Fostering Responsible AI Use Through Negative Expertise: A Contextualized Autocompletion Quiz

Stephen MacNeil
Temple University
Philadelphia, PA, US
stephen.macneil@temple.edu

James Prather
Abilene Christian University
Abilene, TX, USA
james.prather@acu.edu

Rahad Arman Nabid
Temple University
Philadelphia, PA, US
rahad.arman.nabid@temple.edu

Sebastian Gutierrez
Temple University
Philadelphia, PA, US
guts@temple.edu

Silas Carvalho
Temple University
Philadelphia, PA, US
silas.neto.carvalho@temple.edu

Saimon Shrestha
Temple University
Philadelphia, PA, US
tup35165@temple.edu

Paul Denny
University of Auckland
Auckland, New Zealand
paul@cs.auckland.ac.nz

Brent N. Reeves
Abilene Christian University
Abilene, TX, USA
brent.reeves@acu.edu

Juho Leinonen
Aalto University
Espoo, Finland
juho.2.leinonen@aalto.fi

Rachel Louise Rossetti
Google
Boulder, CO, USA
rrossetti@google.com

Abstract

Generative AI tools, like GitHub Copilot, are becoming an industry standard by offering real-time code suggestions that streamline the coding process. Although these systems improve productivity, they also introduce pedagogical challenges. Students may become overly reliant on AI-generated code suggestions, accepting them without critical thought, potentially reducing their ability to engage with the underlying logic of the code. We developed an interactive quiz system within a simulated IDE environment designed to help students think critically about autogenerated code suggestions. Instructors use the tool to create contextualized coding quizzes that present multiple code suggestions at each line. Students must choose the correct option to move on to the next step. Survey responses suggest that this approach could promote critical thinking and scaffold metacognitive skills like planning and reflection. Students reported that the system helped them distinguish between good and bad suggestions. Most students preferred this experience to traditional quizzes or Github Copilot. These findings show the potential to scaffold more critical use of generative AI coding tools.

CCS Concepts

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

Keywords

Auto-completion, Copilot, Negative Expertise, Generative AI

ACM Reference Format:
Stephen MacNeil, James Prather, Rahad Arman Nabid, Sebastian Gutierrez, Silas Carvalho, Saimon Shrestha, Paul Denny, Brent N. Reeves, Juho Leinonen, and Rachel Louise Rossetti. 2025. Fostering Responsible AI Use Through Negative Expertise: A Contextualized Autocompletion Quiz. In *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2025), June 27-July 2, 2025, Nijmegen, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3724363.3729067

1 Introduction

Programming is not just a technical skill but a powerful vehicle for developing problem-solving and critical thinking skills that are fundamental to computing education [31, 36]. However, the successful development of these skills is challenged by the growing prevalence of AI-powered auto-completion tools, such as GitHub Copilot [1, 6, 40]. While these tools can accelerate coding tasks and reduce syntax errors, they can also lead to negative interaction patterns when students fail to critically evaluate AI-generated suggestions [27]. For example, students may develop an over-reliance on AI suggestions leading to an 'illusion of competence' [28]. When students blindly accept a model output that is logically flawed, they can be led down a wrong path that results in unproductive debugging cycles. In addition, frequent interruptions from auto-suggestions can be distracting for novices [28]. We hypothesize that these challenges can be addressed by teaching students how to read and critically evaluate AI-generated code suggestions.

A classic approach for scaffolding code writing involves students constructing programs by selecting fragments of code from a set of fixed choices. Parsons problems [10, 11] and block-based programming [3, 39] are both popular and widely studied examples in this category, differing in the level of flexibility provided in the choices.

However, both are static in the sense that the options available to students are predetermined. In contrast, AI code-suggestion tools are dynamic and they can generate next-line suggestions based on the current context. To scaffold the use of these tools, one potential approach is to present students at each line of code with dynamically generated options—both correct and incorrect—based on the partial program constructed so far. Essentially, a student would build a program by answering a multiple-choice question for each line of code by evaluating the options available at each step. Unlike a Parsons problem with pre-defined distractors [34], our proposed approach has the potential to support a less constrained exploration of the solution space, and the possibility for different students to explore different pathways.

In this work, we introduce an *Autocompletion Quiz* tool which was designed to teach students critical thinking about code while using Copilot-based suggestions. Our work draws from Minsky's concept of *negative expertise* [22] by leveraging the educational benefits of understanding common pitfalls and ineffective strategies. By presenting *contextualized* suggestions [12] in a multiple-choice format, the tool encourages students to engage with both correct suggestions and problematic ones which are rooted in common coding misconceptions [29]. This aligns with the idea that exposing students to incorrect patterns helps them build the skills needed to identify and avoid them when using AI tools that offer code suggestions. Our analysis focuses on the following research questions:

RQ1: How did students interact with the autocompletion quiz tool?
RQ2: How do students describe their experiences using the autocompletion quiz tool?

We make the following contributions in this paper: 1) a novel programming practice format designed to promote critical thinking and 2) empirical insights into students' interactions with the autocompletion quiz tool. As future work, we intend to evaluate the impacts of this approach on student learning outcomes, and explore the benefits of different variants of the tool.

2 The Pitfalls of Generative AI

Generative AI has become increasingly common in computing education, which has led researchers to investigate the potential impacts on teaching and learning. While many benefits have been identified, concerns have also been raised about how effectively these models can solve problems at the CS1 and CS2 levels [13, 32], Parsons problems [16, 30], and graph and tree data structure tasks using only image-based inputs [15]. This has led some instructors to share concerns about students relying on AI-generated code they do not understand and becoming overly dependent on these tools [2, 8, 18, 25, 41].

Recent studies suggest that the benefit of generative AI tools depends heavily on their design and instructional context of use. A recent working group by Prather et al. [26] found that incorporating generative AI into teaching can positively impact tasks like code comprehension, especially for custom-built AI tools that feature pedagogical guardrails. However, for code-writing tasks, results have been mixed, especially when students use industry-grade tools like GitHub Copilot without explicit training.

Generative AI appears to benefit high-performing students disproportionately [21]. Advanced students can use these tools to accelerate their learning, while novice programmers face metacognitive challenges that widen the gap between skill levels [28]. This is because novice programmers struggle to effectively use generative AI tools. For example, they may have trouble understanding code generated by the model [27], and models may lead them down 'debugging rabbit holes' [38]. Usage patterns are similarly uneven with some students using AI tools daily, while others avoid them entirely [17]. This bimodal distribution risks creating a digital divide reminiscent of the early personal computer era [41].

The uneven use of generative AI along with the potential negative impacts on cogntive and metacognitive processes requires more intentional pedagogy and tools to help students use AI more effectively. As an example, Prompt Problems teaches students to prompt by generating code based on their prompt and automatically evaluating it against a test case [7]. Instead, our work focuses more on code comprehension by introducing a novel quiz system designed to give students practice evaluating AI-generated code suggestions. By scaffolding evaluation skills, our goal is to prepare students to use generative AI tools like GitHub Copilot effectively and improve their ability to assess the usefulness and correctness of code suggestions.

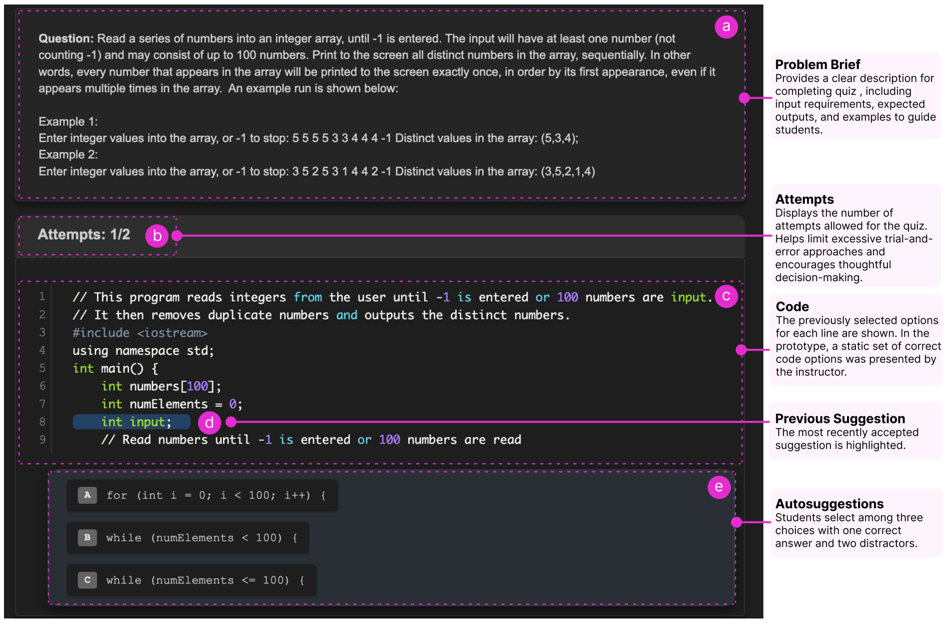
3 System Overview

The *Autocompletion Quiz* system introduces a novel programming practice format, designed to promote critical engagement with AI systems. By leveraging the concept of *negative expertise* [22], the system challenges students to distinguish between good and bad AI suggestions. The system provides an interface to generate quizzes so that instructors can use an LLM to generate distractors or write their own. Figure 1 provides an overview of the system's user interface. In our study, an LLM was used to generate all of the distractors to ensure that distractors were the same for every student in the study.

3.1 Design Rationale

3.1.1 Progressive Quiz Format. The quiz adopts a progressive, step-by-step format in which students need to select the intended suggestion from two distractors for each line of code, and each line builds on prior choices. The design was inspired by Donald Schön's concepts of *reflection-in-action* (critical thinking on current options) and *reflection-on-action* (evaluating previous decisions) [33] with the goal to have students engage with the current suggestions and their prior code. Although the correct options were predefined by the instructor in this study, the system also has the ability to generate them dynamically using an LLM.

3.1.2 Fostering Negative Expertise. Negative expertise [22] is a pedagogical approach students critically engage with incorrect or suboptimal strategies to enable them to better recognize and avoid them. Our quiz leverages this principle by presenting students with plausible but flawed suggestions (i.e.: distractors) alongside an ideal suggestion. To maximize ecological validity, the distractor options were generated using a LLM, following methods inspired by recent work [20] and aligned with prior research that used LLMs to generate correct answers and distractors [9, 23, 37].



Question: Read a series of numbers into an integer array, until -1 is entered. The input will have at least one number (not counting -1) and may consist of up to 100 numbers. Print to the screen all distinct numbers in the array, sequentially. In other words, every number that appears in the array will be printed to the screen exactly once, in order by its first appearance, even if it appears multiple times in the array. An example run is shown below:

Example 1:
Enter integer values into the array, or -1 to stop: 5 5 5 5 3 3 4 4 4 -1 Distinct values in the array: (5,3,4);
Example 2:
Enter integer values into the array, or -1 to stop: 3 5 2 5 3 1 4 4 2 -1 Distinct values in the array: (3,5,2,1,4)

Attempts: 1/2

```
// This program reads integers from the user until -1 is entered or 100 numbers are input.
// It then removes duplicate numbers and outputs the distinct numbers.
#include <iostream>
using namespace std;
int main() {
    int numbers[100];
    int numElements = 0;
    int input;
    // Read numbers until -1 is entered or 100 numbers are read
```

```
A  for (int i = 0; i < 100; i++) {
B  while (numElements < 100) {
C  while (numElements <= 100) {
```

Problem Brief
Provides a clear description for completing quiz , including input requirements, expected outputs, and examples to guide students.

Attempts
Displays the number of attempts allowed for the quiz. Helps limit excessive trial-and-error approaches and encourages thoughtful decision-making.

Code
The previously selected options for each line are shown. In the prototype, a static set of correct code options was presented by the instructor.

Previous Suggestion
The most recently accepted suggestion is highlighted.

Autosuggestions
Students select among three choices with one correct answer and two distractors.

Figure 1: System Overview: An illustration of the key components and workflow of the Autocompletion Quiz system.

These distractors are not strictly 'incorrect' as most code suggestions could still produce correct outputs through later corrections. For example, initializing an accumulator variable with a value other than zero, such as one, is generally a poor practice, but the issue can be resolved by decrementing the value later. To reflect this, we distinguish between **good suggestions**, which represent the intended ideal approach, and **bad suggestions**, which are the distractors.

3.1.3 Ecologically Valid Visual Design. To increase authenticity and knowledge transfer, the system was designed to mimic the visual appearance of a standard integrated development environment (IDE). This design choice, grounded in the theory of situated cognition and transfer learning [5], places students into an environment that replicates real-world programming workflows. By simulating the experience of receiving suggestions from tools like GitHub Copilot, the system was designed to improve ecological validity and increase the likelihood that the skills acquired during practice will transfer when students interact with Github Copilot.

4 Methods

To evaluate our novel interactive system, we conducted a user study with 32 computing students at a research university in the Southwestern United States. We collected log data based on their interactions and participants also completed a summative survey about their experiences and ideas for new features.

4.1 Participants

Participants ($n = 32$) were all in their first programming course. This CS1 course integrated generative AI from the beginning, including utilization in lectures for writing code (GitHub Copilot) and debugging (ChatGPT). Students were given traditional write code questions for homework, but received no credit for them. Instead, weekly quizzes were given in class on paper consisting of three code tracing problems almost exactly like the solutions for the homework plus one write code question.

Although 32 students participated in our study, only 29 completed the demographic survey. Twenty-one participants identified as men, eight identified as women, and no participants identified as non-binary or another gender. Twenty-seven students were ages 18-22, the typical age of university students in the United States, while two were older. Four were international students. Racial diversity reported was 17 Caucasian, 7 Hispanic, 4 Black, and 2 Asian. Fourteen students were Computer Science majors while the rest were from other STEM disciplines. Fifteen students reported that they had taken programming in high school.

This research was approved by the university IRB and all participants signed a consent form to participate voluntarily.

4.2 Procedure

Students were provided some instruction about what to expect in the activity and then provided a link to the tool. Although participants were allowed to use the entire 50-minute class period to complete the quiz and submit the post-assessment survey, all finished before the end of the class session.

4.2.1 Survey Questions. In a post-activity survey, we asked two Likert questions (seen in Figure 2), and these open-ended questions:

- Can you please share your overall experience during this learning activity, including what went well and did not?
- During this learning activity, the lines were presented one-by-one, please share your experience with reading each line and choosing the correct one.

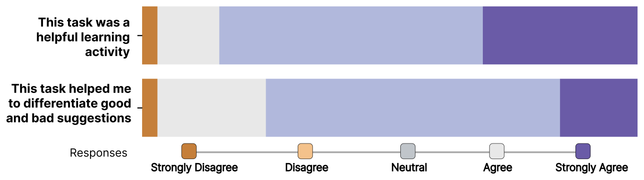


Figure 2: Likert responses from participants.

- Can you compare this learning activity to a traditional quiz, what aspects are better and which are worse?
- Can you compare this learning activity to writing code with Github Copilot, what aspects are better and which are worse?
- As we continue to improve this tool, what do you think are the pros and cons of making each line timed to simulate the speed of using Github Copilot?
- As we continue to improve this tool, what do you think are the pros and cons of suggesting multiple lines of code at a time versus just a single line?
- What other ideas do you have for improving this tool?

4.3 Analysis

Given the relative novelty of our approach, there was no obvious comparable baseline to use in a between-subjects study. Instead, we conducted a qualitative analyses to understand students' experiences with the tool. We also report descriptive statistics to contextualize their experiences.

A single researcher analyzed the students' responses to open response questions using a reflexive thematic analysis [4], starting by open coding [35] each response. Each question was first analyzed independently to identify patterns related to that specific question. Themes were then developed within individual questions and across questions when appropriate to identify broader patterns. Themes were iteratively reviewed and refined multiple times to ensure that they represented the perspectives of participants and that they were meaningful from a research perspective. In the results section, themes are presented with representative quotes to contextualize the findings and to ground them in participants' voices.

5 Results

5.1 RQ1: Student Interactions with the Tool

5.1.1 Completion Times and Distractors. All 32 participants were able to reach the end of the task, i.e. respond to 'line 38.' The mean time for completing the whole quiz was 7:33 and ranged from 4:30 to 11:30. Given the problem description shown to students and the way programs are constructed (e.g. declaring variables near the beginning), we anticipated that some lines would be more complex, and therefore more difficult, than others. Although students clicked distractors on 24 of the program's 38 lines (see Table 1), most of those were only done by a few participants. Furthermore, the number of students who picked a second distractor usually dropped precipitously, often going to zero. This was true for most lines.

Some distractors were simple mistakes, such as a distractor with angle brackets facing the wrong direction on line 11: "`cout » input;`" rather than "`cout « input;`". Other distractors were more difficult to ascertain, such as line the iteration control flow on line 10 where 19 participants initially chose a distractor. The intended answer was "`while(numElements < 100)`", but one distractor was "`while(numElements <= 100)`" and the other distractor was "`for(int i=0; i<100; i++)`". Although possible to use a for loop as suggested, students had been taught in the class to use while loops with indefinite tasks and for loops with definite tasks. It therefore made a more subtle but still appropriate distractor. For this line, 12 participants chose a second distractor.

5.1.2 Students' Responses to Likert Questions. The students responded to two Likert questions along a 5-point scale from Strongly Disagree to Strongly Agree. As shown in Figure 2, the responses to both questions were unaminously neutral to positive, with one exception. This provides promising initial evidence that this approach might be well received by students.

5.2 RQ2: Students' Experiences and Perceptions

Aligning with the Likert responses, students were extremely positive about their experiences using the tool in the open-response questions. Students emphasized benefits such as supporting critical thinking, focusing their attention, and supporting the metacognitive processes of planning and reflection. We present these themes in the following subsections.

5.2.1 Directing Attention to Key Code Variations. Students frequently described how the tool guided their attention toward subtle but important variations in code. By presenting multiple options for each line, students were encouraged to analyze these differences and evaluate which choice was ideal. One student explained:

> *"I felt overall it was just a very helpful learning tool to learn **how to differentiate slight differences in code** that can cause big problems later." (P18)*

This careful comparison of slight differences aligns with the *variation theory*, in which key variations between exemplars (i.e.: learning objects) can support learning [14]. This careful evaluation of minor variations has benefits, but students also expressed facing difficulties when unfamiliar variations were presented. For example, another student remarked,

> *"I was able to solve the next step by looking at my option and thinking of the minor differences of each one... I did get some wrong just because I did not know what some things meant." (P14)*

Beyond attention to the suggestions, a few students talked about how the tool encouraged them to 'slow down' and to pay closer attention. For example, one of the students shared,

> *"**It helped me to slow down and carefully think before moving on.** Usually I just sort of speed through some of the lines and take time to think about others... some lines threw me off in this quiz, but I think it was because I actually had to pay full attention to it rather than just use muscle memory typing." (P18)*

Finally, in two cases, students identified instances where multiple options might be 'correct' at the same time. Both claimed that this forced them to consider efficiency and other aspects. For example, P8 described this experience:

Line Number	3	5	6	7	8	10	11	12	15	16	18	19	21	22	23	24	25	26	29	30	31	35	36	38
Clicked first distractor	2	2	2	10	3	19	9	1	8	1	3	5	8	13	9	8	4	9	16	13	1	6	14	7
Clicked second distractor	0	0	0	5	0	12	1	1	1	0	0	0	0	5	1	0	0	3	0	6	0	1	1	4

Table 1: The number of participants who selected distractors as their first and second choices, organized by line number. Some line numbers are excluded because they correspond to comments or blank lines, which were included to improve readability.

> *"There were a few problems where it felt like it could be either of two answers, but looking back, one was almost always more efficient, even though both could work."*

While for the majority of students the tool encouraged them to slow down and attend to the code and to minor variations between the suggestions, at least two students struggled with this task.

5.2.2 Reflecting on Prior Code. Another theme identified in the analysis was that the tool appeared to foster a reflective practice, with students frequently revisiting the prior code in order to evaluate new suggestions. For example, P20 said:

> *"With the lines being in order it was pretty simple, and choosing the correct next line was just using the previous lines to pick the next best answer." (P20)*

This process of reflecting on earlier design decisions to inform their next steps indicates they are actively thinking about how they're solving the problem which is an important metacognitive activity. P9 reiterated this theme, saying:

> *"It being one line at a time made it a lot simpler being able to use what you've previously put to use as a reference for what to put next." (P9)*

5.2.3 Needing to Think Ahead and Anticipate. One of the benefits that students ascribed to the progressive quiz format was the need to think ahead about how to solve the problem. This need to anticipate the implications of each coding decision is indicative of metacognitive planning, where learners set goals, making a plan, and then monitor their plan as they carried it out

> *"**I had to think one step ahead** and ask myself how this line would fit into the larger task the code was trying to achieve." (P29)*

By showing each new line along with previous lines, students described needing to think ahead to figure out where the code was 'going'. For example, P1 describes this experience of thinking ahead, but also describes 'losing track' of what they were doing:

> *"Overall it went well. A couple of times **I lost track of what the code was doing** and messed up but for the most part it was fine...It made me think ahead about what would come next and also about what line of code helps me get there the best." (P1)*

This experience of getting off track may be related to how students want to solve the problem which is not aligned with how the tool guided them all to the same solution. For instance, P6 remarked,

> *"Some of the things that they were doing where different than my thought process so at times I was not completely sure what they were doing" (P6)*

Another student similarly remarked:

> *"Initially, it was hard to figure out how the problem was going to be solved, which made declaring the variables odd sometimes **because I was thinking of a different way to solve it.** So, it was a little different to have to solve it a certain way, **but beneficial because I needed to think about it.**" (P10)*

5.3 Comparisons to Traditional Quizzes

When comparing their experiences using the tool with traditional quizzes, students consistently highlighted the distinct purposes served by each. They perceived the tool as better suited for **formative** assessment, demonstrating its potential to facilitate learning and skill development, while traditional quizzes were viewed primarily as tools for **summative** evaluation, focusing on gauging mastery. For example, P32 said:

> *"It is good for preparation but traditional quiz is better for checking the root knowledge in a particular concept."*

Across responses, 16 students explicitly described the tool as having advantages over traditional quizzes, with only two students stating that it was less effective. The tool was frequently described as more interactive and requiring deeper engagement, with students reporting that it encouraged active reflection on their decision-making processes. One student remarked:

> *"I found it easier than a traditional quiz, but it also forced me to question myself and **be more conscious of my decisions.**" (P18)*

This included students describing the need to engage more deeply with the content rather than scanning or skimming it:

> *"During a traditional quiz I would more than likely **scan code rather than go through line by line**...it was very unique in the fact that it made the user have to think about each line of code. Usually, I just scan over code and this forced me to think more critically." (P15)*

5.4 Comparisons to Github Copilot

When comparing their experiences to using Github Copilot, students were again largely positive with 12 students preferring the tool and 3 students describing some way in which Copilot was more helpful. The primary reasons provided by students were that the quiz has better support for critical thinking and is more beginner friendly. In terms of critical thinking, P1 and P24 shared:

> *"It was better than using copilot. **Copilot just gives you the answer most of the time and takes out a form of critical thinking** and problem solving that is vital in coding." (P1)*
>
> *"You have to think less with GitHub Copilot because the correct answer is given to you faster..." (P24)*

Despite the majority of students agreeing with this sentiment, P6 shared the opposite opinion, claiming that they wanted to be more involved with driving the suggestions:

> *"I feel like here they pretty much do the code for you, you just have to choose what you think is best, Github I feel is more ourselves and our own thought process."*

5.5 Students' Suggestions for Improvements

Students mentioned trade-offs related to agency, such as wanting to declare their own variables or writing their own code:

> *"The program that was constructed feels like one I would have made, though I prefer to declare all my variables at the top, unlike what was done in the program." (P3)*

Others proposed the opposite, wanting more comments and signposting to understand where the code was leading them. For example, P10 requested,

> *"Keep including comments, those were helpful to know where it was trying to go next." (P10)*

6 Discussion

Generative AI tools such as GitHub Copilot introduce new challenges for students working on programming tasks. These tools often require users to successfully monitor, evaluate, and self-regulate their interactions—metacognitive skills that are already difficult for many students [24]. Without careful guidance, students can fall into unproductive behaviors such as 'debugging rabbit holes' or relying too heavily on AI-generated solutions which can negatively impact learning or frustrate students [38]. Compounding this problem, generative AI tools can also interfere with metacognitive processes [28]. Therefore, scaffolding mechanisms are needed to mitigate the challenges of using AI while also helping students develop skills for critical thinking and metacognition. In this work, we present a system designed to address these challenges by encouraging critical engagement with AI tools such as GitHub Copilot. The system scaffolds students' decision-making by offering correct and flawed code suggestions.

Our findings suggest that our prototype supported the students in several ways. It drew their attention to the code and to key differences between the suggestions, which is an important aspect of variation theory [14]. Students also described engaging in multiple metacognitive processes [19], including reflection, monitoring, and planning. Compared to using Github Copilot, students preferred the tool, citing its ability to promote critical thinking. Some students did not appreciate how opinionated the tool was about the solution and described having a different idea of how the code should have been written. However, in the negotiation of their plan with the AI's 'plan', some students described metacognitive benefits and an opportunity for critical thinking.

Our work is in direct response to the findings recently presented by Prather et al. [28] on novice programmers using AI coding tools, such as GitHub Copilot. They found that novices often struggled to read, comprehend, and identify good autogenerated code suggestions. Some of their participants uncritically accepted code suggestions, which led them down the wrong problem solving path. We designed the quiz tool presented above to mitigate some of these challenges. Students found it to be a useful formative tool that provides a forcing function to pay attention to each line and each suggestion. By incentivizing attention to small details, the tool promotes slowing down to enhance code reading and comprehension. It also helps prepare students to identify bad autogenerated suggestions so as to be less distracted or misled by them.

6.1 Future Work

Our findings provide encouraging preliminary evidence that this approach can help students learn to overcome the metacognitive harms imposed by Generative AI [28]. However, this prototype only scratches the surface of a larger solution space. The tool could be expanded to autogenerate the code suggestions for each line in real-time, rather than before as we did for this study, which would be even more ecologically valid for GitHub Copilot. This would enable multiple 'solution paths' and solve some of the rigidity issues noted by students around a single suggestion. These solutions would then be tested via test cases to ensure correctness. Students could even backtrack from errors to identify points where a suggestion led them off an ideal solution path. Finally, future work should investigate how this tool impacts learning gains.

7 Limitations

This study has a few limitations. First, the sample size was relatively small and was drawn from a single class. Although this limits generalizability, it is consistent with the exploratory nature of our study and appropriate for the use of qualitative research methods. Our findings should therefore be viewed as a foundation for future, larger-scale studies rather than as conclusive evidence.

Second, we designed the tool to maximize ecological validity; however, we presented students with three suggestions rather than the single suggestion typically offered by tools like GitHub Copilot. This gave students more opportunities to critically engage with suggestions. However, alternative designs could be explored as future work. For example, the tool could show a single suggestion that students must either accept or reject.

Third, the distractors used in the study were generated by an LLM which was prompted to create suggestions based on common misconceptions. While this approach aligns with our goal of testing students' critical thinking, we acknowledge that the problematic suggestions students encounter in real-world scenarios may differ. LLM-generated hallucinations can vary widely, encompassing errors unrelated to known student misconceptions.

8 Conclusion

As generative AI tools become increasingly common in educational and professional contexts, it is imperative to design systems that go beyond convenience and automation to actively support critical thinking and metacognitive development. Our work provides a first step in this direction along with suggestions for future work.

Acknowledgments

This research was supported by a Google GARA grant, the Research Council of Finland (Academy Research Fellow grant number 356114), and Boost Funding from the College of Science and Technology at Temple University.

References

[1] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 78 (April 2023), 27 pages.

[2] 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 (SIGCSE 2023)*. ACM, 500–506. https://doi.org/10.1145/3545945.3569759

[3] Luiz Carlos Begosso, Luiz Ricardo Begosso, and Natalia Aragao Christ. 2020. An analysis of block-based programming environments for CS1. In *2020 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–5.

[4] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.

[5] John Seely Brown, Allan Collins, and Paul Duguid. 1989. Situated cognition and the culture of learning. *1989* 18, 1 (1989), 32–42.

[6] 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 (SIGCSE 2023)*. ACM, 1136–1142.

[7] Paul Denny, Juho Leinonen, James Prather, Andrew Luxton-Reilly, Thezyrie Amarouche, Brett A. Becker, and Brent N. Reeves. 2024. Prompt Problems: A New Programming Exercise for the Generative AI Era. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024)*. ACM, 296–302. https://doi.org/10.1145/3626252.3630909

[8] 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 (Jan. 2024), 56–67. https://doi.org/10.1145/3624720

[9] Jacob Doughty, Zipiao Wan, Anishka Bompelli, Jubahed Qayum, Taozhi Wang, et al. 2024. A comparative study of AI-generated (GPT-4) and human-crafted MCQs in programming education. In *Proceedings of the 26th Australasian Computing Education Conference*. 114–123.

[10] Yuemeng Du, Andrew Luxton-Reilly, and Paul Denny. 2020. A Review of Research on Parsons Problems. In *Proceedings of the Twenty-Second Australasian Computing Education Conference (ACE'20)*. ACM.

[11] Barbara J Ericson, Lauren E Margulieux, and Jochen Rick. 2017. Solving parsons problems versus fixing and writing code. In *Proceedings of the 17th koli calling international conference on computing education research*. 20–29.

[12] Micaela Esteves, Bejamim Fonseca, Leonel Morgado, and Paulo Martins. 2008. Contextualization of programming learning: a virtual environment study. In *2008 38th Annual Frontiers in Education Conference*. IEEE, F2A–17.

[13] James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A. Becker. 2023. My AI Wants to Know if This Will Be on the Exam: Testing OpenAI's Codex on CS2 Programming Exercises. In *Proceedings of the 25th Australasian Computing Education Conference*. ACM.

[14] Lingyuan Gu, Rongjin HUNAG, and Ference Marton. 2004. Teaching with variation: A Chinese way of promoting effective mathematics learning. In *How Chinese learn mathematics: Perspectives from insiders*. World Scientific, 309–347.

[15] Sebastian Gutierrez, Irene Hou, Jihye Lee, Kenneth Angelikas, Owen Man, Sophia Mettille, James Prather, Paul Denny, and Stephen MacNeil. 2024. Seeing the Forest and the Trees: Solving Visual Graph and Tree Based Data Structure Problems using Large Multimodal Models. *arXiv preprint arXiv:2412.11088* (2024).

[16] Irene Hou, Owen Man, Sophia Mettille, Sebastian Gutierrez, Kenneth Angelikas, and Stephen MacNeil. 2024. More robots are coming: large multimodal models (ChatGPT) can solve visually diverse images of Parsons problems. In *Proceedings of the 26th Australasian Computing Education Conference*. 29–38.

[17] Irene Hou, Sophia Mettille, Owen Man, Zhuo Li, Cynthia Zastudil, and Stephen MacNeil. 2024. The Effects of Generative AI on Computing Students' Help-Seeking Preferences. In *Proceedings of the 26th Australasian Computing Education Conference*. ACM, 39–48.

[18] Sam Lau and Philip Guo. 2023. From "Ban It Till We Understand It" to "Resistance is Futile": How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools such as ChatGPT and GitHub Copilot. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1 (ICER '23)*. ACM, 106–121.

[19] Dastyni Loksa, Lauren Margulieux, Brett A. Becker, Michelle Craig, Paul Denny, Raymond Pettit, and James Prather. 2022. Metacognition and Self-Regulation in Programming Education: Theories and Exemplars of Use. *ACM Trans. Comput. Educ.* 22, 4, Article 39 (Sept. 2022), 31 pages. https://doi.org/10.1145/3487050

[20] Stephen MacNeil, Magdalena Rogalska, Juho Leinonen, Paul Denny, Arto Hellas, and Xandria Crosland. 2024. Synthetic Students: A Comparative Study of Bug Distribution Between Large Language Models and Computing Students. In *Proceedings of the 2024 on ACM Virtual Global Computing Education Conference V. 1 (SIGCSE Virtual 2024)*. Association for Computing Machinery, 137–143. https://doi.org/10.1145/3649165.3690100

[21] Lauren E. Margulieux, James Prather, Brent N. Reeves, Brett A. Becker, Gozde Cetin Uzun, et al. 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 (ITiCSE 2024)*. ACM, 276–282. https://doi.org/10.1145/3649217.3653621

[22] Marvin Minsky. 1997. Negative expertise. (1997).

[23] Kate Nussenbaum, Dima Amso, and Julie Markant. 2017. When increasing distraction helps learning: Distractor number and content interact in their effects on memory. *Attention, Perception, & Psychophysics* 79 (2017), 2606–2619.

[24] James Prather, Brett A Becker, Michelle Craig, Paul Denny, Dastyni Loksa, and Lauren Margulieux. 2020. What do we think we think we are doing? Metacognition and self-regulation in programming. In *Proceedings of the 2020 ACM conference on international computing education research*. 2–13.

[25] James Prather, Paul Denny, Juho Leinonen, Brett A. Becker, Ibrahim Albluwi, 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 (ITiCSE-WGR '23)*. Association for Computing Machinery. https://doi.org/10.1145/3623762.3633499

[26] James Prather, Juho Leinonen, Natalie Kiesler, Jamie Gorson Benario, Sam Lau, et al. 2024. Beyond the Hype: A Comprehensive Review of Current Trends in Generative AI Research, Teaching Practices, and Tools. arXiv:2412.14732 [cs.CY] https://arxiv.org/abs/2412.14732

[27] James Prather, Brent N. Reeves, Paul Denny, Brett A. Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. "It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. *ACM Trans. Comput.-Hum. Interact.* 31, 1, Article 4 (Nov. 2023), 31 pages. https://doi.org/10.1145/3617367

[28] 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.

[29] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)* 18, 1 (2017), 1–24.

[30] Brent Reeves, Sami Sarsa, James Prather, Paul Denny, Brett A. Becker, Arto Hellas, Bailey Kimmel, Garrett Powell, and Juho Leinonen. 2023. Evaluating the Performance of Code Generation Models for Solving Parsons Problems With Small Prompt Variations. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2023)*. ACM.

[31] Advait Sarkar, XT Xu, N Toronto, I Drosos, and C Poelitz. 2024. When Copilot Becomes Autopilot: Generative AI's Critical Risk to Knowledge Work and a Critical Solution. *EuSpRIG Proceedings* (2024).

[32] Jaromir Savelka, Arav Agarwal, Marshall An, Chris Bogart, and Majd Sakr. 2023. 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*. ACM.

[33] Donald A Schön. 2017. *The reflective practitioner: How professionals think in action.* Routledge.

[34] David H. Smith, Seth Poulsen, Chinedu Emeka, Zihan Wu, Carl Haynes-Magyar, and Craig Zilles. 2024. Distractors Make You Pay Attention: Investigating the Learning Outcomes of Including Distractor Blocks in Parsons Problems. In *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1 (ICER '24)*. Association for Computing Machinery, 177–191.

[35] Anselm L Strauss and Juliet Corbin. 2004. Open coding. *Social research methods: A reader* (2004), 303–306.

[36] Sarah Tasneem. 2012. Critical thinking in an introductory programming course. *Journal of computing sciences in colleges* 27, 6 (2012), 81–83.

[37] Andrew Tran, Kenneth Angelikas, Egi Rama, Chiku Okechukwu, David H Smith, and Stephen MacNeil. 2023. Generating multiple choice questions for computing courses using large language models. In *2023 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–8. https://doi.org/10.1109/FIE58773.2023.10342898

[38] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems (CHI EA '22)*. Association for Computing Machinery, Article 332, 7 pages. https://doi.org/10.1145/3491101.3519665

[39] David Weintrop and Uri Wilensky. 2015. To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th International Conference on Interaction Design and Children (IDC '15)*. ACM, 199–208. https://doi.org/10.1145/2771839.2771860

[40] 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 (SIGCSE 2023)*. ACM.

[41] Cynthia Zastudil, Magdalena Rogalska, Christine Kapp, Jennifer Vaughn, and Stephen MacNeil. 2023. Generative AI in Computing Education: Perspectives of Students and Instructors. In *2023 IEEE Frontiers in Education Conference (FIE)*.