012. All syntax errors are not equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (New York, NY, USA). Association for Computing Machinery, 75–80.

[7] Sue Fitzgerald, Gary Lewandowski, Renée McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: Finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education* 18 (2008), 93–116. Issue 2.

[8] GitHub. 2023. GitHub Copilot. https://github.com/features/copilot Accessed: March 12, 2025.

[9] Philip Guo. 2021. Ten Million Users and Ten Years Later: Python Tutor's Design Guidelines for Building Scalable and Sustainable Research Software in Academia. In *UIST 2021 - Proceedings of the 34th Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery, Inc, 1235–1251.

[10] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. *Proceedings of the AAAI Conference on Artificial Intelligence* 37 (6 2023), 5131–5140. Issue 4.

[11] Irvin R Katz and John R Anderson. 1987. Debugging: an analysis of bug-location strategies. *Hum.-Comput. Interact.* 3 (12 1987), 351–399. Issue 4.

[12] Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, Austin Z. Henley, Paul Denny, Michelle Craig, and Tovi Grossman. 2024. CodeAid: Evaluating a Classroom Deployment of an LLM-based Programming Assistant that Balances Student and Educator Needs. In *Conference on Human Factors in Computing Systems - Proceedings*. Association for Computing Machinery.

[13] Oka Kurniawan, Christopher M. Poskitt, Ismam Al Hoque, Norman Tiong Seng Lee, Cyrille Jégourel, and Nachamma Sockalingam. 2023. How Helpful do Novice Programmers Find the Feedback of an Automated Repair Tool?. In *2023 IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE)*. 1–6.

[14] LeetCode. n.d.. Move Zeroes. https://leetcode.com/problems/move-zeroes/description/ Accessed: March 12, 2025.

[15] LeetCode. n.d.. Summary Ranges. https://leetcode.com/problems/summary-ranges/description/ Accessed: March 12, 2025.

[16] 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 *SIGCSE 2023 - Proceedings of the 54th ACM Technical Symposium on Computer Science Education*, Vol. 1. Association for Computing Machinery, Inc, 563–569.

[17] Renée McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: A review of the literature from an educational perspective. Issue 2.

[18] Yannic Noller, Erick Chandra, Srinidhi Chandrashekar, Kenny Choo, Cyrille Jegourel, Oka Kurniawan, and Christopher M. Poskitt. 2025. Simulated Interactive Debugging. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE'25): New Ideas and Emerging Results*. ACM.

[19] OpenAI. 2022. Introducing ChatGPT. https://openai.com/index/chatgpt/ Accessed: March 12, 2025.

[20] OpenAI. 2024. Learning to reason with LLMs. https://openai.com/index/learning-to-reason-with-llms/ Accessed: March 12, 2025.

[21] Tung Phung, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. 2023. Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models. In *Proceedings of the 16th International Conference on Educational Data Mining, EDM 2023, Bengaluru, India, July 11-14, 2023*. International Educational Data Mining Society.

[22] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive Difficulties Faced by Novice Programmers in Automated Assessment Tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (New York, NY, USA). Association for Computing Machinery, 41–50.

[23] Mohammad Rezaalipour and Carlo A Furia. 2024. FauxPy: A Fault Localization Tool for Python. *arXiv preprint arXiv:2404.18596* (2024).

[24] Rebecca Smith and Scott Rixner. 2019. The error landscape: Characterizing the mistakes of novice programmers. In *SIGCSE 2019 - Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, Inc, 538–544.

[25] Robert A. Virzi. 1992. Refining the Test Phase of Usability Evaluation: How Many Subjects Is Enough? *Human Factors* 34, 4 (1992), 457–468.

[26] Jialu Zhang, José Pablo Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2024. PyDex: Repairing Bugs in Introductory Python Assignments using LLMs. *Proceedings of the ACM on Programming Languages* 8 (4 2024). Issue OOPSLA1.

[27] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A Survey of Learning-based Automated Program Repair. *ACM Trans. Softw. Eng. Methodol.* 33 (12 2023). Issue 2.

[28] Quanjun Zhang, Chunrong Fang, Yang Xie, YuXiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. 2024. A Systematic Literature Review on Large Language Models for Automated Program Repair. (5 2024). http://arxiv.org/abs/2405.01466

[29] Quanjun Zhang, Tongke Zhang, Juan Zhai, Chunrong Fang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. A Critical Review of Large Language Model on Software Engineering: An Example from ChatGPT and Automated Program Repair. (10 2023). http://arxiv.org/abs/2310.08879