

Exploring the Design Space of Cognitive Engagement Techniques with AI-Generated Code for Enhanced Learning

Majeed Kazemitabaar
University of Toronto
Toronto, Ontario, Canada
majeed@dgp.toronto.edu

Oliver Huang
Department of Computer Science,
University of Toronto
Toronto, Ontario, Canada
oliver@dgp.toronto.edu

Sangho Suh
Department of Computer Science,
University of Toronto
Toronto, Ontario, Canada
sangho@dgp.toronto.edu

Austin Z. Henley
Carnegie Mellon University
Pittsburgh, Pennsylvania, United States
azhenley@cmu.edu

Tovi Grossman
Department of Computer Science,
University of Toronto
Toronto, Ontario, Canada
tovi@dgp.toronto.edu

Line-by-
Explanation

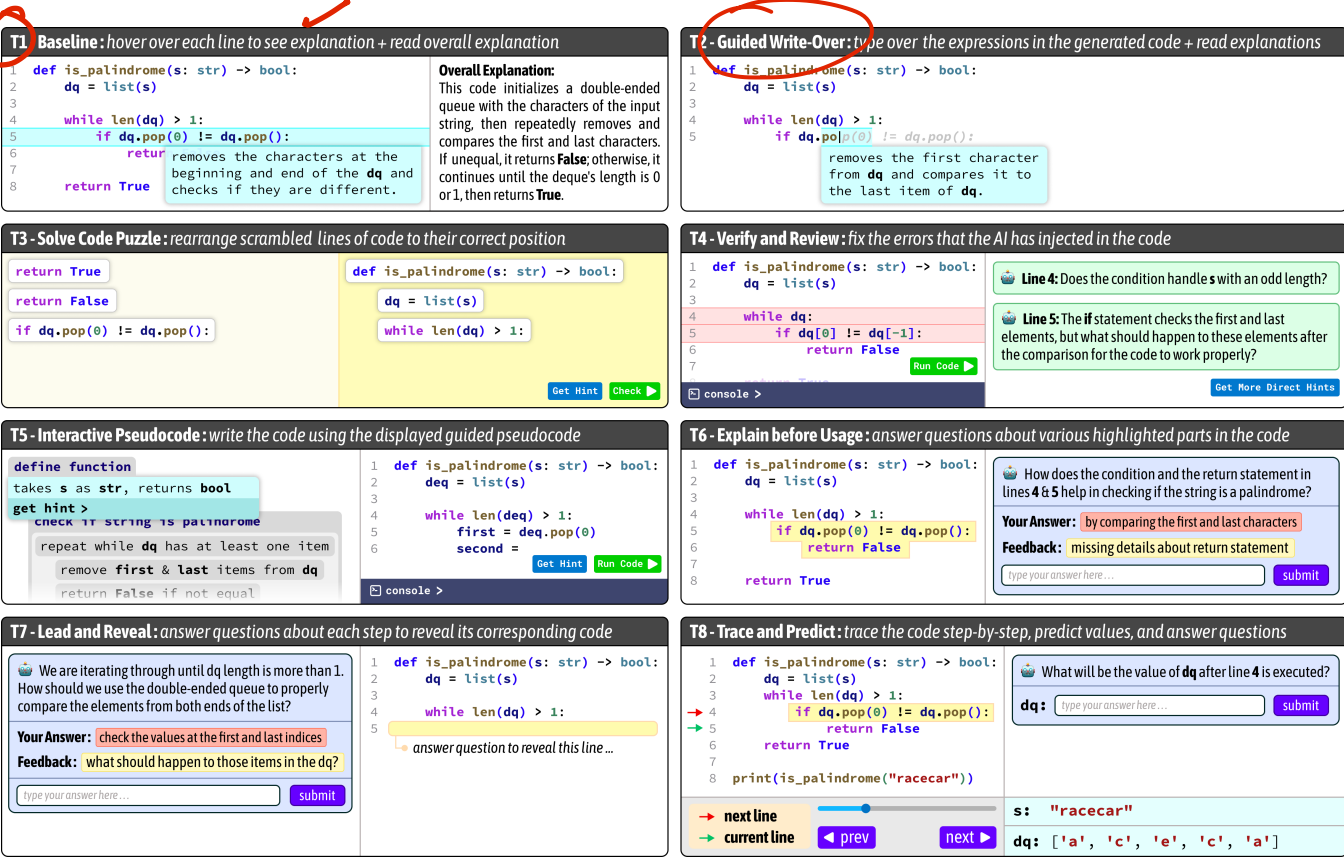


Figure 1: Illustration of the BASELINE technique, displaying the AI-generated code and a comprehensive explanation, along with seven cognitive engagement techniques requiring varying levels of user engagement either before or after revealing the AI-generated code.

ABSTRACT

Novice programmers are increasingly relying on Large Language Models (LLMs) to generate code for learning programming concepts.

However, this interaction can lead to superficial engagement, giving learners an illusion of learning and hindering skill development. To address this issue, we conducted a systematic design exploration to develop seven cognitive engagement techniques aimed at promoting deeper engagement with AI-generated code. In this paper, we describe our design process, the initial seven techniques and results



This work is licensed under a Creative Commons Attribution 4.0 International License.

from a between-subjects study (N=82). We then iteratively refined the top techniques and further evaluated them through a within-subjects study (N=42). We evaluate the friction each technique introduces, their effectiveness in helping learners apply concepts to isomorphic tasks without AI assistance, and their success in aligning learners’ perceived and actual coding abilities. Ultimately, our results highlight the most effective technique: guiding learners through the step-by-step problem-solving process, where they engage in an interactive dialog with the AI, prompting what needs to be done at each stage before the corresponding code is revealed.

CCS CONCEPTS

• **Human-centered computing** → **Natural language interfaces; Interactive systems and tools; Empirical studies in HCI.**

KEYWORDS

AI-Assisted Programming, Generative AI, Copilot, ChatGPT, Cognitive Engagement Enhancement, AI-Assisted Learning, Cognitive Forcing Functions, Task Decomposition, Learning Outcomes

1 INTRODUCTION

Novice programmers often encounter challenges as they learn to code. Initially, learners struggle with understanding fundamental programming concepts and syntax [8, 22, 58, 78, 93], and as they advance, they face further difficulties mastering skills such as algorithmic thinking and software design principles [43, 80, 94, 123].

Recently, Large Language Models (LLMs) are becoming increasingly integrated into novice programmers’ workflow, influencing their help-seeking behaviors [50], and changing the perspectives of both students and educators [71, 121]. Novice programmers now frequently use LLMs for various tasks, such as writing and explaining code, debugging, understanding concepts, and obtain example solutions [33, 50, 121].

One prominent use of LLMs is for code generation [33] as demonstrated by large-scale self-report studies [99] and analyses of student prompts [62, 100]. Learners often cite the ability to frame questions easily and obtain faster, contextually relevant solutions as key advantages of using LLMs over traditional methods like web searches [64].

However, traditionally, learners proactively searched for and found relevant code examples to address gaps in their knowledge, meaningfully engaging themselves in a learning process [10, 15, 70, 101]. But with LLMs, learners can bypass the effortful yet beneficial process of adapting generic code examples to specific contexts [55, 112]. While this shift can boost productivity, it can lower their engagement with the learning process and hinder the development of critical programming and Computational Thinking (CT) skills [117].

Currently, there are many concerns from both learners and educators about over-reliance on AI-generated code which could lead to skill degradation [71, 120, 121]. Learners might accept generated code without fully understanding it, giving them the illusion of learning [92]. This can negatively impact their ability to write, modify, or debug code independently without AI [2, 62].

To address these challenges, prior research has sought to develop new learning activities to enhance prompting and code comprehension skills [17, 20], or coding assistants that do not generate direct code solutions [74] and instead, generate pseudocode [64]. However, given the efficiency and ubiquity of LLMs, it is likely that learners will continue to use unrestricted LLMs for code generation. Therefore, to prevent skill degradation, the focus should shift toward fostering deeper cognitive engagement with AI-generated code, rather than discouraging or limiting its use. Indeed, researchers are increasingly advocating for AI tools that promote metacognitive reflection [106] as a way to augment human cognition.

Engaging more deeply with AI-generated code involves critically analyzing it, evaluating the decisions that were made to solve the problem. This process involves rejecting information that contradicts their existing assumptions and adjusting their own knowledge to incorporate new insights. To promote such behavior, we drew inspiration from the concept of in-the-moment cognitive interventions [12, 86], which have shown to enhance cognitive engagement and incidental learning [30]. Our approach is to introduce a small amount of forced engagement with the AI-generated code—which we conceptualize as “friction”—before learners can use the code.

However, this friction must be carefully designed—it should not cause frustration or be seen as an unnecessary interaction. Instead, it should be thoughtfully designed to engage learners and provide meaningful assistance to their learning by, for example, helping them identify and bridge their knowledge gaps. Ideally, it would empower learners to not only understand the AI-generated code better but also to apply learned concepts to writing, extending, or modifying similar code.

Therefore, we systematically explored a broad design space of various forms of user engagements with code (Section 3). We developed seven distinct cognitive engagement techniques, illustrated in Figure 1. Each technique introduces novel and unique forms of friction, requiring different types and levels of engagement. Another key factor is *Engagement Timing*—whether to require user engagement before or after revealing the code solution. All techniques are designed based on the capabilities of LLMs and informed by prior research about supporting novice programmers in learning to code.

- **T1 - *BASILINE*:** Users can directly use the generated code and accompanying explanation without any required engagement.
- **T2 - *GUIDED-WRITE-OVER*:** Users must type over each line of the generated code while being shown explanations about the purpose of each expression.
- **T3 - *SOLVE-CODE-PUZZLE*:** Users must reorder scrambled lines of AI-generated code into their correct horizontal and vertical spots.
- **T4 - *VERIFY-AND-REVIEW*:** AI injects errors into the generated code and users must identify and fix them with guided support.
- **T5 - *INTERACTIVE-PSEUDO-CODE*:** AI generates pseudocode of the solution that users must manually implement with guided support.
- **T6 - *EXPLAIN-BEFORE-USAGE*:** Users must answer AI-generated questions about various parts of the code.

- **T7 - *LEAD-AND-REVEAL*:** The AI guides users through solving the problem step-by-step, prompting them to explain the necessary actions and decisions for each step before revealing the corresponding line of code.
- **T8 - *TRACE-AND-PREDICT*:** Users must trace the code execution line by line with a sample input and predict value of variables at key steps.

To evaluate the effectiveness of each technique, we conducted two studies with novice programmers recruited from intermediate-level computer science programming courses. From the first study (n=82) we identified the *LEAD-AND-REVEAL* and *TRACE-AND-PREDICT* techniques as the most balanced techniques, fostering learning while eliciting just the right amount of cognitive engagement—enough to induce learning, but not so much as to overwhelm them. We then refined these two techniques based on collected feedback (Section 5). Afterwards, we conducted a second, within-subjects study (n=42). The second study evaluated an additional metric: how effectively do techniques help users align their perceived and actual ability to write code of similar complexity without AI assistance? Our results showed that the *LEAD-AND-REVEAL* technique better aligned participants’ perceived and actual ability compared to the *BASELINE* and *TRACE-AND-PREDICT* techniques, without increasing cognitive load (Section 6).

Lastly, we synthesize our results and discuss the concept of “*Friction-Induced AI*” to enhance short-term productivity gains while preventing long-term productivity loss due to over-reliance on AI.

This paper makes the following contributions:

- **Design Space for Cognitive Engagement with AI-generated Code:** seven novel AI-assisted techniques that vary in engagement level, systematically positioned within the design space (Section 3).
- **Empirical Evaluation:** In a between-subjects study (n=82), we compared the seven techniques with the *BASELINE* and identify *LEAD-AND-REVEAL* and *TRACE-AND-PREDICT* as the most effective techniques for balancing learning gains and friction: (Section 4).
- **Refinement and Validation:** A within-subjects study (n=42) with the updated *LEAD-AND-REVEAL* and *TRACE-AND-PREDICT* techniques shows *LEAD-AND-REVEAL* significantly improves alignment between perceived and actual coding ability without increasing cognitive load (Section 6).
- **Open-Source Release:** We provide the final *LEAD-AND-REVEAL* technique as a new tool for education use and future research: <https://lead-and-reveal.vercel.app>

2 RELATED WORK

This section discusses the challenges faced by novice programmers, established interventions that address these difficulties, and the evolving role of Large Language Models (LLMs) in programming education, including their potential impact on metacognitive skills and over-reliance on AI.

2.1 Challenges in Learning to Code

Despite the growing interest and accessibility, novice programmers face persistent challenges when they start learning to code

[1, 4, 11, 23, 52, 57, 103]. They must simultaneously learn conceptual knowledge, syntactic knowledge, and strategic knowledge like planning, problem-solving, and debugging [93]. End-user programmers, who write code for their own use, face six learning barriers, identified by Ko et al.: uncertainty about next steps, selecting appropriate programming constructs, combining constructs, using code construct correctly, understanding code failures, and inspecting program behavior [66], with similar mistakes and misconceptions identified by other research [1, 65, 67, 76, 79]. Furthermore, as novice programmers make progress, they face difficulties in acquiring advanced skills like algorithmic thinking and software design [43, 80, 94, 123].

To address these challenges, researchers have introduced and experimented with various pedagogical strategies and software tools. A common approach in introductory programming courses is practicing with traditional coding tasks, where students write programs from scratch. While beneficial, these tasks often lead to frustration due to difficulties with syntax and conceptual understanding.

Parsons problems were introduced as an alternative, requiring learners to arrange shuffled lines of code into the correct order, emphasizing logical reasoning over syntax and error correction [87]. Studies show that Parsons problems take less time to complete than fixing buggy code or writing code from scratch, without sacrificing learning outcomes [25]. Further research has investigated variations such as *faded* [115] and *adaptive* Parsons problems [24] with similar results. Worked examples are another alternative, which guide learners step-by-step through solving a problem, helping them understand how expert programmers approach coding tasks. They have been found particularly effective in reducing the time needed to reach proficiency [81], leading to the development of interactive versions [32]. Tracing exercises, where learners must follow the execution steps in a program, are frequently used to build understanding of programs execution and develop debugging skills [69, 77, 109].

Similarly, visualization tools allow learners to observe program execution and state changes, making abstract concepts like recursion and memory more tangible and accessible [102]. A notable example is Python Tutor, which displays visualizations of the program’s data structures at each step of the code [40]. Other tools include Omnicode, which displays a scatter plot matrix of all runtime values for every variable in the program [59], Theseus, which annotates functions in the code editor with the number of times it was called during the current execution [73], and an extension that displays a small graph of how each variable changes over time during execution [47].

Another approach involves generating content to assist learners, such as hints [31, 46, 56, 88], examples [53, 54, 85], tutorials [44], and recommendations [27, 45, 82, 104, 113]. Enhanced error messages provide more informative feedback on syntax and runtime errors, though results on their effectiveness have been mixed [3, 18, 29, 84, 89, 91].

However, as AI tools become increasingly accessible and learners rely on them for generating code solutions, they risk missing out on practicing and mastering essential higher-order cognitive skills like computational thinking [117]. To address this, our work builds on established interventions introduced through programming education research to re-engage learners with AI-generated

code. Specifically, as we further elaborate in Section 3, we draw inspiration from these prior interventions and the associated findings to develop cognitive engagement techniques designed to promote cognitive engagement, deeper learning, and avoid over-reliance on AI.

2.2 The Changing Landscape with Generative AI

The rise of generative AI, especially Large Language Models (LLMs) is transforming programming education. These models can generate code from natural language, explain code, correct errors in code, and offer code completions (e.g., Github Copilot [34]). They are now widely accessible through tools like ChatGPT and Claude AI and can serve as personalized coding assistants and tutors [42]. Recent research demonstrates that OpenAI GPT-3 outperformed students in multiple computer science exams [28] and GPT-4 passing exercises of three Python programming courses without human involvement [98]. Even multi-modal AI models can solve visual tasks like Parsons problems with high accuracy [49].

Although AI-powered tools can arguably offer exciting promises including improved engagement and personalized feedback [60], they also present challenges that limit learning opportunities. One major concern is over-reliance on AI-generated solutions [90, 92]. For novice programmers, tools like Copilot or ChatGPT provide quick answers to coding problems, which limits the opportunity for learners to critically engage with the problem and develop solution strategies [5]. Researchers are already providing empirical evidence on the effect of access to AI on learning outcomes and over-reliance. Kazemitabaar et al. conducted a controlled experiment with 69 high-school students with no prior Python experience, comparing learning outcomes between two groups: one with access to AI and one without, over seven sessions, followed by two evaluation sessions without AI. Their results showed that the AI group experienced less frustration, and improved learning outcomes for students with stronger prior conceptual skills. But a follow-up analysis revealed various types of over-reliance on AI. Particularly, when students relied on AI to solve tasks autonomously, their subsequent coding skills without AI were consistently diminished [61].

One approach to address over-reliance has been to place guardrails around AI so that it would avoid generating direct code solutions. Bastani et al. compared three groups of students, one with AI, one without, and one with an AI that safeguards learning (using prompt engineering) in a high school math class with nearly a thousand students. Their results demonstrate that while access to the AI boosts performance, students who later lose access to it perform worse than those who never had access to AI—except for those who used AI with guardrails [2]. Therefore, researchers and educators have called out for the development of pedagogical AI tools [71] which has led to the development of coding assistants like CodeAid [64] and CodeHelp [74] that avoid displaying direct code solutions. Although these solutions might help, they are common in limiting and discouraging the use of AI-generated code. However, generative AI tools are here to stay and many acknowledge that we have crossed a point of no return with AI integration in programming education, and therefore, we should embrace this new paradigm [71]. In our work, we take an alternative approach

and instead promote learners in deep, cognitive engagement with AI-generated code.

2.3 Cognitive Engagement with AI

Over-reliance on AI is a challenge that is being observed in many domains and workflows beyond learning to code. For example novice designers often take on AI suggestions without putting the effort to explore the alternatives [21, 118]. This over-reliance stems from cognitive biases and tendencies. One prominent example is automation bias, which describes the human tendency to favor suggestions from automated systems, often ignoring contradictory information from non-automated sources, even when the latter is correct. Extensive research, such as studies involving doctors using clinical decision support systems, has demonstrated the pervasiveness of this bias [35]. Another contributing factor is cognitive offloading, which suggests that people rely on external aids—in this case, AI—in order to reduce their cognitive load [95]. Together, these concepts—automation bias and cognitive offloading—form the motivation of our work, highlight the challenges in integrating AI into human workflows and the need for interventions that strike the right balance between engagement and cognitive load.

While this challenge has existed since the development of automated systems, the explosive adoption of AI in everyday life and work has amplified it into a widespread problem, affecting millions of people—not just a few experts using decision support systems. Thus, a growing number of researchers are recognizing the need for interventions that help users engage more deeply with AI. For example, Tankelevitch et al. advocated that all generative AI-powered systems should provide metacognitive support strategies, such as explainability and customizability, and suggested mechanisms for eliciting user reflections on both their own decisions and those of AI [106]. Similarly, Gajos et al. conducted three experiments with nutrition-related decisions and found that providing AI explanations without direct recommendations led to users learning more, as users had to derive their own conclusions from the explanations rather than relying on an ‘answer’ from the AI [13, 30]. Buccinca et al. tested three cognitive forcing functions—interventions that require users to explain why they accept or reject AI recommendations—and found that such interventions can help reduce over-reliance on AI. However, they also observed that people rated these designs less favorably compared to those where they might over-rely on AI, understanding again that it is important to carefully design these interventions [12].

In programming education, Kazemitabaar et al. studied the issue of learners passively accepting AI-generated code by developing CodeAid [64], which generates pseudo-code, or highlights incorrect lines with suggested fixes, rather than directly displaying code solutions. Similarly, Hou et al. developed CodeTailor [51], which responds to learners’ help requests by presenting code solutions as Parsons problems for them to solve, instead of directly providing the fixed solution. Their evaluation study showed how this approach improved learning outcomes compared to a baseline where generated code was directly provided.

Our work builds on these insights in the context of AI-assisted programming and explore new interventions that can help learners

	Learning Opportunity						Engagement Level				Timing	
	Remember	Understand	Apply	Analyze	Evaluate	Create	Passive	Active	Constructive	Interactive	Reveal & Engage	Engage & Reveal
T1 - Baseline												
T2 - Guided Write-Over												
T3 - Solve Code Puzzle												
T4 - Verify and Review												
T5 - Interactive Pseudocode												
T6 - Explain before Usage												
T7 - Lead and Reveal												
T8 - Trace and Predict												

Figure 2: Summary of cognitive engagement techniques mapped within the design space. The first set of columns (in orange) represents learning opportunities aligned with Bloom’s taxonomy. The second set (in blue) corresponds to cognitive engagement levels based on the ICAP framework. The final two columns (in purple) reflect whether the technique either reveals the code before requiring user engagement or requires user engagement before revealing the solution.

cognitively engage with AI-generated code to ensure they retain higher-order CT skills like problem-solving.

3 COGNITIVE ENGAGEMENT TECHNIQUES WITH AI-GENERATED CODE

To properly explore the design space of cognitive engagement techniques, we developed a set of initial prototypes, each with distinct types of user involvement based on the capabilities of LLMs and prior literature in supporting novices in learning to code. We then iteratively refined each technique through design probe sessions with five CS educators.

Additionally, to guide our design, we constructed a design space based on three key dimensions relevant to our context. First, Bloom’s taxonomy which classifies learning objectives by cognitive complexity [7]: *Remember*, *Understand*, *Apply*, *Analyze*, *Evaluate*, and *Create*. Specifically, we use definitions adapted for programming education [108]. Second, the ICAP framework [16], which links active learning outcomes to four categories of engagement level: *Interactive*, *Constructive*, *Active*, and *Passive*. And third, engagement timing: whether to require engagement with the AI-generated code before (*Engage&Reveal*) or after revealing the code solution (*Reveal&Engage*).

We mapped each technique within the design space according to the above three dimensions, illustrated in Figure 2. We acknowledge that some degree of subjective interpretation is inherent in this process. Our design space is not meant to imply that these techniques can fit neatly into distinct areas. Rather, it was to guide us in making intentional design decisions. Below, we describe each

technique, including the prior literature that inspired it and its design.

3.1 T1. BASELINE (Minimal | Passive | Reveal & Engage)

In the BASELINE technique, after the user provides the input prompt, the AI generates code and a detailed explanation using an initial prompt, and it is displayed to the user, similar to tools like ChatGPT. The level of engagement is completely optional and up to users. This technique induces little engagement, therefore classified as passive engagement.

3.2 T2. GUIDED-WRITE-OVER (Remember | Active | Engage & Reveal)

For this technique, we drew inspiration from prior research in computing education, which found that requiring learners to type over worked examples improves learning [32]. This technique increases learning opportunities by forcing deliberate practice [26], preventing passive copying of code without engaging one’s attention to understand it.

In our technique, ghost text for each line of line of the generated code is displayed, similar to Github Copilot [34], but it requires users to type over it. A novel aspect of our technique is that, as users type, explanations for each expression (generated by a separate LLM prompt) are shown below the expression. Additionally, a high-level description of how the line contributes to the overall solution is displayed to the right of the line. Our technique also highlights incorrectly typed characters in yellow, prompting users to retype them. After two incorrect attempts, the system moves to the next character while marking the error in red. After the user successfully types over each line, they can proceed to use the code in their editor.

This technique aligns with the "Remember" level of Bloom’s taxonomy. It prompts users to actively retrieve previously learned concepts and recall relevant programming constructs, reinforcing memory retention and prevents shallow engagement.

3.3 T3. SOLVE-CODE-PUZZLE (Analyze | Constructive | Engage & Reveal)

This technique adapts Parsons problem, which has been widely used in computing education for their demonstrated effectiveness. It requires learners to rearrange mixed code blocks to form a correct program [19, 87]. Empirical research has shown that various forms of these problems, including *faded* [115], *adaptive* [24], and *personalized* [51], can reduce cognitive load while maintaining learning performance [25].

In this technique, each line of the AI-generated code is turned into a draggable block and is mixed inside a drag-and-drop pane, and the indentation is removed in the left pane. The user has to then drag each code block from the left, and put it in the correct spot on the right, as well as its correct level of indentation (horizontal spot). After using all code blocks, they can hit the "check" button that will only tell the user whether their solution is correct or not, and if all blocks are correctly placed they can proceed to use the code in their editor. There is also a "hint" button that will highlight all blocks that are placed incorrectly, and provide arrows to which

direction each block should be moved to without explicitly telling them the exact position.

This technique is positioned at the "Analyze" level as it requires analyzing what each block does and how it contributes to the solution, then they must rearrange them, "construct" sub-programs, and understand how the blocks work together to form a functioning program.

3.4 T4. VERIFY-AND-REVIEW (Evaluate | Constructive | Engage & Reveal)

This technique draws inspiration from the use of debugging—a fundamental CT skill [117]—to help learners grasp programming concepts [36, 72, 122]. Research beyond programming shows that incorporating incorrect solutions in worked examples can enhance learning transfer for advanced learners, but may overwhelm novices unless additional support, such as error highlighting, is provided [37]. We applied these design guidelines to inform the design of the technique.

In this technique, the AI-generated code undergoes an additional LLM prompt that deliberately injects 3-4 errors, depending on the code's length and complexity. The user is then presented with this editable, incorrect code in a code editor, where they can run and debug it. The user is asked to identify and fix the injected errors. Furthermore, to assist users who might struggle with identifying the problems, clicking the "check" button highlights any remaining lines of code that needs to be fixed (using an LLM prompt). If they need help with how to fix the incorrect part, clicking on the "hint" button offers progressively more direct guidance, starting with hints about potential problems and eventually showing the correct code for each line. The user can use the editor to fix the problematic code and then check it with the "check" button. Once the code is error-free, the user can proceed to using the code in the editor.

This technique engages users at the "Evaluate" level by requiring them to actively check whether the code meets requirements or produces the correct output. Moreover, by fixing errors, they are required to critique the logic and quality of the code. They must determine which portions of the code are incorrect and fix those, while keeping the correct parts of the code.

3.5 T5. INTERACTIVE-PSEUDO-CODE (Understand | Constructive | Engage & Reveal)

We were inspired by the use of pseudocode in computing education to support algorithmic thinking before introducing complex syntax [75, 96, 107], as well as by CodeAid, an LLM-powered assistant that encourages active learning by generating pseudocode rather than providing direct code solutions [64].

This technique uses an additional LLM prompt to generate a subgoal-labeled, hierarchical pseudocode, displayed fully to the user without showing the code solution. The pseudocode is organized into subgoals, grouping several lines of code with a descriptive title. Users are provided with an empty code editor beside the pseudocode to write the corresponding code. Two buttons—"check" and "review"—use an LLM prompt to help guide the user by checking their code and offering feedback if errors are found. Initially, only subgoal titles are shown, with an "expand" button to reveal the

pseudocode. Hovering over each pseudocode line provides a detailed explanation, and users can access two progressive levels of hints: syntactical guidance, followed by revealing the actual Python code. After successfully writing the code and verifying with the "check" button, users can proceed to using the code in their editor.

This technique reaches the "Understand" level as it helps them interpret the underlying logic and requires them to translate an abstract representation of the problem to concrete code, in a constructive manner.

3.6 T6. EXPLAIN-BEFORE-USAGE (Understand | Interactive | Reveal & Engage)

This technique is motivated by prior work on self-explanation, where learners were encouraged to explain the code. Researchers found that this can be a successful intervention, increasing engagement with worked examples [110], and that the ability to explain code has a significant relationship with the ability to write code [77, 83, 109].

In this technique, after the AI generates the code, it is immediately displayed to the user. Immediately afterwards, an important part of the code (determined by a second LLM prompt) is then highlighted, followed by a short-answer question for the user to answer. The user's response is then evaluated via another LLM prompt with feedback and a score from 0 to 5, based on accuracy and completeness. The user has up to 3 attempts for each question, with the correct answer automatically shown after the third try. A total of 3-5 questions are asked from various parts of the generated code, depending on the code's complexity, after which the user can proceed to using the code in the editor.

This technique also reaches the "Understand" level as it engages users to interpret, explain, and infer meaning from existing code, without moving towards higher cognitive processes. However, it does this interactively, as users interact with AI to answer and receive feedback on their explanations.

3.7 T7. LEAD-AND-REVEAL (Create | Interactive | Engage & Reveal)

This technique draws inspiration from CS Unplugged activities [6], which teach core concepts without any coding, and have been shown to improve computational thinking skills [9]. We integrate this with scaffolded self-explanation using Socratic questioning, which has proven to enhance coding comprehension [105].

After the code is generated, it is not displayed to the user. Instead, a second prompt is used to generate Socratic multiple-choice questions which are then displayed on one side of the interface. These questions scaffold the step-by-step process of solving the task and prompting the user in "what needs to be done at each step of the algorithm/solution" in a structured format. After the user picks the correct answer (where they have up to three retries), its corresponding line of code (which is the next line) will be displayed on the right with an explanation about how their answer and that line contributes to the overall solution. As each question is answered, its corresponding line of code will be progressively revealed on the right of the interface. This process is continued until all lines of the generated code is revealed and then the user can proceed to use the code in their editor.

This technique reaches the “Create” level as it engages users in creating a novel solution. They are prompted to interactively engage with the AI in hypothesizing and determining the next step in an algorithm. Since the code is not initially displayed, they are required to predict the logic, which stimulates creative thinking.

3.8 T8. TRACE-AND-PREDICT (Apply | Constructive | Reveal & Engage)

This technique is inspired by research showing that visualizing the notional machine is an effective method for learning programming concepts [41, 102]. Particularly, several studies have found a positive correlation between learners’ performance on code tracing tasks and their code writing tasks [69, 77, 109]. Based on these findings, we aimed to strengthen learners’ understanding by engaging them in code tracing tasks with integrated questions to sustain their engagement.

In this technique, the AI-generated code is displayed to the user with a sample input for execution. The user is then required to trace the code in a debugger-like system that includes buttons to step forward or backward through the code execution, and real-time variable values value of variables as they change. At certain complex parts of the code (identified by an LLM prompt), the technique does not allow the user to proceed. It requires the user to predict the value of a variable after that line of code is executed. The user can retry each variable prediction question three times and on the third try, the answer is displayed. This process is continued until several important parts of the code are covered with value prediction questions. After the user finishes tracing the entire code, and answering all the prediction questions along the way, they can then proceed to using the code in their editor.

Lastly, this technique reaches the “Apply” level as it requires users to examine the control flow, understand variable states, and identify how code executes line by line. Users have to actively solve questions about unfamiliar code, applying the rules of execution in a guided environment.

4 STUDY 1: COMPARING ALL TECHNIQUES

We conducted a between-subjects study with 82 participants to evaluate each technique against the BASELINE technique without forced cognitive engagement. Our goal was to evaluate:

- **RQ1 [Performance]:** How effective are the cognitive engagement techniques in supporting participants’ ability to transfer learned programming concepts to isomorphic coding tasks without AI assistance?
- **RQ2 [Friction]:** How do the techniques impact perceived friction, measured through time on task and cognitive load?
- **RQ3 [Perceptions]:** What are participants’ perceptions about technique and the type of user involvement that they require?

4.1 Experiment Tool Design

To evaluate the effectiveness of our techniques, we built a web-based application that allows users to log in and complete AI-assisted or manual coding tasks. This also enabled specific tasks to be assigned to each user by the experimenter. The tool was designed to provide a self-paced and consistent experiment for all

participants. In AI-assisted tasks, users interact with a chatbot to generate code. To avoid generating incorrect or unrelated code, the chatbot compared the user’s prompt with the current task and provided feedback if details were missing. Additionally, to ensure consistent outputs across participants, pre-generated responses for each task and technique were used. In the BASELINE technique, code is immediately shown with an accompanying explanation. For other techniques, a modal that covers the entire screen displays the technique after a 5-second delay to simulate generation time. Once users finish interacting with the technique, the modal disappears and users can see the code and its explanation.

The tool is developed using TypeScript with a client-server architecture. The server, implemented in Node.js, stores user data and logs using a MongoDB database, and interacts with OpenAI APIs (GPT-4). It parses LLM outputs and communicates with a Python shell and language server for running Python code and providing real-time autocomplete and error-detection for the client-side code editor. The Monaco code editor is used for code editing, execution, and submission. The user interfaces can be seen in 1. Full details, including prompts and source code, are available on GitHub: <https://github.com/MajeedKazemi/code-engagement-techniques>.

4.2 Methodology

The study design posed two primary challenges: (1) ensuring participants had similar backgrounds to minimize variance in prior knowledge, and (2) designing programming tasks that would uniformly challenge participants, requiring assistance and enabling both learning and the transfer of learned concepts to isomorphic tasks. The following sections outline the methodology used to address these challenges. An overview of the study procedure is displayed in Figure 3.

4.2.1 Participants. We recruited 82 participants (44 male, 37 female, 1 non-binary), ages 18–23 ($M = 19$), from an undergraduate programming course about data structures at a large public university. To ensure a consistent level of knowledge, all participants were recruited from the same class. Participants reported their frequency of using AI tools like ChatGPT for programming as daily ($n=14$), weekly ($n=33$), monthly ($n=8$), rarely ($n=22$), and never ($n=5$). Additionally, they reported using ChatGPT primarily for fixing code ($n=59$), explaining concepts ($n=54$), providing code snippet explanations ($n=47$), and generating code from descriptions ($n=32$). The study was approved at our institutes ethics review board, and informed consent was obtained from all participants prior to the study. Each participant received \$25 as compensation.

4.2.2 Study Procedure. As shown in Fig. 3, our study procedure consisted of three phases: pre-test, training, and evaluation. We describe each phase in detail below.

- **Phase 1. [Pre-Test] 5 Code-Tracing Questions.** All participants initially attended an online 30-minute session over MS Teams to perform a pre-test, including five multiple-choice code tracing questions about the stack and queue data structures. The scores were then used to create eight balanced groups, each being randomly assigned to one of the seven experimental or the BASELINE techniques. The mean pre-test scores ranged from 43.5% to 49.0%,

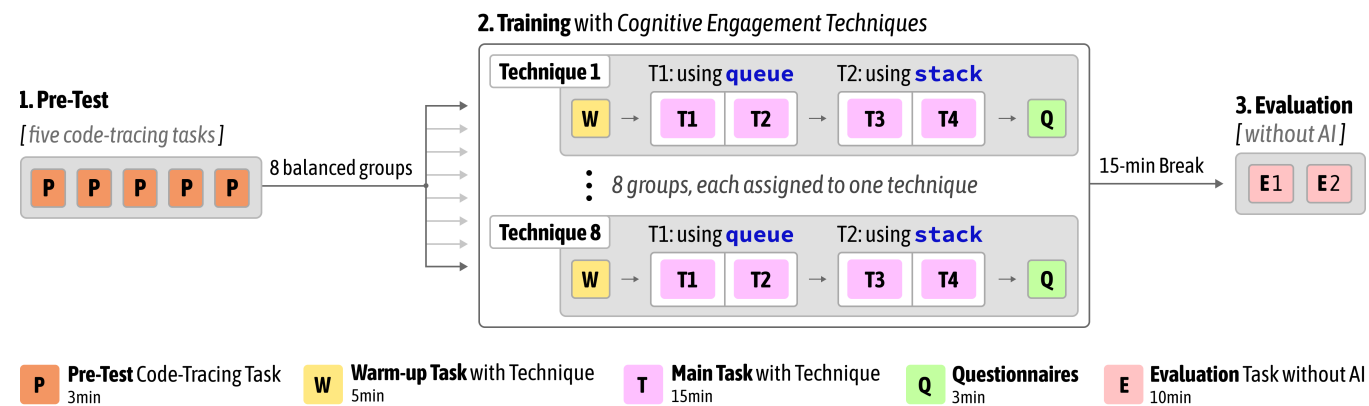


Figure 3: Overview of Study 1 procedure: Participants began with a pre-test, were split into 8 balanced groups, trained with AI using their assigned cognitive engagement technique, completed post-training questionnaires, took a short break, and then performed manual coding tasks for evaluation.

with a standard deviation of 2.9% across the group means. Each group had 10 participants, except for the *VERIFY-AND-REVIEW* and *BASELINE* groups, which included 11 participants.

- **Phase 2. [Training] 4 Tasks with Assigned Cognitive Engagement Technique.** Participants then attended an in-person 2-hour session (including a 10-minute break) for the main study in which they used the web-based experiment tool. Participants logged in with their given credentials, where they were automatically assigned to the technique determined in advance. They then were instructed about how to use their assigned technique, followed by a warm-up task using the technique, and four training tasks with their technique on slightly complex tasks: two using stacks, and two using queues.

The tasks were designed to be slightly outside of participants' zone of proximal development [111]. We consulted with the instructor of the course from which participants were recruited to ensure that the tasks were useful for their learning, were sufficiently challenging to solve independently, and could potentially be learned through sufficient, deep engagement with the code solution.

Each task in our experiment tool first displayed the task description, several test-cases, and a multiple-choice question that asked participants how well they understand the task as a way to prompt them and to make sure they understood the task before starting to work on it. While working on the AI-assisted tasks, participants were given the option to either copy the task description as provided or rephrase it in their own words. Additionally, since engaging with the AI-generated code beyond the requirements of each technique was optional, participants were advised to move on to the next task once they felt confident in their understanding, without exceeding 15 minutes per task.

- **Phase 3. [Evaluation] Survey + 2 Coding Tasks without AI Assistance.** After completing the four training tasks, participants were asked to first fill out a survey, followed by working on two manual coding tasks without AI assistance.

The survey included questions about perceived learning after using each technique, their perceived task load evaluated using the NASA Task Load Index (TLX) [68], and several Likert-questions about their willingness to use the techniques, and open-ended questions about what they liked and disliked about the process of using the technique that they were assigned to.

Participants then proceeded to work on two evaluation tasks, one being isomorphic to the second task about stacks and the other being isomorphic to the fourth training task on queues, with minimal changes in task requirements. No starter code was provided for the evaluation tasks. A timer was displayed on top of these tasks and participants were instructed to skip the task if they were not able to finish the task within 20 minutes.

4.2.3 Data Analysis. Learners' responses to the manual coding tasks were graded using two detailed rubrics, each tailored to one of the two tasks. These rubrics focused on key concepts and the use of data structures and how they were used in the training tasks. After the first author tested and refined the rubric on 25% of the responses, it was applied to all 82 submitted codes for each task by the first author. For each of the training tasks we also collected the time they spent on each task, which could be an indicator of how much friction the technique caused.

For statistical analysis, we used a generalized linear model (GLM) to examine the effect of the intervention techniques on learners' evaluation scores while controlling for pre-test scores. One-way ANOVA is used for normally distributed data with equal variances, while the Kruskal-Wallis H test is the non-parametric alternative for non-normal data. For pairwise comparisons, an independent t-test is applied for normally distributed data, and the Mann-Whitney U test for non-normal data. In this study, we focused on pairwise comparisons between each of the seven experimental techniques and the *BASELINE* technique, yielding a total of seven comparisons. To control for the risk of Type I error due to multiple-comparisons, we applied a Bonferroni correction, setting the threshold for statistical significance at $\alpha = \frac{0.05}{7} = 0.007$.

4.3 RQ1 [Performance] How effective are the cognitive engagement techniques?

Regarding the effect of the techniques on learners' ability to write code manually without AI assistance, the generalized linear model explained 17.7% of the variance in post-test manual coding scores ($Pseudo R^2 = 0.177$). Pre-test scores were a significant predictor of post-test performance ($b = 0.34$, $SE = 0.12$, $z = 2.78$, $p = .005$), indicating that higher pre-test scores were associated with higher post-test scores. Compared to the BASELINE technique ($M = 31.5$, $SD = 28.2$) which involved no forced engagement and demonstrated nearly the lowest mean performance, the LEAD-AND-REVEAL technique ($M = 58.3$, $SD = 39.8$) demonstrated the greatest improvement ($b = 26.8$, $p = .058$), though this did not reach statistical significance. The second most effective technique was SOLVE-CODE-PUZZLE ($M = 54.1$, $SD = 38.2$; $b = 22.8$, $p = .108$), followed closely by TRACE-AND-PREDICT ($M = 54.1$, $SD = 37.9$; $b = 21.8$, $p = .123$).

A post-hoc power analysis ($\eta^2 = 0.112$) showed the study was underpowered with 82 participants. To reach 90% power, a sample size of 153 would be required, suggesting the non-significant findings may be due to insufficient power. Future studies should aim for a larger sample.

4.4 RQ2 [Friction] How do the techniques impact perceived friction?

To compare participants' perceived friction across techniques, we analyzed both task completion time and NASA Task Load Index (TLX) scores. A Kruskal-Wallis H test was used to examine differences in the six TLX dimensions across the eight techniques. Significant differences were found in frustration ($\chi^2(2, N = 82) = 19.5$, $p = .006$), physical demand ($\chi^2(2, N = 82) = 16.0$, $p = .025$), and effort ($\chi^2(2, N = 82) = 14.1$, $p = .05$). A Mann-Whitney U test showed that VERIFY-AND-REVIEW ($M = 72.7$, $SD = 19.6$; $p = .035$) had significantly higher frustration compared to the BASELINE ($M = 46.7$, $SD = 27.9$). Higher physical demand was reported for INTERACTIVE-PSEUDO-CODE ($M = 60.0$, $SD = 19.9$; $p = .005$) and GUIDED-WRITE-OVER ($M = 65.7$, $SD = 28.7$; $p = .015$) compared to the BASELINE ($M = 36.4$, $SD = 14.8$).

Task completion time was analyzed using a one-way ANOVA, which revealed significant differences ($F(7, 74) = 3.64$, $p = .002$). The BASELINE had the shortest time ($M = 596s$, $SD = 268s$), while SOLVE-CODE-PUZZLE ($M = 1055s$, $SD = 272s$; $t(19) = 3.89$, $p < .001$, $d = 1.7$) and GUIDED-WRITE-OVER ($M = 1033s$, $SD = 280s$; $t(19) = 3.64$, $p = .001$, $d = 1.6$) took the longest. LEAD-AND-REVEAL ($M = 914s$, $SD = 347s$) was closest to the BASELINE, with the smallest effect size difference ($t(19) = 2.36$, $p = .029$, $d = 1.0$).

Lastly, the number of times that participants tested and executed the AI-generated code after they finished the technique was statistically significant, indicated by a Kruskal-Wallis H test ($\chi^2(2, N = 82) = 23.16$, $p = .001$). The BASELINE technique had the most number of runs ($M = 3.9$, $SD = 4.7$), while the INTERACTIVE-PSEUDO-CODE ($M = 0.3$, $SD = 0.6$), followed by TRACE-AND-PREDICT ($M = 0.6$, $SD = 0.7$), and VERIFY-AND-REVIEW ($M = 1.0$, $SD = 1.6$) had the least. This could be explained by how the user has already executed the code and checked the output in the techniques before using the code in the editor.

4.5 RQ3 [Perceptions] What are participants' perceptions of the techniques?

Below, we summarize participants' ($P1_{T1}$ – $P10_{T8}$) perceptions of each technique based on their responses to open-ended questions about their likes, dislikes, and the impact on their understanding and engagement with AI-generated code.

4.5.1 T1: BASELINE. Several participants acknowledged that the AI-generated code and explanations helped them grasp core concepts, with one noting that it facilitated understanding of the “main concepts” ($P7_{T1}$) and another stating it helped “figure out the logic [they] needed” ($P3_{T1}$). However, other participants indicated that this approach provided “only a basic idea of how the code works” ($P11_{T1}$). The majority emphasized that having the AI provide solutions directly did not foster meaningful learning. As $P10_{T1}$ remarked, “directly giving [them] the answers was not helpful for learning,” a sentiment echoed by others ($P1_{T1}$, $P2_{T1}$, $P5_{T1}$, $P6_{T1}$). For instance, $P5_{T1}$ reflected, “without actually interacting much with the code the AI had written, I wasn't able to apply it.” This feedback underscores the limited efficacy of passive AI-generated content for promoting deeper engagement and independent code-writing skills.

4.5.2 T2: GUIDED-WRITE-OVER. Participants generally appreciated the GUIDED-WRITE-OVER technique for its ability to simulate the coding process, with several noting that it felt “similar to writing the code [themselves]” ($P8_{T2}$) and encouraged them to “pay attention to the small details” ($P6_{T2}$). They found the inline explanations beneficial for learning, particularly in understanding the purpose of individual expressions ($P4_{T2}$, $P6_{T2}$, $P9_{T2}$). However, some participants found the approach cumbersome, with one stating that it was “tedious and slowed down [their] working pace” ($P7_{T2}$) and others describing it “simple and repetitive” ($P10_{T2}$), and even “extremely annoying” ($P6_{T2}$). A recurring critique was that the explanations focused too heavily on *what* the code did, neglecting the *why* behind the solution in the broader context of problem-solving ($P3_{T2}$, $P8_{T2}$). Furthermore, there were concerns that the technique limited independent problem-solving, with one participant noting that it “forces the user to follow [the AI's] algorithm, which may result in less practice in critical thinking and problem-solving” ($P6_{T2}$), while another mentioned it “did not address discrepancies” between their own expected approach and the AI-generated code ($P7_{T2}$).

4.5.3 T3: SOLVE-CODE-PUZZLE. Participants expressed that the process of actively assembling code fragments fostered deeper critical engagement with code structure and logic, with several noting it as “a great learning tool” ($P2_{T3}$) that “forces you to have some understanding of how the solution works” ($P4_{T3}$). Many appreciated the cognitive challenge, observing that it “asks you to actually solve the problem” ($P1_{T3}$) and “think about how the function should be implemented” ($P7_{T3}$). Some valued the structured breakdown of the task, which “reduced stress” and made the process feel “more interesting, like playing with puzzles” ($P5_{T3}$). However, the task also elicited mixed reactions, with a few participants describing it as “mentally exhausting” ($P6_{T3}$) and “overwhelming” ($P9_{T3}$), particularly when struggling to comprehend the purpose of individual code lines. This led to feelings of being “mentally tired” ($P6_{T3}$) and finding the task “difficult to understand and learn” ($P5_{T3}$). The rigid

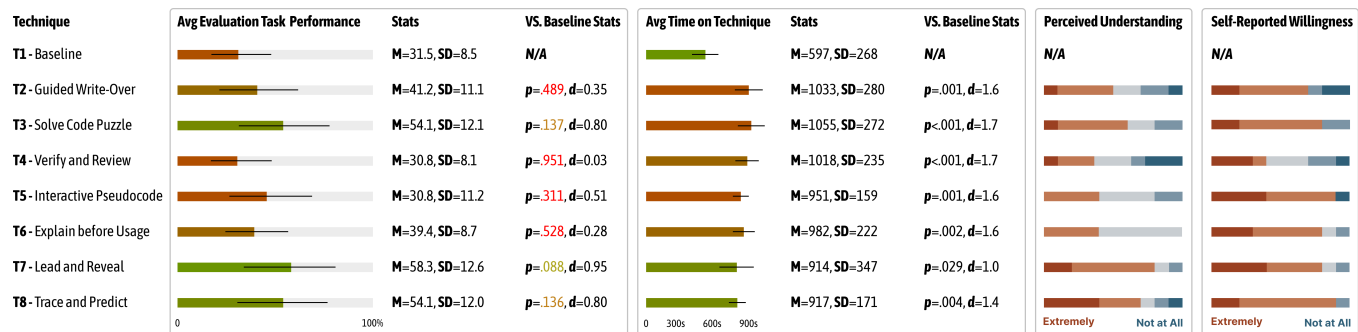


Figure 4: An overview of results from Study 1 (left to right): the performance on the two manual coding tasks. Average time spent on each technique during the training. Perceived understanding and reported willingness to use each technique (on 5-point Likert scales).

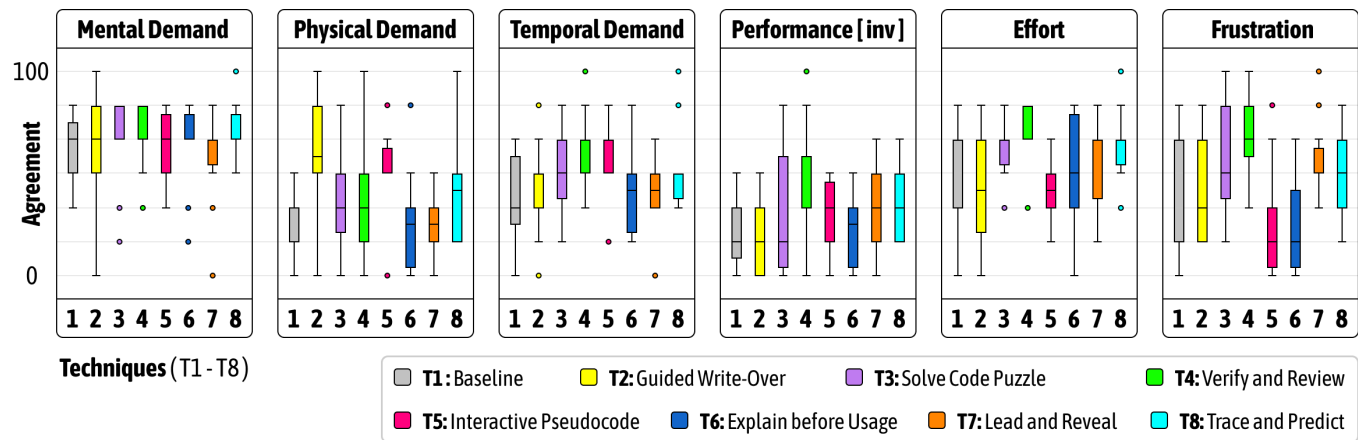


Figure 5: Cognitive load results for 8 techniques. Each box-plot displays the six dimensions of the raw NASA TLX (mental, physical, temporal demand, inverted performance, effort, and frustration) reported by participants after using each the techniques for four coding tasks. Each technique is represented by a different color, with higher values indicating higher task load.

enforcement of code ordering further exacerbated frustration for some, as it “required some of the code to be in an exact order even though it did not matter in some cases” ($P7_{T3}$).

4.5.4 T4: VERIFY-AND-REVIEW. The technique elicited mixed reactions from participants, with several highlighting both its strengths and limitations. Some participants appreciated the active engagement required, with $P1$ noting that it “forced me to manually process the code,” and $P6_{T4}$ emphasizing that the progressive hint system “contributed to my learning as it didn’t give me the answer straight away.” However, this benefit was tempered by concerns over the hint system fostering dependency, with $P9_{T4}$ remarking that the hints became “straight-up corrections” and $P10_{T4}$ explaining that this led to a lack of cognitive effort because they “don’t need to think.” $P7_{T4}$ saw value in the technique’s ability to “enhance the ability to understand code written by others” as it encouraged [them] “to think about the kinds of mistakes that might occur in the code,” yet this was counterbalanced by a more prevalent issue: the injected errors often caused confusion rather than promoting

deeper learning, as expressed by $P2_{T4}$, $P3_{T4}$, $P4_{T4}$, $P5_{T4}$, $P7_{T4}$, and $P11_{T4}$. Additionally, the lack of debugging features in the code editor was a significant frustration for some participants, with $P1_{T4}$ noting that such features are “integral to [their] understanding of code.” Overall, while the technique showed promise in fostering active code comprehension, it also introduced usability challenges that hindered its effectiveness.

4.5.5 T5: INTERACTIVE-PSEUDO-CODE. Participants expressed mixed reactions to the hierarchical pseudocode technique. Several appreciated how it encouraged problem-solving and independent thinking, with $P1_{T5}$ noting it pushed them “to think the solution by [them]self,” and $P2_{T5}$ appreciating the “chance to try [them]self and implement the code based on the skeleton.” The multi-level hint system was also praised, as $P6_{T5}$ valued “how it doesn’t immediately reveal the answer.” However, $P8_{T5}$ felt they were “just following instructions” rather than truly creating code. While $P7_{T5}$ found the pseudocode helpful for “break[ing] down a complicated problem into a digestible manner,” others ($P2_{T5}$, $P9_{T5}$, $P10_{T5}$)

found it unclear, especially for complex tasks. This highlights both the strengths and limitations of the approach in facilitating learning with AI-generated code.

4.5.6 T6: EXPLAIN-BEFORE-USAGE. Participants generally appreciated the technique’s ability to enhance engagement and deepen their understanding of AI-generated code. Many noted that it “helped [them] understand each crucial step in the code” ($P1_{T6}$), “internalize the purpose of each line” ($P7_{T6}$), and “prompted [them] to think deeper” ($P5_{T6}$, $P6_{T6}$), while also being intellectually challenging ($P5_{T6}$). However, some expressed concerns that the technique “only managed to explore surface level knowledge” ($P4_{T6}$) and “did not help [them] understand the code as a whole, just little bits of it” ($P4_{T6}$), particularly when it “would ask about a line of code whose purpose is rather straightforward” ($P9_{T6}$). Participants appreciated the feedback provided, as it “could point out my mistakes” ($P1_{T6}$) and helped them identify “why [their] logic was flawed” ($P7_{T6}$), though some desired more personalized, detailed feedback rather than “just showing them the correct answer” ($P5_{T6}$).

4.5.7 T7: LEAD-AND-REVEAL. Learners appreciated that the technique encouraged them to “think of the purpose of each line of code” ($P3_{T7}$) and helped them “get [their] own idea of what needs to be done to solve it” ($P5_{T7}$), with its sequential approach enabling them to “break down the task into pieces” ($P6_{T7}$). This scaffolded process guided learners to “follow the correct thinking process of the code step by step” ($P7_{T7}$). However, some participants found certain questions lacking “comprehensive explanations and context” ($P2_{T7}$), leading to confusion as they had to “decipher why that approach was used” ($P5_{T7}$), which at times “altered [their] expectations for what [they] needed to do to solve the problem” ($P1_{T7}$) when they had a different solution in mind. This resulted in participants like $P10_{T7}$ feeling they were “constantly guessing what the AI was doing.” Several participants ($P2_{T7}$, $P4_{T7}$, $P7_{T7}$) suggested that providing hints or explanations after incorrect answers would be beneficial in alleviating these challenges.

4.5.8 T8: TRACE-AND-PREDICT. Participants generally found the technique beneficial for fostering deeper engagement with AI-generated code. Many highlighted that tracing variable values “helped [them] slow down and think more about the code” ($P4_{T8}$), promoting a clearer understanding of “each line of the code” ($P2_{T8}$) and “how the code works” ($P5_{T8}$). The prediction questions encouraged users to engage critically, enhancing comprehension of “concepts and strategies used to solve problems” ($P3_{T8}$) and “forcing [them] to think about what was happening” ($P8_{T8}$), contributing to their learning ($P10_{T8}$). However, the technique’s step-by-step process was seen as overly time-consuming and “strenuous” ($P10_{T8}$), with users expressing frustration over “a lot of steps and questions” ($P3_{T8}$) and the lack of a “skip button” ($P3_{T8}$), even for straightforward sections. Participants also disliked the absence of explanations for incorrect predictions or code design choices, stating that “the system should have shown some explanation” ($P10_{T8}$) and that there were “no explanations as to why the code was designed in that way” ($P1_{T8}$).

4.6 Summary of Results

Overall, the LEAD-AND-REVEAL, TRACE-AND-PREDICT, and SOLVE-CODE-PUZZLE techniques emerged as the most effective, showing improvements in learners’ manual coding performance, although these differences did not reach statistical significance. Both LEAD-AND-REVEAL and TRACE-AND-PREDICT had the highest performance overall, without imposing a meaningful task load and having the closest completion time to the BASELINE. Conversely, techniques such as VERIFY-AND-REVIEW and GUIDED-WRITE-OVER induced higher levels of frustration and physical demand, correlating with lower performance and perceived learning. While INTERACTIVE-PSEUDO-CODE and EXPLAIN-BEFORE-USAGE showed some cognitive benefits, they were also seen as cognitively taxing and limited in promoting deeper problem-solving.

These results underscore a critical trade-off within the *Engagement Timing* spectrum, where the final code solution is either revealed before user engagement or withheld until after engagement. When solutions are presented upfront, as in TRACE-AND-PREDICT or EXPLAIN-BEFORE-USAGE, users may feel overwhelmed, but this approach offers a comprehensive view of how each line of code contributes to the overall solution, potentially enhancing comprehension. Interestingly, techniques from both ends of the engagement spectrum—LEAD-AND-REVEAL and TRACE-AND-PREDICT—showed similar improvements over the BASELINE without causing excessive friction. This indicates that these contrasting approaches are equally valuable and warrant further investigation.

5 FINAL DESIGN ITERATION

To further explore the design space, we iterated on the LEAD-AND-REVEAL and TRACE-AND-PREDICT techniques based on participant feedback from Study 1. We focused on these techniques as they effectively balanced performance and imposed friction. Additionally, We updated all techniques, including the BASELINE, to provide line-by-line explanations on hover, inspired by [119]. Below, we describe the updates to the two techniques.

5.1 LEAD-AND-REVEAL (V2)

Instead of using multiple-choice questions, we updated the technique to ask short-answer questions about the functionality of the next step, requiring users to provide natural language descriptions. Each guiding question is now preceded by extensive context, to properly guide the user to think about the next step. Additionally, users receive feedback via an LLM prompt that checks their response, offering hints or pointing out missing detail. We also added **on-hover explanations for each line of revealed code.**

We improved question quality by enhancing the LLM prompt with chain-of-thought prompting techniques [114]. The prompt first decomposes the code solution into a hierarchical JSON, explaining decisions at the subgoal level to each line of code. We found that this additional step significantly improved the quality of the guiding questions.

5.2 TRACE-AND-PREDICT (V2)

Previously, value prediction questions focused on single lines of code, which participants found too easy. The technique now asks about blocks of code, such as loops, conditionals, or multiple lines

with similar goal. Additionally, users may need to predict up to two variables to address more complex scenarios. The technique is also updated to include on-hover explanations for each line.

Additionally, we merged this technique with the technique from EXPLAIN-BEFORE-USAGE. After predicting variable values, users are prompted to explain the purpose of the highlighted code. This was intended to match the new difficulty level of LEAD-AND-REVEAL (V2), which now requires explanations for each line rather than multiple-choice responses.

6 STUDY 2: PRE VS. POST ENGAGEMENT

Building upon Study 1, we designed Study 2 to provide a deeper examination of the differences between the two updated techniques. While Study 1 had participants engage with only one technique, Study 2 adopted a within-subjects design to allow for direct comparison across techniques. Our objective was to evaluate:

- **RQ1 [Performance]:** How effectively do the updated cognitive engagement techniques affect the transfer of learned programming concepts to isomorphic coding tasks without AI assistance?
- **RQ2 [Friction]:** What level of friction do the updated techniques introduce, as measured by task completion time and perceived cognitive load?
- **RQ3 [Metacognitive Self-Assessment]:** How do the techniques impact participants' ability to accurately assess their own performance compared to actual task performance?
- **RQ4 [Perceptions]:** How do participants perceive the usability and effectiveness of each technique, and how willing they are to adopt these techniques in future programming tasks?

6.1 Methodology

This study used the same experiment tool used in the previous study (Section 4.1). A key challenge in designing this study was to increase the sample size in order to detect a potential effect, based on the power analysis conducted after Study 1 (Section 4.3). The within-subjects design would allow us to collect data on each technique from every participant. However, managing the study's duration posed another challenge. We aimed to ensure that participants had sufficient training and exposure to each technique to observe an effect, while minimizing learning effects and preventing fatigue. The following sections describe how these challenges were addressed.

6.1.1 Participants. We recruited 42 participants (27 men, 15 women, 0 non-binary) from a similar population of Study 1 (students taking a data structures class), but at a different semester. Participants were pre-screened to ensure they were comfortable reading and writing in English, confident in their python programming skills, and eager to learn how to design complex algorithms. Informed consent was obtained before the study began, and all participants received \$50.

6.2 Study Procedure

As shown in Fig. 6, this study consisted of two phases: training with cognitive engagement techniques and evaluation, which we explain in detail below.

- **Phase 1. [Training] 3 Tasks with LEAD-AND-REVEAL, TRACE-AND-PREDICT, and BASELINE.** Unlike Study 1, this study did not include a pre-test as all participants experienced using the updated version of all three techniques: LEAD-AND-REVEAL, TRACE-AND-PREDICT, and BASELINE. The study lasted 2.5 hours, including a 15-minute break, and was conducted remotely, using MS Teams. All participants were instructed to join from a quiet and comfortable place, and share their screens to the experimenter for moderation and resolving any potential issues. Similar to the initial study, the experiment tool provided the same self-paced experience for all participants. The process started with watching a video that explained the study procedure. The training tasks with the three cognitive engagement techniques were grouped into three topics: stacks, queues, and double-ended queues (after careful consideration with the course instructor). Each topic included two tasks, a warm-up task and a training task. Topics were presented at the same order, while the techniques for which they used each topic were fully counter-balanced between the participants to reduce order effect. The assigned order of techniques for each participant was included in each participant's account information. For each topic, participants first watched a video that introduced it to the technique for which they were assigned to, followed by working on a warm-up task for 5 minutes, and then the main task of that topic for 15 minutes. Similar to the first study, before each coding task participants were given the task description and test-cases and were asked to check how confident they were in understanding the task before proceeding to working on the task with the AI and the assigned technique. After using the cognitive engagement technique to work on the task, and after finishing the main training task for each topic, participants were given the NASA Task Load Index (TLX) questionnaire, followed a 5-point Likert-question that asked how confident they are in their ability to independently write, modify, or extend code of similar complexity of the task that they just worked on without AI. This value was later used to calculate the correlation between their perceived and actual performance. Participants were then asked to take a short break, before proceeding to working on the evaluation tasks.
- **Phase 2. [Evaluation] 3 Topics \times 2 Coding Tasks + Survey.** The evaluation tasks included three topics, each with two coding tasks. Tasks within each topic were designed to require applying the concepts used in the AI-generated code solution of their corresponding training tasks. To ensure ecological validity and simulate a task that would happen in reality, the tasks were designed to require participants to manually extend code of similar complexity to the tasks they were trained on with AI. Therefore, we designed six fill-in-the-blank tasks, in which a starter code was provided, and three to five important lines of code within each task was erased and were asked by participants to solve manually, without AI assistance. A timer was displayed on top of each evaluation task and participants were asked to skip the task after 10 minutes.

1. Training with Cognitive Engagement Techniques

[same order of tasks] + [fully counterbalanced techniques]

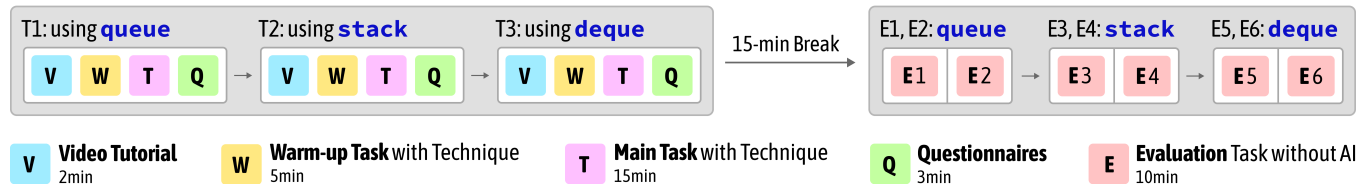


Figure 6: Study 2 Procedure Overview: Participants used each technique to learn complex coding tasks about three topics: queue, stack, and double-ended queue (deque). Each topic involved four steps: a short video tutorial (2 min), a warm-up task (5 min), a main task (15 min), and a questionnaire (3 min). After a 15-minute break, participants complete two evaluation tasks (10 min each) for each technique without AI assistance. Techniques during training were fully counterbalanced, while the topics had the same order in both training and evaluation phases of the experiment.

Lastly, participants were given a short questionnaire in which they were asked several long-answer questions to explain their likes, dislikes, and willingness to use each of the two final code engagement techniques.

6.2.1 Data Analysis. To evaluate the performance on the manual coding tasks, we used a simple rubric that focused on the missing lines that participants were asked to fill. Participants only received a full mark for that line if their code for that line was fully correct. Sometimes participants' solution included additional lines of code, in such cases, they would receive full marks only if it did not change the main concept and how the task was using the given data structure for solving the task, otherwise, they did not receive marks.

In this study, we introduced another metric to determine friction created by each technique: participants' accuracy on answering the questions when participants were using each of the LEAD-AND-REVEAL and TRACE-AND-PREDICT techniques which included several questions. We collected the number of questions that was asked, and how many attempts they required to correctly solve it. Participants received 100% for answering correctly on the first attempt, 50% if they answered correctly on the second attempt, and 0% if they required a third attempt (as the correct answer was provided afterward). For each participant, we calculated the average accuracy for each question type, by determining the mean of their performance across all prompted questions. We looked at all three comparisons, therefore, an alpha value of $\alpha = \frac{0.05}{3} = 0.017$ is used to determine statistical significance.

6.3 RQ1 [Performance] How effective are the techniques in supporting learning?

A Friedman test did not show a statistical difference between learners' performance on the post-test evaluation tasks ($\chi^2 = 3.79$, $p = .150$). The TRACE-AND-PREDICT technique achieved the highest mean ($M = 64.3$, $SD = 34.1$), followed by BASELINE ($M = 57.6$, $SD = 39.2$), and LEAD-AND-REVEAL ($M = 55.0$, $SD = 41.1$).

6.4 RQ2 [Friction] What level of friction do the techniques introduce?

A Friedman test revealed a statistically significant difference in completion times among the techniques ($\chi^2(2, N = 30) = 37.80$, $p <$

.001; completion times were not normally distributed as indicated by a Shapiro-Wilk test). The BASELINE technique resulted in the shortest completion time ($M = 470.9$, $SD = 318.2$), followed by LEAD-AND-REVEAL ($M = 857.4$, $SD = 298.6$), while TRACE-AND-PREDICT ($M = 1251.2$, $SD = 469.8$) took almost three times longer than the BASELINE to complete.

In terms of the accuracy of learners' responses to the questions prompted by each technique, for the TRACE-AND-PREDICT technique, learners were asked an average of 8.3 tracing questions (to predict values) and 3.7 short-answer questions to explain the highlighted part of the code. For the LEAD-AND-REVEAL technique, learners were asked an average of 8.7 questions. Learners performed significantly better on the TRACE-AND-PREDICT technique ($M = 76.3\%$, $SD = 14.2\%$) compared to their performance on the LEAD-AND-REVEAL technique ($M = 58.0\%$, $SD = 12.4\%$), as indicated by a paired t-test ($p < .001$).

The Task Load Index asked participants specifically about their cognitive load to *understand the AI-generated code*. For analysis, we averaged the results on the six subscales across the three techniques. A Repeated-measures ANOVA test (since task load index passed normality with a Shapiro-Wilk test) showed that task load across was statistically different across the three techniques ($F = 7.4$, $p = .001$), with the TRACE-AND-PREDICT having the highest task load index ($M = 67.1$, $SD = 13.4$), while the task load showed no statistical difference between the BASELINE ($M = 58.9$, $SD = 14.7$) and LEAD-AND-REVEAL ($M = 59.6$, $SD = 11.9$) techniques.

On particular task load subscales, a Friedman test showed statistically significance on the temporal subscale ($p < 0.001$) and a Wilcoxon signed rank showed the TRACE-AND-PREDICT technique having the highest temporal demand ($M = 75.2$, $SD = 19.2$), while it was similar between BASELINE ($M = 55.2$, $SD = 24.1$) and the LEAD-AND-REVEAL technique ($M = 62.9$, $SD = 18.0$). A similar effect was seen across the physical ($p = 0.019$), frustration ($p = 0.021$), and performance ($p = 0.039$) subscales in which the TRACE-AND-PREDICT had a higher demand, while the LEAD-AND-REVEAL had a similar demand compared to the BASELINE technique, while mental demand, perceived effort, and physical demand was similar across the three techniques.

6.5 RQ3 [Metacognitive Self-Assessment] How do the techniques impact the ability to assess one’s own performance?

A Spearman correlation analysis examined the alignment between participants’ perceived ability (self-reported via Likert-scale responses) and their actual ability (measured through post-test manual coding tasks) across the three techniques. The LEAD-AND-REVEAL intervention showed a moderate, significant correlation, $r = .26, p = .017$, indicating a better alignment of perceived and actual abilities. The BASELINE technique had a weak, non-significant correlation $r = .16, p = .136$, and the TRACE-AND-PREDICT technique similarly showed no significant correlation, $r = .15, p = .171$. These results suggest that the LEAD-AND-REVEAL technique supported participants’ self-awareness during their engagement with the AI-generated code more effectively compared to the other techniques.

6.6 RQ4 [Perceptions] How do participants perceive each technique?

After the second study, participants were asked about their likes, dislikes, and willingness to use each tool. Our qualitative analysis on their responses revealed the following themes:

6.6.1 LEAD-AND-REVEAL (V2). Participants frequently mentioned that the tool facilitated guided problem-solving and provided step-by-step guidance (n=26), which promoted logical thinking (n=17). They appreciated how it enhanced their understanding of the code structure (n=10), offered hints (n=11), and provided detailed feedback on mistakes (n=6). However, participants also pointed out several drawbacks: lack of flexibility in the implementation (n=10), being a time-consuming process (n=9), and difficulty in interpreting certain feedback (n=8).

6.6.2 TRACE-AND-PREDICT (V2). The most common positive feedback centered around real-time variable tracking (n=18). Participants also appreciated the newly added short-answer questions and provided feedback (n=14), which helped them better understand the generated code (n=14). Some compared the experience to using debugging tools (n=10), as it enabled them to visualize code execution (n=7).

On the downside, 15 participants expressed reluctance to use the technique, citing its time-consuming nature and inefficiency for simple or long codes (n=7). Others found the questions too simple or repetitive (n=7). Additionally, some participants (n=6) reported experiencing disruptions to their workflow (n=6).

6.7 Summary of Results

In summary, there was **no significant difference in performance across the three techniques**. However, LEAD-AND-REVEAL improved participants’ alignment between perceived and actual ability without increasing perceived task load, despite taking 1.82x longer than BASELINE. TRACE-AND-PREDICT, while taking 2.66x longer, introduced higher friction. Participants preferred LEAD-AND-REVEAL for its support in problem-solving and computational thinking, addressing key challenges in AI-assisted learning by actively engaging users in the process. While this warrants future studies to explore

the long-term impact and scalability of these techniques, **our current results point to the effectiveness of LEAD-AND-REVEAL in fostering self-awareness and enhancing problem-solving skills during AI-assisted programming.**

7 DISCUSSION AND FUTURE WORK

Our results indicate that the LEAD-AND-REVEAL technique achieved the highest performance in Study 1 (although not statistically significant) and demonstrated the best alignment between perceived and actual learning in Study 2, without increasing cognitive load compared to BASELINE in both studies. To facilitate further research and usage, we have made the LEAD-AND-REVEAL technique and an accompanying task builder publicly available at <https://lead-and-reveal.vercel.app>, enabling others to create new tasks using this technique.

In the rest of this section, we discuss the limitations of our work, synthesize design implications, and propose directions for future work.

7.1 Limitations

Our results are limited by the small sample size in Study 1 and limited exposure of participants to each technique in Study 2. These factors may explain the lack of observed statistical effects. Additionally, both studies focused on algorithm-heavy tasks, limiting task diversity. While tasks involving unfamiliar syntax or APIs were considered (as in Yan et al.’s study [119]), we focused on tasks that posed a greater challenge for engagement with AI-generated code beyond simple memorization. Instead our tasks require engagement from multiple angles including decomposition, pattern recognition, data representation, abstraction, and algorithm design. Although this may limit the generalizability of our results, we believe our study addresses the most critical aspects of the issue.

In our studies, we assumed AI-generated solutions were correct, although LLMs can generate incorrect code. For techniques like TRACE-AND-PREDICT and EXPLAIN-BEFORE-USAGE, where the code is displayed before user interaction, this assumption is less problematic since users can read and test the code for verification. However, for techniques that involve user engagement before revealing the final solution, this could reduce perceived value of the interaction if the final code is incorrect. Although this can be mitigated by improved models and techniques (e.g., Chain-of-Thought [114] that power recent models like GPT-o1), future techniques could verify AI-generated code by running test cases, engaging users only if the tests pass. Additionally, the techniques did not allow users to ask follow-up questions from the AI. Future research could explore follow-up interactions as another metric for user engagement with AI-generated code. Participants were also assigned to specific techniques in our experiments, we envision that in real-world settings, users would have the option to choose their preferred type of friction. Future work could explore providing this agency and examine user preferences over time. These simplifications in our study designs ensured consistency and allowed us to focus on evaluating the effectiveness of the types of friction introduced with each technique.

Lead-and-reveal.
not S.S.

API initialization
pipeline, etc.

Chain-of-thought -
- Task uses to
verify AI
code

* Follow-up Qs.
* choose type
of friction

7.2 Friction-Induced AI

To ensure long-term productivity gains, researchers in human-AI interaction have proposed various concepts, such as promoting ‘metacognitive’ reflection [106], antagonistic or sycophantic AI [14], or making AI act as coach [48], or AI as a provocateur [97]. A common theme among these concepts, is a call for action that AI should augment and not automate. In alignment with these approaches, we introduce *Friction-Induced AI*, wherein the AI does not allow users to use its generated solution immediately. Instead, it engages the user in the process of generation, while challenging them (similar to LEAD-AND-REVEAL) and providing opportunities for reflection. We believe this concept generalizes beyond just AI-assisted programming but human-AI interaction more generally. The term “friction” here does not refer to creating frustration or unnecessary difficulty. Rather, it refers to temporarily slowing the user’s interaction, ensuring they engage critically, rather than passively using AI’s output. However, our results—specifically in code generation—show that for effective outcomes, friction must be properly designed. Below we highlight key dimensions in effectively designing *Friction-Induced AI*, particularly in AI-assisted programming.

7.2.1 Effective Friction Type. An interesting finding from the second study (Section 6) was that participants performed significantly less accurately on the leading short-answer questions in LEAD-AND-REVEAL (58%) compared to TRACE-AND-PREDICT (76%), indicating that users were more challenged by the questions in LEAD-AND-REVEAL. Notably, despite the increased challenge, the use of LEAD-AND-REVEAL did not result in a higher cognitive load compared to BASELINE, whereas TRACE-AND-PREDICT did. This suggests that effective and beneficial friction should be designed to engage users in meaningful activities, similar to LEAD-AND-REVEAL, which engaged users in the problem-solving aspect of the task. The technique also allowed reflection in a short, interactive conversation with the AI. Similarly, SOLVE-CODE-PUZZLE provided a scaffolded approach to constructing code from scrambled code blocks, which previous research has shown to be both fun and engaging without imposing high cognitive demands [19, 87, 115].

Additionally, the LEAD-AND-REVEAL technique inherently has gamification in its engagement and learning process: the next line of code is revealed only after the user successfully explains the action required in that line. This could be a rewarding experience and increase motivation while engaging the user in a process that mirrors real-world problem-solving. However, not all tasks are solved linearly, from top to bottom. Often, code is constructed from a combination of patterns that are distributed throughout the code. Future tools should therefore focus on decomposing code hierarchically and guiding users in the hierarchical decision making process.

In contrast, effective friction should avoid engaging the user in simple and repetitive tasks like typing over generated code (GUIDED-WRITE-OVER), or merely translating pseudocode to code (INTERACTIVE-PSEUDO-CODE). It should also avoid tasks where the engagement does not create new knowledge like repeatedly tracing values of different variables (TRACE-AND-PREDICT), or confusing the user by mixing correct and incorrect code (VERIFY-AND-REVIEW).

7.2.2 Beyond Code Generation. AI-assisted programming extends beyond mere code generation or completion. It includes various scenarios where introducing appropriate friction can enhance cognitive engagement and help prevent skill degradation. In these contexts, AI operates autonomously, solving a user’s problem like an agent without additional user involvement. Here we discuss two particular scenarios: debugging and AI-assisted decision making.

First, when AI is used for debugging and fixing a user’s code—particularly for novice or end-user programmers [66]—it often generates the fixed solution. However, debugging is a fundamental computational thinking skill [117], and must be practiced to ensure programming proficiency. Researchers have already explored introducing similar forms of friction to support help-seeking behaviors. CodeAid [64] highlights incorrect lines of code with suggested fixes that users must apply themselves. Similarly, CodeTailor [51] generates a scrambled version of the fixed code, requiring users to rearrange it, similar to SOLVE-CODE-PUZZLE. Another form of meaningful friction could involve engaging the user in the step-by-step debugging process alongside the AI, similar to LEAD-AND-REVEAL. This approach could enhance debugging skills while offering a greater sense of accomplishment.

Second, when AI is being used to make decisions for programming tasks—especially in scenarios involving complex trade-offs, such as machine learning, data analysis [38, 39], and software design [116]—it is crucial to encourage programmers to think critically about these decisions. Reflecting on these decisions promotes their critical thinking and problem-solving skills, and their ability to verify AI-generated code and decisions. Researchers are already developing tools to assist novices in making better software design and architectural decisions with generative AI [21]. However, when AI autonomously solves tasks for the user, there is a risk that this could negatively impact their decision-making skills. A more engaged, user-involved AI assistance might result in better decisions and improved mental models, an area that future research should explore.

7.2.3 Beyond Educational Contexts. Lastly, an important design implication emerges from the synergy between adding friction and enhanced verification in AI-assisted programming. Although we studied friction in the context of programming with novices learning about complex algorithmic tasks, but friction can potentially be suitable in productivity scenarios as well. Friction-induced AI can potentially prevent long-term productivity losses, as well as enhance short-term verification of AI outputs, leading to immediate productivity gains. Introducing friction, like cognitive forcing functions, has been shown to support AI-assisted decision-making and reduce over-reliance on AI [12]. Similarly, adding intervention points and employing progressive disclosure has been another type of friction that has been demonstrated to increase programmers’ sense of control and ability to verify outputs during AI-assisted data analysis [63].

This dual benefit underscores how adding meaningful and effective friction could make it suitable for productivity scenarios and not just for educational contexts. It not only improves short-term verification and productivity but also helps prevent long-term productivity loss by augmenting human cognition and encouraging higher-order cognitive engagement. As a result, techniques like

Friction:
Design

Baseline?

sycophantic AI?

LEAD-AND-REVEAL could be designed to incorporate intervention points that allows users to progressively verify and correct the AI’s output, while also promoting deeper learning as code is generated.

8 CONCLUSION

Over-reliance on AI poses considerable risk to the development of computational thinking skills among novice programmers. However, as generative AI continues to change the landscape of programming, it is essential to harness its benefits by understanding how to effectively integrate it into educational settings. In this paper, we introduced the concept of *friction-induced AI*, which requires users to cognitively engage with AI-generated content (specifically code), to enhance short-term productivity gains while preventing long-term productivity loss due to over-reliance on AI.

We systematically explored the design space of cognitive engagement techniques, developing seven distinct interfaces that introduce varying levels and types of engagement. Through an iterative design process and two empirical evaluations, we identified that the **LEAD-AND-REVEAL technique balances imposed friction**, learning gains, and improved self-assessment accuracy. Specifically, LEAD-AND-REVEAL improved the alignment between learns’ perceived and actual coding abilities without increasing cognitive load, demonstrating its potential to improve learning and prevent over-reliance. Beneficial friction can be viewed as a supportive scaffold that empowers students rather than acting as an obstacle. By requiring users to engage actively, these techniques promote metacognitive reflection and improve problem-solving skills. Future research should explore more diverse programming tasks and user-driven engagement strategies to further validate these findings. Ultimately, designing AI tools that support deeper cognitive reflection remains crucial to maintaining both short-term productivity and long-term skill development in programming.

ACKNOWLEDGMENTS

We would like to thank the computing education researchers for their valuable feedback on our techniques. We also sincerely appreciate the course instructors for their input on the design of the programming tasks, and we are grateful to the students who participated in our experiments.

REFERENCES

- [1] Amjad Altadmri and Neil C.C. Brown. 2015. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) (*SIGCSE ’15*). ACM, New York, NY, USA, 522–527. <https://doi.org/10.1145/2676723.2677258>
- [2] Hamsa Bastani, Osbert Bastani, Alp Sungu, Haosen Ge, Özge Kabakcı, and Rei Mariman. 2024. *Generative AI Can Harm Learning*. Research Paper. The Wharton School. <https://doi.org/10.2139/ssrn.4895486> Available at SSRN: <https://ssrn.com/abstract=4895486> or <http://dx.doi.org/10.2139/ssrn.4895486>.
- [3] Brett A. Becker. 2016. An Effective Approach to Enhancing Compiler Error Messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) (*SIGCSE ’16*). ACM, New York, NY, USA, 126–131. <https://doi.org/10.1145/2839509.2844584>
- [4] Brett A. Becker. 2016. A New Metric to Quantify Repeated Compiler Errors for Novice Programmers. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (Arequipa, Peru) (*ITICSE ’16*). ACM, New York, NY, USA, 296–301. <https://doi.org/10.1145/2899415.2899463>
- [5] Brett A. Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming Is Hard - Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto ON, Canada) (*SIGCSE 2023*). Association for Computing Machinery, New York, NY, USA, 500–506. <https://doi.org/10.1145/3545945.3569759>
- [6] Timothy C Bell, Ian H Witten, and Mike Fellows. 1998. *Computer Science Unplugged: Off-line Activities and Games for All Ages* (1st ed.). Computer Science Unplugged.
- [7] Benjamin S Bloom, Max D Engelhart, Edward J Furst, Walker H Hill, David R Krathwohl, et al. 1956. *Taxonomy of Educational Objectives: The Classification of Educational Goals. Handbook 1: Cognitive Domain*. Longman New York.
- [8] Jeffrey Bonar and Elliot Soloway. 2013. Preprogramming knowledge: A major source of misconceptions in novice programmers. In *Studying the novice programmer*. Psychology Press, 325–353.
- [9] Christian P. Brackmann, Marcos Román-González, Gregorio Robles, Jesús Moreno-León, Ana Casali, and Dante Barone. 2017. Development of Computational Thinking Skills through Unplugged Activities in Primary School. In *Proceedings of the 12th Workshop on Primary and Secondary Computing Education* (Nijmegen, Netherlands) (*WiPSCE ’17*). Association for Computing Machinery, New York, NY, USA, 65–72. <https://doi.org/10.1145/3137065.3137069>
- [10] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Boston, MA, USA) (*CHI ’09*). Association for Computing Machinery, New York, NY, USA, 1589–1598. <https://doi.org/10.1145/1518701.1518944>
- [11] Neil C.C. Brown and Amjad Altadmri. 2014. Investigating Novice Programming Mistakes: Educator Beliefs vs. Student Data. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (Glasgow, Scotland, United Kingdom) (*ICER ’14*). ACM, New York, NY, USA, 43–50. <https://doi.org/10.1145/2632320.2632343>
- [12] Zana Bućinca, Maja Barbara Malaya, and Krzysztof Z. Gajos. 2021. To Trust or to Think: Cognitive Forcing Functions Can Reduce Overreliance on AI in AI-assisted Decision-making. *Proc. ACM Hum.-Comput. Interact.* 5, CSCW1, Article 188 (apr 2021), 21 pages. <https://doi.org/10.1145/3449287>
- [13] Zana Bućinca, Maja Barbara Malaya, and Krzysztof Z. Gajos. 2021. To Trust or to Think: Cognitive Forcing Functions Can Reduce Overreliance on AI in AI-assisted Decision-making. *Proc. ACM Hum.-Comput. Interact.* 5, CSCW1 (apr 2021).
- [14] Alice Cai, Ian Arawjo, and Elena L. Glassman. 2024. Antagonistic AI. arXiv:2402.07350 <https://arxiv.org/abs/2402.07350>
- [15] Preetha Chatterjee, Minji Kong, and Lori Pollock. 2020. Finding help with programming errors: An exploratory study of novice software engineers’ focus in stack overflow posts. *Journal of Systems and Software* 159 (2020), 110454. <https://doi.org/10.1016/j.jss.2019.110454>
- [16] Michelene T. H. Chi and Ruth Wylie. 2014. The ICAP Framework: Linking Cognitive Engagement to Active Learning Outcomes. *Educational Psychologist* 49, 4 (2014), 219–243. <https://doi.org/10.1080/00461520.2014.965823>
- [17] 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* (Portland, OR, USA) (*SIGCSE 2024*). Association for Computing Machinery, New York, NY, USA, 296–302. <https://doi.org/10.1145/3626252.3630909>
- [18] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. 2014. Enhancing Syntax Error Messages Appears Ineffectual. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education* (Uppsala, Sweden) (*ITICSE ’14*). ACM, New York, NY, USA, 273–278. <https://doi.org/10.1145/2591708.2591748>
- [19] Paul Denny, Andrew Luxton-Reilly, and Beth Simon. 2008. Evaluating a new exam question: Parsons problems. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) (*ICER ’08*). Association for Computing Machinery, New York, NY, USA, 113–124. <https://doi.org/10.1145/1404520.1404532>
- [20] Paul Denny, David H. Smith, Max Fowler, James Prather, Brett A. Becker, and Juho Leinonen. 2024. Explaining Code with a Purpose: An Integrated Approach for Developing Code Comprehension and Prompting Skills. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) (*ITICSE 2024*). Association for Computing Machinery, New York, NY, USA, 283–289. <https://doi.org/10.1145/3649217.3653587>
- [21] J Andrés Díaz-Pace, Antonela Tommasel, and Rafael Capilla. 2024. Helping Novice Architects to Make Quality Design Decisions Using an LLM-Based Assistant. In *Software Architecture*. Springer Nature Switzerland, Cham, 324–332.
- [22] Benedict Du Boulay. 2013. Some difficulties of learning to program. In *Studying the novice programmer*. Psychology Press, 283–299.
- [23] Alireza Ebrahimi. 1994. Novice programmer errors: Language constructs and plan composition. *International Journal of Human Computer Studies* 41, 4 (1994), 457–480.
- [24] Barbara J. Ericson, James D. Foley, and Jochen Rick. 2018. Evaluating the Efficiency and Effectiveness of Adaptive Parsons Problems. In *Proceedings of the*

- 2018 ACM Conference on International Computing Education Research (Espoo, Finland) (ICER '18). Association for Computing Machinery, New York, NY, USA, 60–68. <https://doi.org/10.1145/3230977.3231000>
- [25] 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* (Koli, Finland) (Koli Calling '17). Association for Computing Machinery, New York, NY, USA, 20–29. <https://doi.org/10.1145/3141880.3141895>
- [26] K Anders Ericsson, Ralf T Krampe, and Clemens Tesch-Römer. 1993. The role of deliberate practice in the acquisition of expert performance. *Psychological review* 100, 3 (1993), 363. <https://doi.org/10.1037/0033-295X.100.3.363>
- [27] Ethan Fast, Daniel Steffee, Lucy Wang, Joel R. Brandt, and Michael S. Bernstein. 2014. Emergent, Crowd-scale Programming Practice in the IDE. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, Ontario, Canada) (CHI '14). ACM, New York, NY, USA, 2491–2500. <https://doi.org/10.1145/2556288.2556998>
- [28] 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* (Melbourne, VIC, Australia) (ACE '23). Association for Computing Machinery, New York, NY, USA, 97–104. <https://doi.org/10.1145/3576123.3576134>
- [29] T. Flowers, J. Jackson, and C. Carver. 2004. Empowering students and building confidence in novice programmers through Gauntlet. In *34th Annual Frontiers in Education, 2004. FIE 2004.(FIE)*, Vol. 00. T3H/10–T3H/13 Vol. 1. <https://doi.org/10.1109/FIE.2004.1408551>
- [30] Krzysztof Z. Gajos and Lena Mamykina. 2022. Do People Engage Cognitively with AI? Impact of AI Assistance on Incidental Learning. In *Proceedings of the 27th International Conference on Intelligent User Interfaces* (Helsinki, Finland) (IUI '22). Association for Computing Machinery, New York, NY, USA, 794–806. <https://doi.org/10.1145/3490099.3511138>
- [31] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) (ICSE 2014). ACM, New York, NY, USA, 653–663. <https://doi.org/10.1145/2568225.2568250>
- [32] Adam M. Gaweda, Collin F. Lynch, Nathan Seamon, Gabriel Silva de Oliveira, and Alay Deliwa. 2020. Typing Exercises as Interactive Worked Examples for Deliberate Practice in CS Courses. In *Proceedings of the Twenty-Second Australasian Computing Education Conference* (Melbourne, VIC, Australia) (ACE'20). Association for Computing Machinery, New York, NY, USA, 105–113. <https://doi.org/10.1145/3373165.3373177>
- [33] Aashish Ghimire and John Edwards. 2024. Coding with AI: How Are Tools Like ChatGPT Being Used by Students in Foundational Programming Courses. In *Artificial Intelligence in Education*, Andrew M. Olney, Irene-Angelica Chounta, Zitao Liu, Olga C. Santos, and Ig Ibert Bittencourt (Eds.). Springer Nature Switzerland, Cham, 259–267.
- [34] GitHub. 2024. GitHub Copilot: Your AI Pair Programmer. <https://github.com/features/copilot> Accessed: 2024-09-01.
- [35] Kate Goddard, Abdul Roudsari, and Jeremy C Wyatt. 2012. Automation bias: a systematic review of frequency, effect mediators, and mitigators. *Journal of the American Medical Informatics Association* 19, 1 (2012), 121–127.
- [36] Jean M. Griffin. 2019. Designing Intentional Bugs for Learning. In *Proceedings of the 2019 Conference on United Kingdom & Ireland Computing Education Research* (Canterbury, United Kingdom) (UKICER '19). Association for Computing Machinery, New York, NY, USA, Article 5, 7 pages. <https://doi.org/10.1145/3351287.3351289>
- [37] Cornelia S Groe and Alexander Renkl. 2007. Finding and fixing errors in worked examples: Can this foster learning outcomes? *Learning and Instruction* 17, 6 (2007), 612–634. <https://doi.org/10.1016/j.learninstruc.2007.09.008>
- [38] Ken Gu, Madeleine Grunde-McLaughlin, Andrew McNutt, Jeffrey Heer, and Tim Althoff. 2024. How Do Data Analysts Respond to AI Assistance? A Wizard-of-Oz Study. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) (CHI '24). Association for Computing Machinery, New York, NY, USA, Article 1015, 22 pages. <https://doi.org/10.1145/3613904.3641891>
- [39] Ken Gu, Ruoxi Shang, Tim Althoff, Chenglong Wang, and Steven M. Drucker. 2024. How Do Analysts Understand and Verify AI-Assisted Data Analyses?. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) (CHI '24). Association for Computing Machinery, New York, NY, USA, Article 748, 22 pages. <https://doi.org/10.1145/3613904.3642497>
- [40] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for Cs Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) (SIGCSE '13). ACM, New York, NY, USA, 579–584. <https://doi.org/10.1145/2445196.2445368>
- [41] Philip J. Guo. 2013. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) (SIGCSE '13). Association for Computing Machinery, New York, NY, USA, 579–584. <https://doi.org/10.1145/2445196.2445368>
- [42] Philip J. Guo. 2023. Six Opportunities for Scientists and Engineers to Learn Programming Using AI Tools Such as ChatGPT. *Computing in Science & Engineering* 25, 3 (2023), 73–78. <https://doi.org/10.1109/MCSE.2023.3308476>
- [43] Sally Hamouda, Stephen H Edwards, Hicham G Elmongui, Jeremy V Ernst, and Clifford A Shaffer. 2017. A basic recursion concept inventory. *Computer Science Education* 27, 2 (2017), 121–148. <https://doi.org/10.1080/08993408.2017.1414728> arXiv:<https://doi.org/10.1080/08993408.2017.1414728>
- [44] Kyle J. Harms, Dennis Cosgrove, Shannon Gray, and Caitlin Kelleher. 2013. Automatically Generating Tutorials to Enable Middle School Children to Learn Programming Independently. In *Proceedings of the 12th International Conference on Interaction Design and Children* (New York, New York, USA) (IDC '13). ACM, New York, NY, USA, 11–19. <https://doi.org/10.1145/2485760.2485764>
- [45] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. 2010. What Would Other Programmers Do: Suggesting Solutions to Error Messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) (CHI '10). ACM, New York, NY, USA, 1019–1028. <https://doi.org/10.1145/1753326.1753478>
- [46] A. Head, C. Appachu, M. A. Hearst, and B. Hartmann. 2015. Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 3–12. <https://doi.org/10.1109/VLHCC.2015.7356972>
- [47] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2018. Augmenting Code with In Situ Visualizations to Aid Program Understanding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) (CHI '18). ACM, New York, NY, USA, Article 532, 12 pages. <https://doi.org/10.1145/3173574.3174106>
- [48] Jake Hofman, Daniel G. Goldstein, and David Rothschild. 2023. A Sports Analogy for Understanding Different Ways to Use AI. *Harvard Business Review* (December 2023). <https://www.microsoft.com/en-us/research/publication/a-sports-analogy-for-understanding-different-ways-to-use-ai/>
- [49] Irene Hou, Owen Man, Sophia Mettillle, 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* (Sydney, NSW, Australia) (ACE '24). Association for Computing Machinery, New York, NY, USA, 29–38. <https://doi.org/10.1145/3636243.3636247>
- [50] Irene Hou, Sophia Mettillle, 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* (Sydney, NSW, Australia) (ACE '24). Association for Computing Machinery, New York, NY, USA, 39–48. <https://doi.org/10.1145/3636243.3636248>
- [51] Xinying Hou, Zihan Wu, Xu Wang, and Barbara J. Ericson. 2024. CodeTailor: LLM-Powered Personalized Parsons Puzzles for Engaging Support While Learning Programming. In *Proceedings of the Eleventh ACM Conference on Learning @ Scale* (Atlanta, GA, USA) (L@S '24). Association for Computing Machinery, New York, NY, USA, 51–62. <https://doi.org/10.1145/3657604.3662032>
- [52] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and correcting Java programming errors for introductory computer science students. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (Reno, Nevada, USA) (SIGCSE '03). Association for Computing Machinery, New York, NY, USA, 153–156. <https://doi.org/10.1145/611892.611956>
- [53] M. Ichinco, W. Hnin, and C. Kelleher. 2016. Suggesting examples to novice programmers in an open-ended context with the example guru. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 230–231. <https://doi.org/10.1109/VLHCC.2016.7739691>
- [54] Michelle Ichinco, Wint Yee Hnin, and Caitlin L. Kelleher. 2017. Suggesting API Usage to Novice Programmers with the Example Guru. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) (CHI '17). ACM, New York, NY, USA, 1105–1117. <https://doi.org/10.1145/3025453.3025827>
- [55] Michelle Ichinco and Caitlin Kelleher. 2015. Exploring novice programmer example use. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 63–71. <https://doi.org/10.1109/VLHCC.2015.7357199>
- [56] Michelle Ichinco and Caitlin Kelleher. 2018. Semi-automatic Suggestion Generation for Young Novice Programmers in an Open-ended Context. In *Proceedings of the 17th ACM Conference on Interaction Design and Children* (Trondheim, Norway) (IDC '18). ACM, New York, NY, USA, 405–412. <https://doi.org/10.1145/3202185.3202762>
- [57] J. Jackson, M. Cobb, and C. Carver. 2005. Identifying Top Java Errors for Novice Programmers. In *Proceedings Frontiers in Education 35th Annual Conference*. T4C–T4C. <https://doi.org/10.1109/FIE.2005.1611967>
- [58] Lisa C. Kaczmarczyk, Elizabeth R. Petrick, J. Philip East, and Geoffrey L. Herman. 2010. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (Milwaukee,

- Wisconsin, USA) (*SIGCSE '10*). Association for Computing Machinery, New York, NY, USA, 107–111. <https://doi.org/10.1145/1734263.1734299>
- [59] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) (*UIST '17*). ACM, New York, NY, USA, 737–745. <https://doi.org/10.1145/3126594.3126632>
- [60] Enkelejda Kasneci, Kathrin Sessler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günemann, Eyke Hüllermeier, Stephan Krusche, Gitta Kutyniok, Tilman Michaeli, Claudia Nerdel, Jürgen Pfeffer, Oleksandra Poquet, Michael Sailer, Albrecht Schmidt, Tina Seidel, Matthias Stadler, Jochen Weller, Jochen Kuhn, and Gjergji Kasneci. 2023. ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and Individual Differences* 103 (2023), 102274. <https://doi.org/10.1016/j.lindif.2023.102274>
- [61] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) (*CHI '23*). Association for Computing Machinery, New York, NY, USA, 23 pages. <https://doi.org/10.1145/3544548.3580919>
- [62] Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara Jane Ericson, David Weintrop, and Tovi Grossman. 2024. How Novices Use LLM-based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research* (Koli, Finland) (*Koli Calling '23*). Association for Computing Machinery, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3631802.3631806>
- [63] Majeed Kazemitabaar, Jack Williams, Ian Drosos, Tovi Grossman, Austin Henley, Carina Negreanu, and Advait Sarkar. 2024. Improving Steering and Verification in AI-Assisted Data Analysis with Interactive Task Decomposition. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology* (Pittsburgh, PA, USA) (*UIST '24*). Association for Computing Machinery, New York, NY, USA, 19 pages. <https://doi.org/10.1145/3654777.3676345>
- [64] Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, Austin Zachary Henley, Paul Denny, Michelle Craig, and Tovi Grossman. 2024. CodeAid: Evaluating a Classroom Deployment of an LLM-based Programming Assistant that Balances Student and Educator Needs. In *Proceedings of the CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) (*CHI '24*). Association for Computing Machinery, New York, NY, USA, Article 650, 20 pages. <https://doi.org/10.1145/3613904.3642773>
- [65] Cazembe Kennedy and Eileen T. Kraemer. 2019. Qualitative Observations of Student Reasoning: Coding in the Wild. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland UK) (*ITiCSE '19*). Association for Computing Machinery, New York, NY, USA, 224–230. <https://doi.org/10.1145/3304221.3319751>
- [66] Amy J. Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six Learning Barriers in End-User Programming Systems. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing*. 199–206. <https://doi.org/10.1109/VLHCC.2004.47>
- [67] Yifat Ben-David Kolikant. 2005. Students' alternative standards for correctness. In *Proceedings of the First International Workshop on Computing Education Research* (Seattle, WA, USA) (*ICER '05*). Association for Computing Machinery, New York, NY, USA, 37–43. <https://doi.org/10.1145/1089786.1089790>
- [68] Thomas Kosch, Jakob Karolus, Johannes Zagermann, Harald Reiterer, Albrecht Schmidt, and Paweł W. Woźniak. 2023. A Survey on Measuring Cognitive Workload in Human-Computer Interaction. *ACM Comput. Surv.* 55, 13s, Article 283 (July 2023), 39 pages. <https://doi.org/10.1145/3582272>
- [69] Amruth N. Kumar. 2013. A study of the influence of code-tracing problems on code-writing skills. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education* (Canterbury, England, UK) (*ITiCSE '13*). Association for Computing Machinery, New York, NY, USA, 183–188. <https://doi.org/10.1145/2462476.2462507>
- [70] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A study of the difficulties of novice programmers. *SIGCSE Bull.* 37, 3 (jun 2005), 14–18. <https://doi.org/10.1145/1151954.1067453>
- [71] 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* (Chicago, IL, USA) (*ICER '23*). Association for Computing Machinery, New York, NY, USA, 106–121. <https://doi.org/10.1145/3568813.3600138>
- [72] Michael J. Lee, Faezeh Bahmani, Irwin Kwan, Jilian LaFerte, Polina Charters, Amber Horvath, Fanny Luor, Jill Cao, Catherine Law, Michael Beswetherick, Sheridan Long, Margaret Burnett, and Amy J. Ko. 2014. Principles of a debugging-first puzzle game for computing education. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 57–64. <https://doi.org/10.1109/VLHCC.2014.6883023>
- [73] Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing Misconceptions About Code with Always-on Programming Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, Ontario, Canada) (*CHI '14*). ACM, New York, NY, USA, 2481–2490. <https://doi.org/10.1145/2556288.2557409>
- [74] Mark Liffiton, Brad E Sheese, Jaromir Savelka, and Paul Denny. 2024. Code-Help: Using Large Language Models with Guardrails for Scalable Support in Programming Classes. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research* (Koli, Finland) (*Koli Calling '23*). Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. <https://doi.org/10.1145/3631802.3631830>
- [75] Marcia C Linn and Michael J Clancy. 1992. The case for case studies of programming problems. *Commun. ACM* 35, 3 (1992), 121–132.
- [76] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. 2006. Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) (*ITiCSE '06*). Association for Computing Machinery, New York, NY, USA, 118–122. <https://doi.org/10.1145/1140124.1140157>
- [77] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) (*ICER '08*). Association for Computing Machinery, New York, NY, USA, 101–112. <https://doi.org/10.1145/1404520.1404531>
- [78] Richard E. Mayer. 1981. The Psychology of How Novices Learn Computer Programming. *ACM Comput. Surv.* 13, 1 (mar 1981), 121–141. <https://doi.org/10.1145/356835.356841>
- [79] Renee 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. *Computer Science Education* 18, 2 (2008), 67–92. <https://doi.org/10.1080/08993400802114581> arXiv:<https://doi.org/10.1080/08993400802114581>
- [80] Daphne Miedema, Efthimia Aivaloglou, and George Fletcher. 2022. Identifying SQL misconceptions of novices: findings from a think-aloud study. *ACM Inroads* 13, 1 (feb 2022), 52–65. <https://doi.org/10.1145/3514214>
- [81] Briana B. Morrison, Lauren E. Margulieux, and Mark Guzdial. 2015. Subgoals, Context, and Worked Examples in Learning Computing Problem Solving. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (Omaha, Nebraska, USA) (*ICER '15*). Association for Computing Machinery, New York, NY, USA, 21–29. <https://doi.org/10.1145/2787622.2787733>
- [82] Dhawal Mujumdar, Manuel Kallenbach, Brandon Liu, and Björn Hartmann. 2011. Crowdsourcing Suggestions to Programming Problems for Dynamic Web Development Languages. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems* (Vancouver, BC, Canada) (*CHI EA '11*). ACM, New York, NY, USA, 1525–1530. <https://doi.org/10.1145/1979742.1979802>
- [83] Laurie Murphy, Sue Fitzgerald, Raymond Lister, and Renée McCauley. 2012. Ability to 'explain in plain english' linked to proficiency in computer-based programming. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research* (Auckland, New Zealand) (*ICER '12*). Association for Computing Machinery, New York, NY, USA, 111–118. <https://doi.org/10.1145/2361276.2361299>
- [84] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. 2008. Compiler Error Messages: What Can Help Novices?. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (Portland, OR, USA) (*SIGCSE '08*). ACM, New York, NY, USA, 168–172. <https://doi.org/10.1145/1352135.1352192>
- [85] Stephen Oney and Joel Brandt. 2012. Codelets: Linking Interactive Documentation and Example Code in the Editor. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Austin, Texas, USA) (*CHI '12*). ACM, New York, NY, USA, 2697–2706. <https://doi.org/10.1145/2207676.2208664>
- [86] Joon Sung Park, Rick Barber, Alex Kirlik, and Karrie Karahalios. 2019. A Slow Algorithm Improves Users' Assessments of the Algorithm's Accuracy. *Proc. ACM Hum.-Comput. Interact.* 3, CSCW, Article 102 (nov 2019), 15 pages. <https://doi.org/10.1145/3359204>
- [87] Dale Parsons and Patricia Haden. 2006. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52* (Hobart, Australia) (*ACE '06*). Australian Computer Society, Inc., AUS, 157–163.
- [88] Barry Peddycord III, Andrew Hicks, and Tiffany Barnes. 2014. Generating hints for programming problems using intermediate output. In *Educational Data Mining 2014*. Citeseer.
- [89] Raymond S. Pettit, John Homer, and Roger Gee. 2017. Do Enhanced Compiler Error Messages Help Students?: Results Inconclusive.. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) (*SIGCSE '17*). ACM, New York, NY, USA, 465–470. <https://doi.org/10.1145/3017680.3017768>

- [90] James Prather, Paul Denny, Juho Leinonen, Brett A. Becker, Ibrahim Albluwi, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, Andrew Luxton-Reilly, Stephen MacNeil, Andrew Petersen, Raymond Pettit, Brent N. Reeves, and Jaromir Savelka. 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* (Turku, Finland) (*ITiCSE-WGR '23*). Association for Computing Machinery, New York, NY, USA, 108–159. <https://doi.org/10.1145/3623762.3633499>
- [91] James Prather, Raymond Pettit, Kayla Holcomb McMurtry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. 2017. On Novices' Interaction with Compiler Error Messages: A Human Factors Approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) (*ICER '17*). ACM, New York, NY, USA, 74–82. <https://doi.org/10.1145/3105726.3106169>
- [92] 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* (Melbourne, VIC, Australia) (*ICER '24*). Association for Computing Machinery, New York, NY, USA, 469–486. <https://doi.org/10.1145/3632620.3671116>
- [93] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18, 1, Article 1 (oct 2017), 24 pages. <https://doi.org/10.1145/3077618>
- [94] Noa Ragonis and Mordechai Ben-Ari. 2005. A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education* 15, 3 (2005), 203–221. <https://doi.org/10.1080/08993400500224310> arXiv:<https://doi.org/10.1080/08993400500224310>
- [95] Evan F Risko and Sam J Gilbert. 2016. Cognitive offloading. *Trends in cognitive sciences* 20, 9 (2016), 676–688.
- [96] William Robinson. 2016. From Scratch to Patch: Easing the Blocks-Text Transition. In *Proceedings of the 11th Workshop in Primary and Secondary Computing Education* (Münster, Germany) (*WiPSCE '16*). Association for Computing Machinery, New York, NY, USA, 96–99. <https://doi.org/10.1145/2978249.2978265>
- [97] Advait Sarkar. 2024. AI Should Challenge, Not Obey. *Commun. ACM* (Sept. 2024), 5 pages. <https://doi.org/10.1145/3649404> Online First.
- [98] 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 - Volume 1* (Chicago, IL, USA) (*ICER '23*). Association for Computing Machinery, New York, NY, USA, 78–92. <https://doi.org/10.1145/3568813.3600142>
- [99] Andreas Scholl and Natalie Kiesler. 2024. How Novice Programmers Use and Experience ChatGPT when Solving Programming Exercises in an Introductory Course. *arXiv preprint arXiv:2407.20792* (2024).
- [100] Andreas Scholl, Daniel Schiffner, and Natalie Kiesler. 2024. Analyzing Chat Protocols of Novice Programmers Solving Introductory Programming Tasks with ChatGPT. *arXiv preprint arXiv:2405.19132* (2024).
- [101] James Skripchuk, Neil Bennett, Jeffrey Zhang, Eric Li, and Thomas Price. 2023. Analysis of Novices' Web-Based Help-Seeking Behavior While Programming. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto ON, Canada) (*SIGCSE 2023*). Association for Computing Machinery, New York, NY, USA, 945–951. <https://doi.org/10.1145/3545945.3569852>
- [102] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Trans. Comput. Educ.* 13, 4, Article 15 (nov 2013), 64 pages. <https://doi.org/10.1145/2490822>
- [103] James C. Spohrer and Elliot Soloway. 1986. Novice mistakes: are the folk wisdoms correct? *Commun. ACM* 29, 7 (July 1986), 624–632. <https://doi.org/10.1145/6138.6145>
- [104] R. Suzuki, G. Soares, A. Head, E. Glassman, R. Reis, M. Mongiovi, L. D'Antoni, and B. Hartmann. 2017. TraceDiff: Debugging unexpected code behavior using trace divergences. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 107–115. <https://doi.org/10.1109/VLHCC.2017.8103457>
- [105] Lasang Jimba Tamang, Zeyad Alshaikh, Nisrine Ait Khayi, Priti Oli, and Vasile Rus. 2021. A Comparative Study of Free Self-Explanations and Socratic Tutoring Explanations for Source Code Comprehension. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) (*SIGCSE '21*). Association for Computing Machinery, New York, NY, USA, 219–225. <https://doi.org/10.1145/3408877.3432423>
- [106] Lev Tankelevitch, Viktor Kewenig, Auste Simkute, Ava Elizabeth Scott, Advait Sarkar, Abigail Sellen, and Sean Rintel. 2024. The Metacognitive Demands and Opportunities of Generative AI. In *Proceedings of the CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) (*CHI '24*). Association for Computing Machinery, New York, NY, USA, Article 680, 24 pages. <https://doi.org/10.1145/3613904.3642902>
- [107] Allison Elliott Tew and Mark Guzdial. 2011. The FCS1: a language independent assessment of CS1 knowledge. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (Dallas, TX, USA) (*SIGCSE '11*). Association for Computing Machinery, New York, NY, USA, 111–116. <https://doi.org/10.1145/1953163.1953200>
- [108] Errol Thompson, Andrew Luxton-Reilly, Jacqueline L. Whalley, Minjie Hu, and Phil Robbins. 2008. Bloom's taxonomy for CS assessment. In *Proceedings of the Tenth Conference on Australasian Computing Education - Volume 78* (Wollongong, NSW, Australia) (*ACE '08*). Australian Computer Society, Inc., AUS, 155–161.
- [109] Anne Venables, Grace Tan, and Raymond Lister. 2009. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop* (Berkeley, CA, USA) (*ICER '09*). Association for Computing Machinery, New York, NY, USA, 117–128. <https://doi.org/10.1145/1584322.1584336>
- [110] Camilo Vieira, Alejandra J. Magana, Michael L. Falk, and R. Edwin Garcia. 2017. Writing In-Code Comments to Self-Explain in Computational Science and Engineering Education. *ACM Trans. Comput. Educ.* 17, 4, Article 17 (aug 2017), 21 pages. <https://doi.org/10.1145/3058751>
- [111] Lev S. Vygotsky. 1978. *Mind in Society: The Development of Higher Psychological Processes*. Harvard University Press, Cambridge, MA.
- [112] Wengran Wang, Archit Kwatra, James Skripchuk, Neeloy Gomes, Alexandra Milliken, Chris Martens, Tiffany Barnes, and Thomas Price. 2021. Novices' Learning Barriers When Using Code Examples in Open-Ended Programming. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Virtual Event, Germany) (*ITiCSE '21*). Association for Computing Machinery, New York, NY, USA, 394–400. <https://doi.org/10.1145/3430665.3456370>
- [113] Christopher Watson, Frederick WB Li, and Jamie L Godwin. 2012. BlueFix: using crowd-sourced feedback to support programming students in error diagnosis and repair. In *International Conference on Web-Based Learning*. Springer, 228–239.
- [114] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
- [115] Nathaniel Weinman, Armando Fox, and Marti A. Hearst. 2021. Improving Instruction of Programming Patterns with Faded Parsons Problems. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) (*CHI '21*). Association for Computing Machinery, New York, NY, USA, Article 53, 4 pages. <https://doi.org/10.1145/3411764.3445228>
- [116] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. 2024. *Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design*. Springer Nature Switzerland, Cham, 71–108. https://doi.org/10.1007/978-3-031-55642-5_4
- [117] Jeannette M. Wing. 2006. Computational thinking. *Commun. ACM* 49, 3 (mar 2006), 33–35. <https://doi.org/10.1145/1118178.1118215>
- [118] Xiaotong (Tone) Xu, Jiayu Yin, Catherine Gu, Jenny Mar, Sydney Zhang, Jane L. E, and Steven P. Dow. 2024. Jamplate: Exploring LLM-Enhanced Templates for Idea Reflection. In *Proceedings of the 29th International Conference on Intelligent User Interfaces* (Greenville, SC, USA) (*IUI '24*). Association for Computing Machinery, New York, NY, USA, 907–921. <https://doi.org/10.1145/3640543.3645196>
- [119] Litao Yan, Alyssa Hwang, Zhiyuan Wu, and Andrew Head. 2024. Ivie: Lightweight Anchored Explanations of Just-Generated Code. In *Proceedings of the CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) (*CHI '24*). Association for Computing Machinery, New York, NY, USA, Article 140, 15 pages. <https://doi.org/10.1145/3613904.3642239>
- [120] Ramazan Yilmaz and Fatma Gizem Karaoglan Yilmaz. 2023. Augmented intelligence in programming learning: Examining student views on the use of ChatGPT for programming learning. *Computers in Human Behavior: Artificial Humans* 1, 2 (2023), 100005. <https://doi.org/10.1016/j.chbah.2023.100005>
- [121] 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)*. 1–9. <https://doi.org/10.1109/FIE58773.2023.10343467>
- [122] Andrew Hicks Zhongxiu Liu, Rui Zhi and Tiffany Barnes. 2017. Understanding problem solving behavior of 6–8 graders in a debugging game. *Computer Science Education* 27, 1 (2017), 1–29. <https://doi.org/10.1080/08993408.2017.1308651>
- [123] Daniel Zingaro, Cynthia Taylor, Leo Porter, Michael Clancy, Cynthia Lee, Soohyun Nam Liao, and Kevin C. Webb. 2018. Identifying Student Difficulties with Basic Data Structures. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Espoo, Finland) (*ICER '18*). Association for Computing Machinery, New York, NY, USA, 169–177. <https://doi.org/10.1145/3230977.3231005>