Advancing Personalized, Engaging Learning and Assessment in the AI Era

by

Xinying Hou

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Information)
in The University of Michigan
2026

Doctoral Committee:

Associate Professor Barbara J Ericson, Chair
Professor Mark Guzdial
Professor Kenneth R Koedinger, Carnegie Mellon University
Associate Professor Steve Oney
Assistant Professor Xu Wang

Xinying Hou
xyhou@umich.edu
ORCID iD: 0000-0002-1182-5839

© Xinying Hou 2026

Dedication

To those who are still experiencing life and wish to make this world a better place, and to those who are afraid but still choose to try.

Acknowledgements

I am deeply appreciative of my PhD advisor, Dr. Barbara J. Ericson, whose guidance and warmth have sustained me both intellectually and personally throughout this journey. Her mentorship has shaped not only my growth as a researcher, but also my understanding of how to navigate academic life while cherishing the people we love. My deepest thanks also go to the members of my dissertation committee, Dr. Xu Wang, Dr. Steve Oney, Dr. Mark Guzdial, and Dr. Kenneth Koedinger, whose thoughtful feedback and suggestions have helped me sharpen and strengthen this dissertation.

I am fortunate to have developed my research interests within the field of education and technology, surrounded by so many bright and driven individuals who genuinely care about education, who research how technology can make it better, and who think deeply about every educational actor navigating an increasingly technological world. I have learned from a lot of outstanding mentors throughout my academic journey: Dr. John Stamper, Dr. Carol Forsyth, Dr. Jessica Andrews-Todd, Dr. Yang Jiang, Dr. Diego Zapata-Rivera, Dr. Arthur Graesser, Dr. Huy A. Nguyen, Dr. Steven Moore, Dr. Tomohiro Nagashima, Dr. Paulo F. Carvalho, Dr. J. Elizabeth Richey, Dr. Bruce M. McLaren, Dr. Vincent Aleven, and many more. They have helped me grow from someone who had little idea of what the intersection of education and technology could be, to someone who now has the confidence to pursue an assistant professor position on my own. I also thank the senior scholars who graciously shared their wisdom and guidance at key milestones along this journey, helping me navigate my future career and its many possibilities: Dr. Katie Cunningham, Dr. Anjali Singh, Dr. Qiao Jin, Dr. Yang Shi, Dr. David H. Smith IV, Dr. Benjamin Xie, Dr. David Weintrop, Dr. Ido Roll, and Dr. Eliane Wiese, among many others whose support left a lasting mark on my path.

I am deeply grateful for my wonderful collaborators, internship friends, workshop co-organizers, and labmates, with friendships that have naturally blossomed beyond academia and enriched my life in ways I never expected: Ruiwei Xiao, Qing Xiao, Runlong Ye, Shan Zhang, Dr. Majeed Kazemitabaar, Ying-Jui Tseng, Minh Tran, Jessica Yauney, Alex Chao, Michael Link, Grace Barkhuff, Katherine Braught, Elliot Roe, Dr. Zihan Wu, Dr. Tamara Nelson-Fromm, Dr. Aadarsh Padiyath, Dr. Emma Dodoo, Xingjian Gu, Seth Bernstein, and Ashlyn Campbell. Their creativity, care, humor, and insights have made this journey not only more productive, but richer and more joyful than I ever could have imagined. I also thank the friends woven into my jour-

ney from Pittsburgh to Ann Arbor, who shine their lights across tech, education, and academia, carrying my warmest memories with them.

I wish to acknowledge with gratitude the UMich CSCAR (Consulting for Statistics, Computing & Analytics Research) unit for their invaluable statistical consulting and expert guidance on the statistical modeling employed in this dissertation. I also want to thank the insightful reviewers in the Artificial Intelligence in Education, Computing Education, Learning Sciences, and Human-Computer Interaction communities. To every reviewer who took the time to engage with my work: while the reviews and feedback were at times challenging to receive, they have been invaluable in pushing my work forward and making it stronger. I also extend my thanks to every study participant who graciously gave their time and energy, as none of it would have been possible without their contributions.

I want to close with my appreciation for those who have been by my side throughout this journey. To the one who began this journey as my boyfriend and became my husband: you have been my calm in every storm, my strongest support, and the one I could always turn to. To Bloom and Mando, thank you for being cute and healthy, and for being there to remind me of the brighter side of life. To my cousin, thank you for always picking up, for the shared memories that grow with every passing year, and for all the little things only we understand. To my parents and family members, whose love, warmth, and kindness have carried me through in more ways than I can count. Finally, to myself, thank you for finding the strength to keep going even in the hardest times. Hard times pass, and good things come. I look forward to beginning the next chapter, with so much more of this world still left to explore.

This dissertation work was supported in part by the National Science Foundation under Grant No. 2143028, and by a Candidate Rackham Graduate Student Research Grant (2025–2026) from the University of Michigan. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

Table of Contents

Dedication ii

Acknowledgements iii

List of Figures ix

List of Tables xi

List of Acronyms xii

Abstract xiv

Chapter

1 Introduction **1**
- 1.1 Developing Theory-Informed Active Learning Mechanisms without Personalization 3
  - 1.1.1 Using Adaptive Parsons Puzzles to Scaffold Code-Writing 3
  - 1.1.2 Applying Prompted Self-Explanation for Improving Code Comprehension 4
- 1.2 Leveraging a Single AI-Powered System to Advance Personalization and Engagement 4
  - 1.2.1 Comparing the Effects of Active Puzzle Support versus Passively Delivered Code Solution 5
  - 1.2.2 Comparing the Effects of Personalized versus Common Solution-Based Puzzle Support 5
- 1.3 Orchestrating More Complex Personalization and Engagement Through Multi-Agent AI Architectures 6
  - 1.3.1 Exploring a Four-Agent Architecture with Two Student-Facing and Two Behind-the-Scenes Agents for Engaging Assessment 6
  - 1.3.2 Exploring a Dual-Agent Design for Personalized Non-Code Planning Puzzle Generation and Evaluation 7
- 1.4 Notes on Dissertation Preparation 7
- 1.5 Thesis Statement 8
- 1.6 Contributions 8

2 Related Work **9**
- 2.1 Theoretical foundations and frameworks 9
  - 2.1.1 Cognitive load theory 9

2.1.2 The zone of proximal development and scaffolding . . . 10
2.1.3 The ICAP framework . . . 11
2.1.4 Self-regulated learning and self-efficacy . . . 13
2.1.5 Program comprehension frameworks . . . 15
2.2 Pedagogical approaches grounded in learning theory . . . 17
2.2.1 The completion problem effect and Parsons puzzles . . . 17
2.2.2 The self-explanation effect and prompted self-explanation . . . 19
2.2.3 The use of conversation-based assessment . . . 21
2.2.4 Practice planning explicitly in problem-solving . . . 22
2.3 The growing use of LLMs for personalized education . . . 23
2.3.1 Leveraging a single LLM to power personalization . . . 23
2.3.2 Coordinating multiple specialized LLM agents . . . 24

3 Using Adaptive Parsons Puzzles to Scaffold Code-Writing 25
3.1 Introduction . . . 25
3.2 Study 1: Within-subjects think-aloud study . . . 27
3.2.1 Method . . . 28
3.2.2 Data Analysis . . . 30
3.2.3 Findings . . . 31
3.2.4 Discussion and Design Suggestions . . . 36
3.2.5 Limitations and Future work . . . 38
3.3 Study 2: Between-subjects classroom evaluation study . . . 38
3.3.1 System Interface . . . 39
3.3.2 Participants and Procedure . . . 39
3.3.3 Materials . . . 41
3.3.4 Results . . . 44
3.3.5 Discussion . . . 47
3.3.6 Limitations and Future work . . . 49
3.4 Conclusion . . . 49

4 Prompted Self-Explanation for Improving Code Understanding 51
4.1 Introduction . . . 51
4.2 Design of two SE prompt types with Parsons puzzles: Select and Fill . . . 53
4.3 In-depth within-subjects study: Student perceived values and preferences . . . 53
4.3.1 Method . . . 55
4.3.2 Findings . . . 56
4.4 Classroom between-subjects study: Student learning . . . 57
4.4.1 Methods . . . 58
4.4.2 Results . . . 59
4.5 Discussion . . . 60
4.6 Limitations and Future work . . . 61
4.7 Conclusion . . . 61

5 CodeTailor: LLM-Powered Personalized Parsons Puzzles for Engaging Support While Learning Programming 62
5.1 Comparing the Effects of Personalized Active Puzzle Support versus Passively Delivered Code Solution . . . 62
5.1.1 Introduction . . . 62
5.1.2 System Design . . . 65
5.1.3 Implementation . . . 68
5.1.4 Technical Evaluation: CodeTailor's Material Quality . . . 71
5.1.5 User Study . . . 72
5.1.6 Discussion and Future work . . . 79
5.1.7 Limitations . . . 82
5.2 Do Students Recognize Personalization? Comparing the Effects of Personalized versus Common Solution-Based Puzzle Support . . . 82
5.2.1 Introduction . . . 82
5.2.2 Methods & Materials . . . 83
5.2.3 Results . . . 85
5.2.4 Discussion, Future work, and Limitations . . . 87
5.3 Conclusion . . . 88

6 Exploring a Four-Agent Architecture with Two Student-Facing and Two Behind-the-Scenes Agents for Engaging Assessment 89
6.1 Introduction . . . 89
6.2 An LLM-powered Multi-Agent Architecture for Conversation-Based Assessment . 90
6.2.1 Agent Roles and Prompt Structures . . . 90
6.2.2 Hypothetical Student Use Cases . . . 94
6.3 Student Usability Evaluation Study . . . 95
6.3.1 Data Analysis . . . 95
6.3.2 Findings . . . 97
6.4 Discussion and Future work . . . 100
6.4.1 LLM-based multi-agent architecture to give humans more control in the AIED systems for assessment purposes . . . 100
6.4.2 Coordinating multiple agents introduces challenges that require fine-grained solutions . . . 100
6.5 Limitations . . . 100
6.6 Conclusion . . . 101

7 Exploring a Dual-Agent Design for Personalized Non-Code Planning Puzzle Generation and Evaluation 102
7.1 Introduction . . . 102
7.2 LadderPuzzle: Personalized Parsons Puzzles Powered by Multi-Agent LLMs for Practicing Solution Planning . . . 103
7.2.1 Overview of LadderPuzzle . . . 103
7.2.2 Implementation . . . 104
7.3 Evaluation Study . . . 105
7.3.1 Methods . . . 106

7.3.2 Findings . . . 109
7.4 Discussion . . . 113
7.4.1 More dedicated practice opportunities for higher-order computational skills 113
7.4.2 Connecting higher-order problem-solving to code implementation . . . 114
7.5 Limitations and Future work . . . 114
7.6 Conclusion . . . 115

8 Conclusions and Future Directions **116**
8.1 Summary of Contributions and Insights . . . 116
8.2 Future Directions . . . 118
8.2.1 Near-Term Directions . . . 119
8.2.2 Broader Directions for Future Research . . . 120

Bibliography . . . 122

List of Figures

Figure 1.1 This dissertation advances the design of active and personalized educational experiences within technology-enhanced learning environments. This objective is achieved through six investigations organized into three main approaches: 1) drawing on learning sciences theory to inform active learning; 2) introducing personalization through a single AI-powered system to support more engaging learning experiences; and 3) orchestrating more complex personalization and engagement through multi-agent AI architectures. 2

Figure 2.1 An illustration of the zone of proximal development: the range of tasks students can accomplish with support, but not yet independently. 10
Figure 2.2 The ICAP framework: modes of cognitive engagement (bold) and characteristic behaviors (italics), with corresponding terminology for different modes [38]. 13
Figure 2.3 The Block Model framework for program comprehension [270]. 16
Figure 2.4 Sample of a two-dimensional adaptive Parsons programming puzzle. . . . 18
Figure 2.5 The continuum of different self-explanation prompt formats [339]. 20

Figure 3.1 Screenshot of a write-code problem with unit test results. 26
Figure 3.2 Screenshot of using a Parsons problem to scaffold a write-code problem. . 27
Figure 3.3 Screenshot of the Adaptive Parsons problem interface. 29
Figure 3.4 Screenshot of the first part of the basic knowledge instruction. 43
Figure 3.5 Stacked bar chart of Parsons scaffolding usefulness ratings for each practice problem. 47

Figure 4.1 Example of an adaptive Parsons puzzle (Step 1). 52
Figure 4.2 Example of a Select prompt with immediate feedback and hints (highlights; not a whole question). 54
Figure 4.3 Example of a Fill prompt with immediate feedback and hints (highlights; not a whole question). 54
Figure 4.4 Example of two open-ended short-answer test questions. 58

Figure 5.1 The CodeTailor interface contains a programming area on the left and a pop-up personalized Parsons puzzle as support on the right. It provides timely feedback after checking (G) and allows students to copy (H) the finished solution (I). 63

Figure 5.2 Three example scenarios with CodeTailor. 67
Figure 5.3 Pipeline for generating a personalized correct solution from an incorrect code in CodeTailor (Stage 1). 68
Figure 5.4 The main prompt template used in CodeTailor. 69
Figure 5.5 Two practice versions: In Version A, students do two list exercises with personalized Parsons puzzles (CodeTailor) and two dictionary exercises with just the AI-generated code solutions. Version B has the reverse setting. 74
Figure 5.6 A complex shorthand one-line block in CodeTailor. 79
Figure 5.7 The student writes code (Step 1), requests help (Step 2), and receives a Parsons puzzle (Step 3). 83

Figure 6.1 This architecture includes four LLM agents that serve different roles: Expert Agent, Peer Agent, Formative Assessor, and Summative Assessor. A Watcher is used to supervise and manage this CBA flow. 91
Figure 6.2 Secondary-level students (N=37) completed a thunderstorm forecasting task where they watched videos, engaged with CBAs, completed concept maps and other activities about storm formation. This work focuses on one CBA. 96
Figure 6.3 Standardized average cohesion scores and an example of Student Answer-Follow-up Agent interaction for each speech act category. 98

Figure 7.1 Screenshot of a LadderPuzzle: a student's initial proposed plan (left); the corresponding personalized mixed-up planning puzzle (right). 104
Figure 7.2 An open-ended planning question. 106
Figure 7.3 Overview of the Study Procedure. 107

List of Tables

Table 3.1 Distractor Locations by Version 30
Table 3.2 Four Write-code Practice Problems 39
Table 3.3 Questions about student general CS self-efficacy level 40
Table 3.4 Questions and ratings on prior-practice knowledge levels in Python `class` 41
Table 3.5 Pre-practice knowledge level and general CS self-efficacy level by condition, reported in *M (SD)*, *Mdn* (25th percentile - 75th percentile) format . . . 44

Table 4.1 Test Performance Statistics by Condition (*Mean*, *SD*) 59

Table 5.1 Descriptive statistics on RQ1 and RQ2 metrics, reported in Mean (SD), Median format 76
Table 5.2 Survey questions on students' perceived solution alignment and scaffolded solution understanding 84
Table 5.3 Results of the Mixed Linear Model (Perceived Solution Alignment as the dependent variable) 86
Table 5.4 Results of the Mixed Linear Model (Scaffolding Puzzle Solution Understanding as the dependent variable) 87

Table 6.1 Guidelines used for classifying student responses into speech act categories 93
Table 6.2 Watcher's agent assignment rules and instructions for different categories (<student>, <answer>, <question>, and <domain>are placeholders) . . 94
Table 6.3 Results of the Mixed Linear Model (The reference level is CORRECT) 99

Table 7.1 Example student responses to an open-ended planning question. 107
Table 7.2 Problems used in the practice and mid-progress assessment 108
Table 7.3 Survey questions used in the LadderPuzzle evaluation study (7-point Likert scale: 1 = "strongly disagree" and 7 = "strongly agree") 109

List of Acronyms

ACM Association for Computing Machinery

AI Artificial Intelligence

AIED Artificial Intelligence in Education

API Application Programming Interface

AST Abstract Syntax Tree

CBA Conversation-Based Assessment

CI Confidence Interval

CLES Common Language Effect Size

CLT Cognitive Load Theory

CP Common solution-based Parsons Puzzles

CS Computer Science

CSV Comma-Separated Values

ECD Evidence-Centered Design

GPT Generative Pre-trained Transformer

HTML HyperText Markup Language

ICAP Interactive-Constructive-Active-Passive

IRB Institutional Review Board

ITS Intelligent Tutoring System

JSON JavaScript Object Notation

LLM Large Language Model

M Mean

Mdn Median

MOOC Massive Open Online Course

NLP Natural Language Processing

NP No-Parsons-Scaffolding

PP Personalized solution-based Parsons Puzzles

PS Parsons-Scaffolding

PZ Solve the Puzzle only

PZ+SE Solve a Puzzle with a Select SE prompt

RQ Research Question

SD Standard Deviation

SE Self-Explanation

SOLO Structure of Observed Learning Outcome

SRL Self-Regulated Learning

US United States

USD United States Dollar

ZPD Zone of Proximal Development

Abstract

Conventional learning environments often fail to engage students or meet their varied needs. Emerging technologies offer opportunities to foster deeper engagement and personalization in education at scale. This dissertation advances the design of engaging and personalized educational experiences within technology-enhanced learning environments through a series of empirical investigations and design explorations, organized into three approaches. The first approach drew on learning sciences theories to inform active learning design. I incorporated adaptive mixed-up code (Parsons) puzzles to scaffold novices' short programming problem-solving, and paired them with prompted self-explanation questions to support code comprehension. This approach focused on active learning design and did not yet incorporate personalization. The second approach introduced personalization via a single AI-powered system to support engaging learning experiences. I designed and evaluated CodeTailor, an interactive system that leveraged a large language model to deliver real-time mixed-up code puzzles tailored to students' incorrect code. The third approach explored multi-agent AI architectures for tasks that benefit from decomposition into subtasks, where several specialized agents coordinate to deliver personalized, engaging experiences. I designed a four-agent architecture for conversation-based formative assessment and developed LadderPuzzle, a dual-agent design that produced personalized, non-code mixed-up puzzles to make solution planning an explicit and deliberate practice step. This dissertation provided evidence for the effectiveness of using Parsons puzzles to scaffold short programming problems, and evidence that retrospective prompted self-explanation improved code comprehension without additional mental effort. It introduced CodeTailor, which students found more engaging than passively delivered solutions and recognized as better aligned with their own approach than common solution-based alternatives. It presented a four-agent architecture for conversation-based assessment, in which student-facing agents worked to keep the dialogue on topic while behind-the-scenes agents collected assessment evidence at varying granularity. Finally, it delivered LadderPuzzle, which students rated as more useful and more effective for improving solution quality than open-ended planning questions. Students also expressed willingness to use it for explicit solution planning practice in future learning. Overall, this dissertation contributes to the design of effective technology-enhanced learning environments, supporting learners as they build foundational knowledge and skills through engaging, personalized experiences.

Chapter 1

Introduction

Thousands of years ago, the ancient Chinese philosopher Confucius put into practice the idea of adapting instruction to the individual needs and abilities of each student [149, 292]. This can be seen as one of the earliest notions of personalized learning. As societies progress and economic needs change, education has transformed from an exclusive privilege into a shared right, one now charged with the vital responsibility of fostering universal enlightenment [100]. Nevertheless, instructional resources often remain limited [53, 135]. Given the gap between the rising demand for education and the limited availability of instructional resources, education has traditionally relied on standardized instruction [95] and lecture-centered classrooms [293]. Students often spend hours sitting in classrooms, passively listening to lectures without actively engaging in practice or hands-on work. Similarly, traditional large-scale assessment has relied on uniform tests whose feedback rarely extends beyond a final score [82, 295, 298, 334]. However, such instruction and assessment practices often prevent students from reaching their potential, contributing to disengagement [169, 321] and high dropout rates [247, 343].

Technology can help address these challenges by enabling more personalized [23] and active learning experiences [41, 298, 345], to turn the centuries-old educational philosophy into reality, even in classrooms with high student-to-teacher ratios. With this in mind, this dissertation is driven by an overarching motivation: **How can technology-enhanced learning environments, ranging from established pedagogical mechanisms to advanced AI-based architectures, be designed to foster engagement in student learning?**

To that end, this dissertation draws on learning sciences theories and emerging technologies to advance personalized and active learning experiences for learners in the early stages of knowledge and skill acquisition. Organized around three core approaches (Figure 1.1), it progresses from theory-informed active learning mechanisms, through single AI-powered personalized active learning, to multi-agent AI architectures to facilitate more complex personalized active learning experiences. Specifically, first, I investigated the use of mixed-up code puzzles and prompted self-explanation questions to support programming novices in completing short

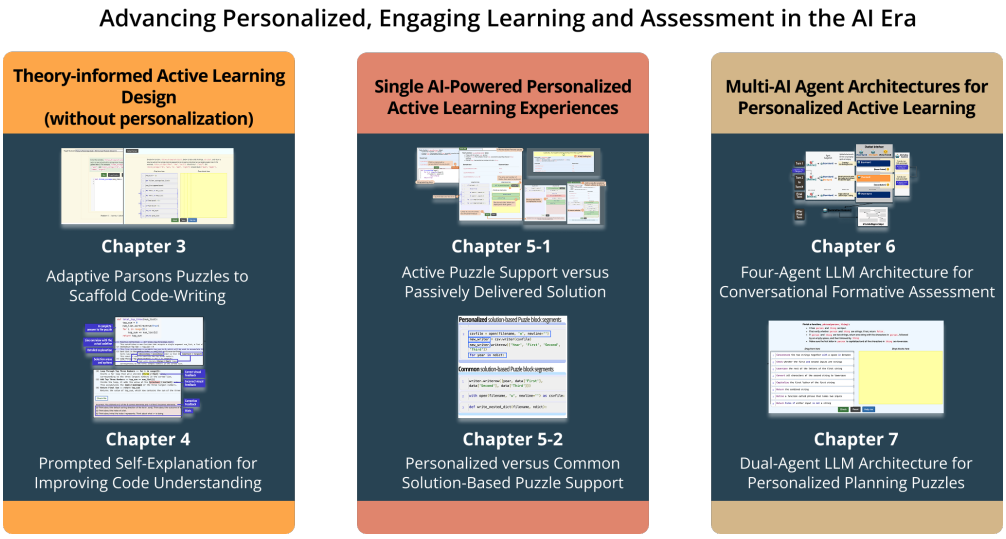


Figure 1.1: This dissertation advances the design of active and personalized educational experiences within technology-enhanced learning environments. This objective is achieved through six investigations organized into three main approaches: 1) drawing on learning sciences theory to inform active learning; 2) introducing personalization through a single AI-powered system to support more engaging learning experiences; and 3) orchestrating more complex personalization and engagement through multi-agent AI architectures.

programming tasks and deepening their code understanding. Second, I incorporated single LLM-powered personalization into mixed-up code puzzles to better support programming novices in completing short programming tasks. Finally, drawing on collaborative specialized AI agent design, I employed multi-agent architectures in two studies. One facilitated personalized and interactive formative assessment, while the other supported novices in problem-solving planning through personalized mixed-up planning puzzles. Collectively, these investigations examine AI-powered personalization and active learning across diverse educational contexts, offering empirical guidance for future development. It is worth noting that this dissertation is situated at a transformative moment in education, as AI technologies grow increasingly accessible and their potential to reshape learning continues to unfold. As part of this early wave, this dissertation makes timely and grounded contributions toward the thoughtful integration of AI-enhanced approaches in education. The following provides an overview of each chapter, outlining how the studies build upon one another to advance personalized and active educational experiences (Figure 1.1).

In the subsections that follow, I briefly present the six investigations structured around three main approaches: 1) developing theory-informed active learning mechanisms without personalization; 2) leveraging a single AI-powered system to advance personalization and engagement; and 3) orchestrating complex personalization and engagement via multi-agent AI architectures.

1.1 Developing Theory-Informed Active Learning Mechanisms without Personalization

Learning sciences is an interdisciplinary field that investigates how learning occurs and how learning environments can be designed to promote it [267]. Early programming courses (e.g., Computer Science 1 (CS1) and CS2) require novices to master a range of distinct skills, such as code comprehension and code writing [271, 341], and lay the groundwork for future coursework and the development of expertise. Despite this critical role, novice learners often struggle to acquire these skills in these courses [142, 327]. This makes introductory programming a context where learning sciences theory can meaningfully inform the design of learning mechanisms that better support students [24]. In **Chapter 3** and **Chapter 4**, I investigated theory-informed learning mechanisms to support novice programmers in code writing and code comprehension (Figure 1.1).

1.1.1 Using Adaptive Parsons Puzzles to Scaffold Code-Writing

In **Chapter 3**, building on existing learning sciences theories such as cognitive load theory [290] and the zone of proximal development [265, 317], I introduced Parsons puzzles to scaffold novice programmers who were struggling while solving write-code problems. Parsons puzzles are active practice tasks where students arrange mixed-up code blocks into the correct order to solve a problem [217]. To understand novices' perceptions and the effectiveness of Parsons puzzles as scaffolding in learning, I conducted two studies in which novices were given equivalent Parsons puzzles as optional scaffolding while solving write-code problems.

In a think-aloud study followed by retrospective interviews with undergraduate students, I investigated when, why, and how students used the Parsons problems as well as their perceptions of the benefits and challenges [112]. Results showed that students utilized the Parsons puzzles before writing a solution to get ideas about where to start, while writing a solution to overcome roadblocks, and after writing a solution to debug errors and look for better strategies. Semi-structured interviews with the same group of undergraduate students provided evidence that using Parsons puzzles to scaffold write-code problems helped students to reduce the difficulty, learn problem-solving strategies, and refine their programming knowledge. However, some students found them less useful if the Parsons solution did not match their approach or if they did not understand the solution.

A between-subjects classroom study was then conducted with undergraduate programming learners on a topic in which they had limited code-writing experience. Students were randomly assigned to one of two conditions. Students in one condition practiced writing code without any scaffolding, while students in the other condition were provided with scaffolding in the form of

an equivalent Parsons puzzle. Students who received scaffolding achieved significantly higher practice performance and in-practice problem-solving efficiency compared to those without any scaffolding. This study contributes classroom evidence for the effectiveness of providing Parsons problems to scaffold students on write-code tasks.

1.1.2 Applying Prompted Self-Explanation for Improving Code Comprehension

Although code writing has long been a core skill in introductory programming education [242], the growing use of AI tools for code generation has highlighted the importance of code comprehension [43, 238], which is essential for validating AI-generated code [81, 200]. To deepen students' understanding of code after solving a problem, I developed a novel pedagogical approach that pairs Parsons puzzle problems with prompted self-explanation (SE) questions as a post-solution step (**Chapter 4**). I investigated two different designs for prompted self-explanation questions and conducted classroom evaluations to assess their effectiveness [116].

Inspired by existing self-explanation literature [339], I developed two types of prompted SE questions, Select and Fill, both designed to have students complete a full code explanation of the Parsons puzzle they had just solved. In Select, students choose from a set of options, while in Fill, students type in their answers. I conducted a within-subjects think-aloud study to understand whether novices found this approach valuable, which type of SE questions they preferred, and the reasons for their preference. All novices valued having a paired prompted SE question after a Parsons puzzle. Of the two SE types, the Select format was generally preferred over the Fill format due to its reduced ambiguity and lower frustration. I then conducted a between-subjects study with 110 students in an intro-level CS course: one condition solved Parsons puzzles only, while the other solved pairs of puzzles and Select SE questions. Because the current Fill design may cause high frustration and lead students to disengage from SE questions, this classroom study included only Select SE questions. The group that solved pairs of Parsons puzzles and Select SE questions showed significantly greater learning gains from pretest to posttest while reporting similar in-practice mental effort. This work offers a lightweight approach to improve students' code understanding in the context of early CS courses.

1.2 Leveraging a Single AI-Powered System to Advance Personalization and Engagement

With the integration of approachable generative AI into education, new opportunities for personalized learning experiences have emerged [194, 227]. In **Chapter 5**, I integrated AI-powered

personalization into Parsons puzzles and developed CodeTailor. CodeTailor is an interactive system that leverages an LLM to deliver real-time, on-demand personalized puzzles as programming support. I first conducted a technical evaluation to assess the quality of the generated personalized puzzles. Results demonstrated that CodeTailor could deliver high-quality (correct, personalized, and concise) Parsons puzzles based on the student's incorrect code. After that, I evaluated CodeTailor's effectiveness from the perspectives of both *active engagement* and *personalization*.

1.2.1 Comparing the Effects of Active Puzzle Support versus Passively Delivered Code Solution

To evaluate CodeTailor's active engagement capabilities compared to receiving a static, personalized code solution (a typical output students receive from generative AI tools), I conducted a within-subjects study with novice programmers. Participants perceived CodeTailor as more engaging than just receiving an LLM-generated solution (the baseline condition). In addition, participants applied more supported elements from the scaffolded practice to the posttest when using CodeTailor than with the baseline. Overall, most participants preferred using CodeTailor over just receiving the LLM-generated code for learning. Qualitative observations and interviews also provided evidence for the benefits of CodeTailor, including thinking more about solution construction, fostering continuity in learning, promoting reflection, and boosting confidence.

1.2.2 Comparing the Effects of Personalized versus Common Solution-Based Puzzle Support

In addition, to evaluate CodeTailor's personalization capabilities, I conducted a between-subjects classroom study with undergraduate programming learners, comparing personalized Parsons puzzles versus non-personalized ones as code-writing support. Results indicated that students who received a personalized puzzle rated the alignment of the puzzle solution with their own approach as 16% higher than when they received a common-solution puzzle. Early results provided evidence that personalized puzzles improved perceived solution understanding via higher perceived alignment with students' approaches. However, nearly half of the scaffolding requests occurred before the student wrote any code, which limited the opportunity for personalization.

1.3 Orchestrating More Complex Personalization and Engagement Through Multi-Agent AI Architectures

Educational activities typically demand careful orchestration [266] and a high standard of correctness [278], both of which can exceed the capabilities of a single LLM. For example, a single LLM may fail to self-check, leading to hallucinations [120, 342], and it often struggles to simultaneously fulfill the multiple specialized roles required for effective orchestration [354]. These problems have led to the development of multi-agent designs, which combine the capabilities of several agents, each with its own roles [106], strategies [34], expertise [110], and/or functionalities [132, 294]. I explored two designs: the first uses four agents to orchestrate a complex, conversation-based assessment (**Chapter 6**); the second employs a dual-agent structure to deliver natural-language puzzles as explicit practice before coding (**Chapter 7**).

1.3.1 Exploring a Four-Agent Architecture with Two Student-Facing and Two Behind-the-Scenes Agents for Engaging Assessment

Formative assessment plays an important role in helping novices monitor their progress and identify learning gaps early enough to address them. However, learners can disengage during low-stakes formative assessments [4, 304, 335, 344, 346, 348], missing valuable opportunities to improve before high-stakes testing [104, 123]. Conversation-based assessments (CBA), which evaluate student knowledge through interactive dialogues with artificial agents on a given topic, can help address non-effortful formative test-taking and the lack of adaptability in traditional assessment. In **Chapter 6**, I explored four specialized AI agents that power a conversation-based assessment, fostering student engagement and effort through a personalized, interactive approach to low-stakes formative assessment. This architecture includes four LLM agents: two student-facing agents and two behind-the-scenes agents. The two student-facing agents (the expert and peer agents) focus on asking leading questions and engaging students in providing thoughtful answers. The two behind-the-scenes agents (a formative assessor and a summative assessor) focus on analyzing and collecting evidence statements throughout the conversation. All agents are monitored by a non-LLM agent (the Watcher), which manages the assessment flow through updated instructions to agents and turn control. This architecture was evaluated in the context of inquiry-based learning with secondary-school students, focusing on the agents' capabilities. Results showed that the two student-facing agents were able to differentiate diverse student responses and tried to keep the conversation on topic. How students responded to the redirections was left to future work. The two behind-the-scenes agents were able to analyze student answers in real time and collect evidence for future diagnosis. The work concluded by

examining the strengths and limitations of this multi-agent architecture with suggestions for future work.

1.3.2 Exploring a Dual-Agent Design for Personalized Non-Code Planning Puzzle Generation and Evaluation

Rapid advances in AI have transformed computing learning, shifting its emphasis toward higher-order problem-solving skills such as planning, the process of coming up with algorithmic steps to address a computational problem [197, 333]. Despite its importance, support for solution planning remains limited in early computing instruction. In **Chapter 7**, I introduced LadderPuzzle, a personalized non-code Parsons puzzle designed to support solution planning as a deliberate and explicit practice step. A LadderPuzzle adopts a dual-agent LLM architecture in which a generation agent produces a personalized natural language step-by-step plan based on each student's proposed plan, and an evaluation agent verifies whether the generated plan can be implemented as a correct code solution. A within-subjects study with novice programming learners found that LadderPuzzles were rated as significantly more useful and more effective at improving solution quality than open-ended planning questions. Students expressed clear willingness to engage with both planning activities in the future. Some further suggested that the choice between activities should depend on expertise level, recommending LadderPuzzle for novices and open-ended questions for more advanced learners. These findings provide evidence for the value of personalized, puzzle-based planning exercises in early computing instruction.

1.4 Notes on Dissertation Preparation

Portions of this dissertation are adapted from prior work by the author and collaborators. I am the first author on all work included in this dissertation. To acknowledge co-authorship, I use the first-person plural ("we" / "our") where appropriate. See each chapter for information about where it was published. Portions of this dissertation were prepared with assistance from generative AI tools. Where used, they assisted with brainstorming throughout the process, with code support (e.g., debugging, syntax clarification), and with manuscript refinement for clarity, spelling, and grammar. All suggestions and generated content were reviewed, evaluated, revised or discarded as needed, and verified by the author.

1.5 Thesis Statement

Active engagement in learning tasks is essential for students to develop knowledge and skills toward expertise. The projects described above motivate the following thesis statement:

> *By designing theory-grounded active learning mechanisms and personalizing them to the learner, we can foster engagement and improve learning across different early-stage educational contexts.*

Specifically, the following chapters examine three design approaches: theory-informed active learning mechanisms without personalization, single-AI systems that personalize the mechanisms, and multi-agent architectures for richer personalization. Together, they provide empirical evidence and design insight for creating more engaging technology-supported learning experiences.

1.6 Contributions

This dissertation makes the following contributions toward the design of personalized, engaging learning experiences in technology-enhanced environments. (1) Parsons programming puzzles as optional scaffolding can support short programming problem-solving, with empirical evidence that they improved novices' code-writing performance and problem-solving efficiency without reducing posttest performance (Chapter 3). (2) Retrospective self-explanation prompts after problem-solving can support novices' code understanding, with empirical evidence that they improved code comprehension without additional mental effort and that novices perceived them as useful (Chapter 4). (3) Parsons programming puzzles can be powered by a single LLM (CodeTailor) to generate correct, personalized, and concise support from students' own code. Students found this support more engaging than a passively delivered solution, and recognized the personalization, rating the puzzle as better aligned with their own approach than a common solution-based puzzle (Chapter 5). (4) A four-agent LLM architecture can support personalized conversation-based formative assessment, with empirical evidence that student-facing agents worked to keep the dialogue on topic while behind-the-scenes agents analyzed answers in real time to collect evidence for future diagnosis (Chapter 6). (5) Non-code Parsons puzzles powered by a dual-agent LLM architecture (LadderPuzzle) can support explicit solution planning. Students rated LadderPuzzles as more useful and more effective for improving solution quality than open-ended planning questions, and expressed willingness to engage with explicit planning practice in future learning (Chapter 7).

Chapter 2

Related Work

To situate the work that follows, this chapter reviews prior research that informs the theoretical and empirical foundations. It is organized in three parts, moving from theory to pedagogy to technology. The theory part covers relevant foundational theories and frameworks. The pedagogy part reviews pedagogical mechanisms grounded in these theories. The technology part examines recent work on applying LLMs to deliver personalized educational experiences. Portions of this chapter draw on previously published materials presented in Chapters 3-6, as well as earlier PhD milestones, including the pre-candidacy proposal, pre-candidacy paper, and preliminary literature review.

2.1 Theoretical foundations and frameworks

2.1.1 Cognitive load theory

Cognitive load theory (CLT) explains the relationship between human cognitive architecture and learning, and what that implies for instructional design [198, 290]. This theoretical framework considers the complexity of the information to be learned and how humans process it to guide effective instructional design practices [312]. According to cognitive load theory, the human mind has multiple memory stores, including working and long-term memory [269]. However, working memory has limited capacity and duration and can be overloaded when processing new information. Cognitive load refers to the working memory resources required when learning new information [289]. The updated cognitive load model distinguishes two core categories: intrinsic and extraneous cognitive load [290]. Intrinsic cognitive load is caused by the inherent difficulty of the material. Such cognitive load can be managed with strategies like segmenting (breaking complex information into smaller chunks) and pre-training (introducing key terms before the actual main session) [46]. Extraneous cognitive load is determined by the delivery of instructional information and the tasks expected of the learner [290]. Germane processing refers to the actual working memory resources committed to dealing with intrinsic cognitive load [290], and tends

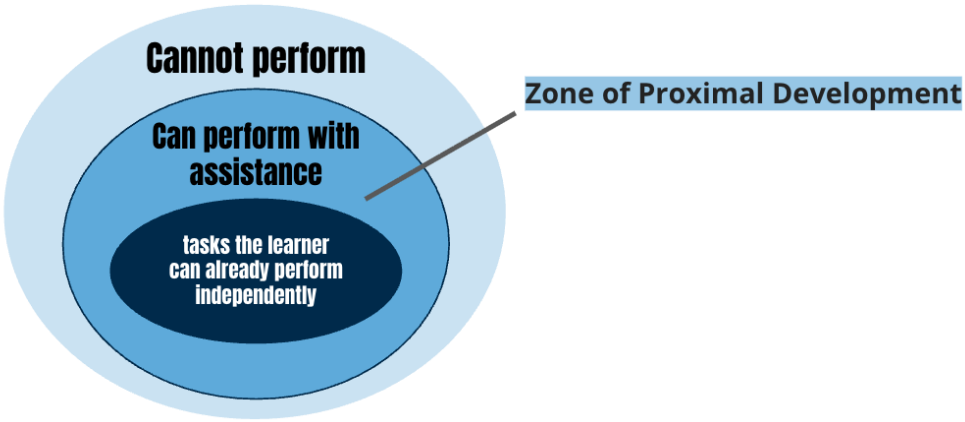


Figure 2.1: An illustration of the zone of proximal development: the range of tasks students can accomplish with support, but not yet independently.

to play a minor role in the new model [70].

With the development of digital learning, some researchers have reconceptualized CLT specifically around interactive learning media. For example, rather than showing information in the most straightforward way (like directly displaying a solution), learning and motivation might be more enhanced by low and middle levels of interaction while preventing cognitive overload [282].

2.1.2 The zone of proximal development and scaffolding

In educational contexts, the term scaffolding is defined as the timely, well-matched support that allows students to complete tasks or finish practice they could not yet achieve independently [54, 137, 184, 337]. These supports can help students finish a task or build new understanding so that they can perform comparable activities on their own later. Bloom demonstrated that one-on-one human tutoring helped students improve their learning by two standard deviations over classroom instruction [29]. Scaffolding can be linked with Vygotsky's theory of the zone of proximal development (ZPD; Figure 2.1) [317]. The 'zone' refers to a 'space' encompassing behaviors, competencies, and potential problems [291]. Vygotsky (1962, as cited in [274]) introduced this concept in the context of assessment, criticizing traditional testing for reflecting only learners' current level of achievement [274], and recommending instead that they be assessed while working with an adult or more capable peer (More Knowledgeable Other) to highlight the potential for future development [291]. This approach made it possible to identify what learners could accomplish with appropriate support, thereby mapping the contents of their ZPD [291]. Since its introduction, the concept of ZPD has been extended to a range of educational domains and learner populations [326].

Based on ZPD, the assistance students receive should (1) enable them to achieve, in collabo-

ration, what they cannot yet do alone and (2) match the learner's own purposes [326]. Later, Pea (2004) identified two key properties of effective scaffolding [221]:

> "1. Channeling and focusing: Reducing the degrees of freedom for the task at hand by providing constraints that increase the likelihood of the learner's effective action; recruiting and focusing attention of the learner by marking relevant task features (in what is otherwise a complex stimulus field), with the result of maintaining directedness of the learner's activity toward task achievement.
>
> 2. Modeling: Modeling more advanced solutions to the task." [221, p. 432]

The theory of ZPD has motivated a range of educational technology designs. For instance, intelligent tutoring systems, which model human tutoring to give students timely and tailored guidance and feedback, illustrate how the ZPD can inform the design of educational technology [45, 160, 202]. Specifically, Ecolab [171, 172] applied the ZPD to develop an interactive learning environment that helped children learn about food webs and chains. Guided by the ZPD, the system aimed to provide appropriately challenging activities while adjusting the amount of assistance [171, 172]. The ZPD has also informed adaptive online learning systems. For example, prior work developed an adaptive scaffolding e-learning system that personalized learning by providing help and tailoring task difficulty to each learner [36]. In addition, a personalized recommendation system used the idea of the ZPD to guide the selection of recommended items during adaptive learning [52].

2.1.3 The ICAP framework

This dissertation adopts the term "active learning" to refer broadly to instructional approaches that foster deeper and more meaningful engagement with learning materials [167]. Active learning refers to instructional methods that engage students cognitively and meaningfully with the instructional materials in a learning task [41, 244]. Instead of just passively receiving information, students interact with the instructional materials and actively digest the content [30]. For any instructional task, students can engage in different modes of activities or different activities within the same mode.

This dissertation draws on the ICAP framework primarily for its central distinction among engagement modes and its links between those modes, instructional activity types, and student behaviors. According to the ICAP (Interactive-Constructive-Active-Passive) framework (Figure 2.2) from Chi and Wylie [41], as students become more engaged with the learning materials and transition from passive to active to constructive to interactive, their learning will increase [40, 41].

The ICAP framework divides students' cognitive engagement modes into four categories: *passive* (just receiving the material), *active* (actively moving and participating), *constructive* (self-explaining and generating), and *interactive* (peer discussion), from least effective to most effective [41]. Concretely, *passive* engagement is when learners receive information from instructional materials without actively doing anything else related to learning. Learners' engagement can be considered *active* if they undertake overt physical actions or manipulation. *Constructive* behaviors involve learners generating or producing additional external outputs or products beyond what was provided in the learning materials. Lastly, *interactive* behaviors are dialogues that meet two criteria: (a) both partners' utterances are constructive, and (b) there is sufficient turn-taking. Apart from being cognitively engaged with instructional resources in various modes, students might also be disengaged, such as being off-task or exploiting help from the scaffolding [6, 38, 94].

The ICAP framework has informed the design of various educational technologies and has been used to understand how students interact with learning activities both offline and online. For example, a study examining technology-enhanced courses in higher education found that the technology-enhanced interactive learning activities strengthened students' perceived learning, while passive activities had negative effects on affective-motivational outcomes [325]. The ICAP framework has been used to investigate students' interaction in online learning environments [67, 281, 303, 305, 320]. For instance, one study with older adults used the ICAP framework to examine their engagement with interactive video, finding that although participants were positive, most engagement stayed at the active level, with few constructive or interactive behaviors [303]. Such results suggested that deeper learning still requires deliberate scaffolding [303]. Another study coded students' actions in an online Python programming environment into different cognitive engagement levels using ICAP and found that the constructive and passive levels significantly predicted learning gains [281]. However, the original ICAP framework focuses on in-person settings and relies on the physical presence of teachers to make student behaviors visible [41, 229]. This limitation becomes especially apparent in online and technology-enhanced learning environments that lack a physical teacher presence. To address this, later work proposed a revised ICAP framework that makes formative feedback integral to each engagement mode for online learning activities [229]. In this context, an activity was categorized as constructive if it required student input and then provided feedback on that input [229]. Examples included online quizzes with feedback and drag-and-drop concept checks in which students placed each term in its correct location [229]. In contrast, an activity was categorized as active if it did not require students to integrate the information further and provided no feedback [229]. Examples included taking one's own study notes or simply using a "click and reveal" feature [229].

The ICAP framework has also guided educational technology designs [26, 155, 353]. One study drew on ICAP to build an AI-based robot for a blended learning setting, and an evaluation study

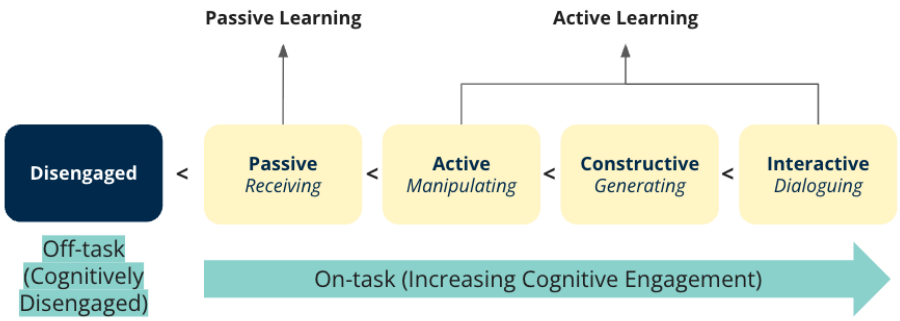


Figure 2.2: The ICAP framework: modes of cognitive engagement (bold) and characteristic behaviors (italics), with corresponding terminology for different modes [38].

showed that the AI-based robot significantly enhanced students' motivation, satisfaction, and computational thinking [155]. Another design drew on the ICAP framework to build an AI-powered interactive learning system aimed at deepening cognitive engagement through interactive activities [26]. In the context of early childhood learning, the ICAP framework informed the design of three AI-powered narrative supports, with iterative and autonomous support both improving children's narrative skills during active engagement [353].

2.1.4 Self-regulated learning and self-efficacy

Self-regulated learning (SRL) describes how learners actively set, monitor, adapt, and sustain their cognitions, feelings, and behaviors in their own learning process to reach their personal goals [361, 363, 366]. There are three key features in the definitions of students' self-regulated learning, including *"their use of self-regulated learning strategies, their responsiveness to self-oriented feedback about learning effectiveness, and their interdependent motivational processes"* [360, p. 6]. The use of self-regulated learning strategies is essential to students' academic performance [359, 360] and their continued educational success [15]. Zimmerman (2000) and Zimmerman & Moylan (2009) pictured SRL as a cyclical process with three phases, including forethought phase, performance phase and self-reflection phase [215, 362, 365]. The forethought phase includes two task-analysis processes, goal setting and strategic planning, along with several self-motivation beliefs. The performance phase involves several self-control strategies and self-observation. Finally, the self-reflection phase comprises self-judgment and self-reaction [215, 362, 365].

Self-efficacy, one of the self-motivation beliefs in Zimmerman's forethought phase [362, 365], describes people's perceptions of their own ability to complete a task [17]. Self-efficacy is important in education as individuals' self-efficacy can influence their willingness to put effort into a

task, decisions about future involvement in a task, and attitudes when facing obstacles [17]. For example, students with high self-efficacy are usually more enthusiastic about participating in and completing learning tasks than students with low self-efficacy. When facing difficulties, students who believe in their abilities (high self-efficacy) do not avoid difficult tasks but view them as challenges that must be overcome. Self-efficacy also predicts the usage of strategies during learning [32, 111]. Higher academic self-efficacy is linked to more effective learning practices such as using cognitive strategies, self-monitoring, and goal setting [228, 332, 364]. Self-efficacy has been shown to be an essential factor in predicting student achievement across many different subjects, such as math, science, and writing [213, 214]. A previous review conducted a meta-analysis of 109 studies and found that college students with high academic self-efficacy were more likely to have better college outcomes (performance and persistence) than students with low self-efficacy [259].

Self-efficacy has become one of the most studied constructs to understand programming learning outcomes and persistence in computing careers [158, 234]. Specifically, researchers have found that self-efficacy is positively connected with two distinct programming course outcomes: performance in debugging tasks and the overall course grade [329]. In another study, researchers interviewed 31 students at two public universities and discovered that one crucial aspect in their decision to major in CS was their perception of their CS ability (self-efficacy in CS) [153]. Similarly, researchers applied self-efficacy surveys and found students with higher self-efficacy were more likely to enroll in a computer science course in college [195]. Given that self-efficacy is important for success in CS learning, the next step is to discover how to help students develop and maintain their self-efficacy.

Students' self-efficacy grows as a result of positive learning experiences, and increased self-efficacy facilitates learning performance [216]. Therefore, previous studies have explored several scaffolding approaches to provide a positive learning experience and improve student self-efficacy in CS learning. For example, one study found that students who received peer instruction in an introductory programming course had significantly higher post-course self-efficacy than those who received traditional instruction such as lectures and assignments [367]. In addition, another study applied three different types of scaffolding methods for programming learning in the C language, and found that eye-tracking scaffolding and peer scaffolding methods significantly enhanced learners' self-efficacy [285]. Further research also looked into whether the advantage of visual programming languages could benefit students with varying levels of self-efficacy, and found that such scaffolding had a stronger impact on student programming learning in low and moderate self-efficacy groups [302].

2.1.5 Program comprehension frameworks

Program comprehension (also called code comprehension) involves understanding the elements, structure, execution behavior, and purpose of a program both from the programmer's perspective and in the domain context [124]. Such comprehension is the process of constructing a mental model of a program. Novice programmers should develop program comprehension skills while learning to code because it enables them to effectively read, reason about, and reflect on code written by others as well as their own [124]. Typical program comprehension tasks can range from explaining code to annotating code to modifying code, each requiring different levels of engagement. When explaining or articulating, the learner reads the code and then explains what they think the code is doing. Annotation tasks involve adding comments, or highlights, or creating external representations for the provided code piece such as trace tables. Modification tasks involve adjusting or reworking the code to fix a bug, improve readability, or add a feature [124]. These three frameworks have been used to design explanation tasks in programming learning and to assess the quality of students' explanations: the Block Model [270], SOLO Taxonomy [159], and Pennington's program comprehension model [226, 271].

The Block Model The Block Model (Figure 2.3) provides a framework to classify program comprehension on two dimensions: from single elements to the whole program, and simultaneously from text surface to the purpose [270]. This model views a program in two ways: (1) examining it at different levels, from individual expressions to the relations between blocks and to the entire program; and (2) considering it in three dimensions, including the program code, program execution, and the program's overall function and purpose. The model distinguishes three levels: atoms, blocks, and the overall program. Atoms are basic elements like expressions or simple command lines, while blocks are sequences of related statements. The top level is the overall program, including relationships between blocks and the purpose behind them. These levels increase in structural complexity and focus on the sequence of steps in the comprehension process, which is flexible, context-bound, bottom-up, and cyclic [270, 271]. The columns reflect the different dimensions of a program: the surface text, the execution, and the intention. Based on the original block model, scholars also made a distinction between *denotation* and *connotation* when extracting knowledge from textual language. Denotation refers to the meaning of an element in a context-free sense, like the meaning of a word as described in a dictionary. In contrast, connotation refers to discerning the meaning in the concrete context of its use [124].

SOLO Taxonomy The SOLO (structure of observed learning outcome) taxonomy is a model that describes levels of increasing complexity in students' thinking and understanding [27, 159]. Students proceed from factual, knowledge-based thinking to conceptual and abstract thinking

***M*acro structure**	Understanding the overall structure of the program text	Understanding the "algorithm" of the program	Understanding the goal/ the purpose of the program (in its context)
***R*elations**	References between blocks, e.g.: method calls, object creation, accessing data, ...	Sequence of method calls "object sequence diagrams"	Understanding how subgoals are related to goals, how function is achieved by subfunctions
***B*locks**	'Regions of Interests' (ROI) that syntactically or semantically build a unit	Operation of a block, a method, or a ROI (as sequence of statements)	Function of a block, maybe seen as subgoal
***A*toms**	Language elements	Operation of a statement	Function of a statement. Goal only understandable in context
	***T*ext surface**	***P*rogram execution (data flow and control flow)**	***F*unctions (as means or as purpose), goals of the program**
Duality	***Structure***		***Function***

Figure 2.3: The Block Model framework for program comprehension [270].

as they think across levels. This taxonomy can be used to understand the quality and levels of students' self-explanations as a means of classifying student-generated explanations in terms of their complexity. When students provide a prestructural response, it indicates either a significant misunderstanding of programming or the use of a preconception that is unrelated to programming [159]. Moving to a unistructural response, students demonstrate a partial understanding of the problem, grasping some aspects correctly but not all. In a multistructural response, students show an understanding of all parts of the problem but struggle to recognize the connections between them, failing to see the bigger picture. However, in a relational response, students integrate the various parts of the problem into a coherent structure and use it to effectively solve the task, demonstrating an ability to see the forest rather than just the trees [159].

Pennington's program comprehension model Pennington investigated the mental representations formed by programmers studying programs in the imperative style and developed a program comprehension model [226, 271]. The model consists of progressively more abstract layers: the surface form representation of the program text; the program model, which includes knowledge of operations and control flow; and the domain model, which includes knowledge of data flow and functions. These layers range from low to high levels of abstraction and vary in their explicitness and distribution within a program [271].

These frameworks around program comprehension have been used to guide the design of

program comprehension tasks (e.g., prompts to ask to explain a code piece to a certain level [152]), to evaluate the quality of student self-generated explanations [57, 201], and to design instructional explanations [143]. When designing the two prompted self-explanation types (Chapter 4), this work drew on the Block Model to guide students to explain code at a particular level.

2.2 Pedagogical approaches grounded in learning theory

2.2.1 The completion problem effect and Parsons puzzles

There are several instructional recommendations inspired by cognitive load theory. For example, worked examples are used commonly in programming learning to decrease the high cognitive load caused by conventional write-code problems. Worked examples demonstrate an expert's complete solution to a problem [3, 269]. They have been widely adopted, especially in early learning contexts [24]. However, one potential disadvantage is that worked examples might lead to passive rather than active learning [290]. Specifically, students do not always spontaneously explain the worked examples in a way that relates the actions in the worked examples to the principles and concepts [39, 249]. In response, scholars began exploring completion problems [310] as a way to engage students more actively with learning artifacts. Completion problems provide students with a partially written program that they must carefully understand and then finish [290, 309, 310].

Parsons programming puzzles (Figure 2.4, also known as Parsons puzzles, Parsons problems, or mixed-up code puzzles) are a type of completion problem [217]. A Parsons puzzle provides students with a set of mixed-up blocks and asks them to arrange these blocks into a correct solution to a given problem [73, 217]. Students receive immediate feedback for each attempt they make [217]. Puzzle types can vary based on how students need to order the code fragments, the block types, and practice adaptivity [73]. Parsons puzzles fall into two categories by dimension. In one-dimensional Parsons puzzles, code blocks only need to be placed in the correct order, but two-dimensional Parsons puzzles also require the blocks to be correctly indented [122]. Some Parsons puzzles also add distractor blocks [64, 74, 217]. Distractor blocks are lines of code that include syntactic or semantic flaws. There are two types of distractor blocks: paired distractors, which are displayed adjacent to the correct blocks, and unpaired distractors, which are randomly mixed in with the correct blocks. Prior studies have shown that visually paired distractors make Parsons puzzles less difficult to solve than unpaired distractors [64]. Prior work has explored various forms of feedback, such as block-based feedback that highlights blocks placed incorrectly or in the wrong order, and execution-based feedback that returns error messages from running the student's code [107, 338].

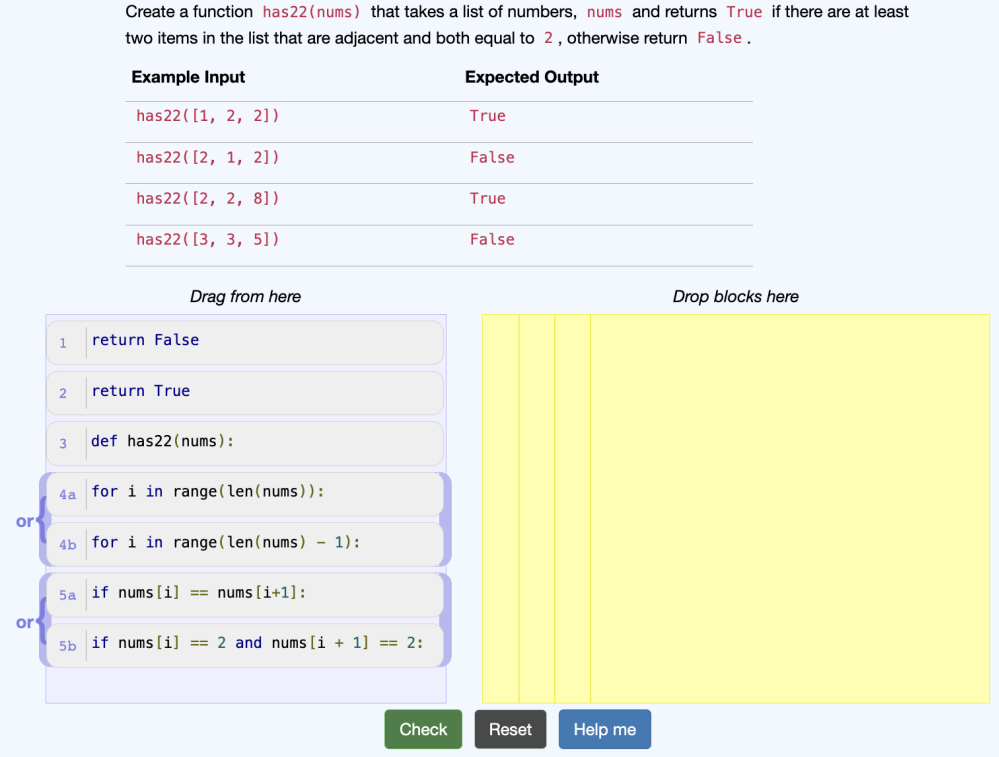


Figure 2.4: Sample of a two-dimensional adaptive Parsons programming puzzle.

Some Parsons puzzles are adaptive [72, 142]. Prior work introduced two types of adaptation for Parsons puzzles: intra-problem and inter-problem [72]. Intra-problem adaptation can reduce the difficulty of the current Parsons problem and inter-problem adaptation affects the difficulty of the next problem [74]. Parsons puzzles have been explored as both formative and summative assessments [68]. Studies have provided evidence that most students find Parsons puzzles to be engaging [68, 217], and students can achieve the same level of learning as writing the equivalent code, but with higher learning efficiency [74]. While Parsons puzzles offer structural support, they do not guarantee that students understand the solution [64].

Despite the well-documented exploration of Parsons programming puzzles in computing education, few efforts have investigated non-code variants. Prather et al. [235] explored how abstract Parsons problems can scaffold solution planning by having students engage with a problem prompt before coding, and found that novices who did so beforehand solved the task more successfully. Garcia et al. [90] and Cunningham et al. [51] likewise investigated abstract Parsons puzzles built around a predetermined solution. These implementations are largely instructor-defined and rarely reflect a student's own language or individual problem-solving approach. However, prior work has shown that students can approach the same problem differently [84] and express plans in varied ways [255].

2.2.2 The self-explanation effect and prompted self-explanation

Explaining is a crucial cognitive process that humans use to understand unfamiliar phenomena. They ask why something is the way it is to make sense of the world [168]. In the context of learning, this process is significant as it can lead to transferable and meaningful learning [185, 296]. Self-Explanation (SE) is a trainable learning strategy to encourage such active engagement, which involves learners reflecting on and generating explanations about the material. The ICAP framework has been used to provide justification for adding self-explanation prompts into diverse learning activities [79, 143, 316, 356]. According to the ICAP framework, SE is the activity of explaining an idea or concept aloud to oneself as one learns, or one can think of it as trying to make sense of the learning materials. SE can often be implemented in a constructive way that can generate new knowledge beyond what was initially encoded. This allows the knowledge of procedures to be applied to novel contexts or distant problems, and the knowledge of concepts enables the interpretation and explanation of new concepts [41].

In procedural domains, SE is traditionally implemented by having students explain each step in worked examples to themselves, justifying how a solution step follows from a prior step [39]. It can also be implemented by providing incomplete worked-out examples, even involving fading, thus requiring students to infer the omitted steps as a form of SE [41]. Previous research has proven self-explanation to be one of the most robust and effective learning strategies [28, 41] across different domains [248, 272, 336] and educational settings [145]. For example, early research found that students who conducted self-explanation during problem-solving achieved better test outcomes [204].

Ideally, students should initiate this process themselves, but many students are not able to conduct in-depth self-explanation on their own [250]. Even when students engage in self-explanation, incorrect explanations can occur [250, 251], which might cause misconception construction and severely hinder learning [49]. To address these barriers to effective self-explanation, different types of prompted self-explanations were created to facilitate this process [25, 28]. Given that learners often do not conduct self-explanation naturally, one way is to trigger their self-explanation through prompted questions.

Research suggested that prompted self-explanation activities should be designed so that they do not increase cognitive load beyond learners' working memory capacity [130]. Wylie and Chi (2014) presented a *prompted self-explanation format spectrum* [339] (Figure 2.5). In this spectrum, one end is open-ended prompted self-explanation questions and the other end is menu-based (choice-based) prompted self-explanation questions. This open-ended approach uses leading questions that prompt students to construct complete explanations on their own [156], but typically does not provide immediate feedback. The other end is menu-based (choice-based) self-



Figure 2.5: The continuum of different self-explanation prompt formats [339].

explanation questions. This format lets students choose from a list of predetermined explanation options rather than constructing things themselves [339]. Research has shown that using menu-based self-explanation questions to prompt students about key principles in worked examples [12] and problem-solving [5] can enhance learning outcomes. In the middle of the spectrum lies scaffolded self-explanation in which students are prompted to complete a given explanation by filling in missing text [339]. Activities with open-ended and focused SE prompts can be seen as constructive because the student must produce their own content [339]. Focused prompts are like open-ended prompts, but they give more specific instructions about what the self-explanation should include [339]. In addition to being applied as a constructive activity, SE can be used as an active activity in which students select an explanation or elaboration from a range of options, such as scaffolded, resource-based, and menu-based self-explanations [339]. According to cognitive load theory, some types of prompted SE questions might impede learning when they lead to cognitive overload [130]. For instance, less-scaffolded SE questions, such as those that are open-ended, could lead students to spend excessive time generating a self-explanation that is not correct or complete [152]. To address the problem of incomplete or incorrect self-explanation [49], scholars have explored providing instructional explanations as hints when needed [250] and offering immediate feedback when students self-explain incorrectly [5], helping them recognize their errors and correct their flawed mental models.

In the context of computing learning, SE questions have been used to engage students with instructional code examples [315] and videos [42]. SE questions have also been added to student submissions in program writing tasks [152] and debugging tasks [145]. One approach is to utilize prompts to ask students to justify their own program submissions. For instance, Lehtinen et al. [152] asked students to explain the structure and execution of their program after submitting their programs. Results showed that one-third of the students struggled to explain their code [152]. Researchers have also explored several ways to integrate SE prompts into problem-solving activities. For example, Fabic et al. [78] and Fabic et al. [79] asked students to fill in incomplete code lines within Parsons puzzle blocks and then answer a prompted SE question about their entries. In Fabic et al. [79], the prompts appeared partway through the puzzle-solving

process, though the study did not specify what students received after answering.

When students solve code-writing problems, simply passing all test cases does not guarantee a full understanding of key concepts, which can be evident from their inadequate explanations in their submissions [152]. This is because they often rely on trial and error to arrive at the correct answer [71]. Asking students to compare their solution with the instructor's solution is one low-effort way for the teacher to support continued learning after the student has solved the problem. Price et al. [242] took a different approach: students first completed a code-writing task, and then answered prompted SE questions about an instructor's solution [242]. In one variant, students selected from three multiple-choice options embedded within the explanation text to construct a complete explanation. In another variant, students were given open-ended questions that asked them to compare their code with the instructor's solution [242].

2.2.3 The use of conversation-based assessment

Conversation-based assessment (CBA) is a type of performance-based assessment that measures student knowledge, skills, and abilities in a highly interactive environment with natural language conversations [347]. Under both the original ICAP framework [41] and the revised version that incorporates feedback for online activities [229], conversation-based assessment can be positioned as a form of evaluation that engages students at the interactive level. Traditional written examinations and low-stakes digital assessments, by contrast, tend to place students in an active mode, or at best a constructive one, depending on the assessment type and whether feedback is provided [121, 229]. Conversation-based assessment goes further, as it requires students to respond, elaborate, and justify their understanding dynamically and in real time, pushing them to engage in multiple rounds of receiving and responding to feedback [229, 347]. Consistent with ZPD theory, by working with artificial agents, CBA has the potential to help students demonstrate levels of understanding and performance within their ZPD through appropriate scaffolding [125, 126].

CBAs implement principles of Evidence-Centered Design (ECD) to establish and maintain the linking of evidence to assessment constructs [193]. ECD is a principled methodology for assessment design as it requires an evidence-based chain of reasoning to support the validity of a given measurement [192]. The chain of evidence connects information provided by a student via responses to tasks aligned to specific constructs. By establishing a turn-taking dialogue between students and computer agents, students may be more engaged and provide additional evidence than in traditional assessment formats [161]. Two main aspects of facilitating dialogue in CBAs are (1) knowing how to respond based on student answers to obtain information about the assessed topic and (2) maintaining a natural and human-like conversation style. Traditionally, such dialogues were implemented through an expensive knowledge engineering approach [97]. This

approach drew on an established dialogue framework, expectation-misconception tailored dialogue [97, 98]. It also utilized NLP techniques to evaluate student responses, tag them with different speech acts, and provide targeted scaffolding or prompts (i.e., pumps, hints, follow-up questions) [97]. These speech acts reflect common response types that students provide in tutorial dialogue systems, such as AutoTutor [97]. In order to provide human-like conversations, student input was analyzed with latent semantic analysis [146] and regular expressions.

2.2.4 Practice planning explicitly in problem-solving

Planning has long been recognized as an important step in computational problem-solving. Thinking computationally often involves four key components [197, 333]: decomposition (analytical thinking), planning (algorithmic thinking), debugging, and generalization (abstraction). Polya defined four main steps when solving a problem, including *"understand the problem," "make a plan," "carry out our plan,"* and *"look back"* [231, p. 5]. Similarly, in problem-solving, scholars commonly identify several shared steps: understanding the problem, devising a plan, translating it into code, testing, and debugging [56, 77, 108, 187, 208, 254]. Treating planning as an explicit practice separate from the broader problem-solving process can reduce the cognitive load students need to manage at any one time, allowing them to focus on individual component skills critical to the whole [212, 275].

In earlier work, Pea and colleagues investigated the role of LOGO programming environments in developing planning skills through breaking problems into more manageable subproblems [220, 223]. However, planning has consistently posed challenges for novice programmers. Pea and colleagues [222, 223] found that students engage in very little preplanning during programming. This pattern was further confirmed by findings that novices, in contrast to experts, spend far less time planning and often jump directly into coding without a clearly proposed plan [260]. Nevertheless, scholars showed that novices who adopt effective initial strategies are capable of learning to program independently, highlighting the potential value of early planning support [261]. In light of this, prior work has explicitly advocated for teaching planning before programming, defining planning as *"formulating and defining the algorithm that will need to be implemented to meet the objective of the program"* [13].

However, prior work on supporting solution planning remains limited. Early efforts prompted learners to articulate their plans [101], but such an approach was open-ended and provided no immediate feedback. Later work incorporated subgoal labels into worked examples [199], though labels were instructor-generated rather than student-driven. Prather et al. [235] explored how abstract Parsons problems can scaffold solution planning by engaging students with a problem prompt before writing code, finding that novices who worked on an abstract Parsons problem

prior to coding were more successful in solving the task. Similarly, Garcia et al. [90] and Cunningham et al. [51] examined the use of abstract Parsons puzzles with fixed correct answers as a planning support mechanism. However, such approaches are typically predefined by instructors and seldom mirror the learner's own phrasing or solution strategy, limiting their ability to adapt to each learner's thinking. Prior work has shown that students may take different paths to the same problem [84] and express plans using distinct language [255]. Most recently, Rivera et al. [255] introduced independent planning practice with immediate personalized feedback, but without embedding it in the context of solving programming problems.

2.3 The growing use of LLMs for personalized education

Personalized learning is a process of adapting instructional materials and delivering learning content based on individual learner characteristics [225, 277, 352]. The procedure of personalization could target student characteristics at diverse granularities, such as interests, current knowledge level, thinking, strategies, and prior experience [23, 203, 307]. After understanding student characteristics, a system can then provide personalized learning through adaptive materials [44, 313]. From a cognitive load perspective, personalized learning plays a crucial role, as it helps to optimize the involved cognitive load to promote the desired learning outcome [23, 141, 157, 196]. When the instructional materials are adapted to students' characteristics, such as generating problems that align with their interests, cultural backgrounds [164], and skill levels [59], students become more familiar with the general context. This familiarity helps reduce extraneous cognitive load [358] and bridge the gap between their existing and new knowledge [55, 99]. Also, personalization based on students' current knowledge and skill level could potentially reduce the intrinsic cognitive load of the material difficulty [358].

2.3.1 Leveraging a single LLM to power personalization

With the development of generative AI, educational researchers are studying how these technologies can contribute to instructional content generation and personalized learning experiences [14, 103, 262]. LLMs have been shown to be capable of providing personalized artifacts such as feedback [340], error messages [318], code fixes, explanations [134], and pedagogical conversations [129] based on the student's current code and problem-solving status.

While the above work highlights LLMs' capability to benefit learning, concerns over the inappropriate application of such systems in education are growing [210, 283], especially given students' overreliance on LLMs' code-generation capabilities. After speaking with instructors of beginner-level programming courses, Lau and Guo [148] reported that a common concern was

that students who relied on AI code generation tools to get answers would not learn the material. As Kazemitabaar et al. [133] found, when students were given an AI code generator directly, more than half of the time they used it before trying to write any code, and 43% of the uses generated the entire solution. Furthermore, novice students who overly rely on AI code generators may experience fake training progress, and thus miss learning opportunities during practice, which makes it harder for them to apply fundamental concepts later [133]. To prevent improper usage of AI code generation tools, one recent study developed a system to generate textual responses to student requests that are similar to what a human tutor would deliver [154]. However, some students found it difficult to use because they had to type their input and they did not always know what to ask.

2.3.2 Coordinating multiple specialized LLM agents

There is increased interest in integrating LLMs to power agents that complete tasks and adopt defined roles [127, 128]. When properly prompted, these models can effectively simulate various educational roles [37, 170] with contextually appropriate conversational replies and feedback [284]. However, the use of a single LLM agent has drawbacks. In response, researchers have turned to multi-agent designs that distribute a task across several agents, each assigned distinct roles [162], strategies [34], and/or functionalities [132, 294]. An LLM-based multi-agent system often includes two main components: an orchestration platform that defines the interaction style, allocates duties, and manages the information flow; and multiple LLM-based agents with distinct skills and specialized functions (either predefined or adaptively produced) to handle more complex tasks together [106]. For instance, a multi-agent framework was proposed where LLM agents collaborate with diverse roles (e.g., skill identifier, learner profiler, path scheduler) within an intelligent tutoring system for the purpose of goal-oriented personalized learning [319]. In addition, a growing body of work has explored the use of multi-agent systems to generate personalized instructional materials. Specifically, Nguyen et al. [206] introduced a multi-agent synthesis technique employing expert and student agents to generate high-quality programming tasks. Researchers also proposed similar multi-agent frameworks with specialized agents to recommend personalized programming exercises [357].

Chapter 3

Using Adaptive Parsons Puzzles to Scaffold Code-Writing

This chapter presents the first investigation within the dissertation's first approach: developing theory-informed active-learning mechanisms. It focuses on code writing, an essential skill in early programming, and examines a mechanism for scaffolding novices as they work to master it. Portions of this chapter draw on material published at the 2022 ACM Conference on International Computing Education Research [112], the 2023 ACM Conference on Innovation and Technology in Computer Science Education [113], and the 2023 ACM Koli Calling International Conference on Computing Education Research [114]. The analyses here extend those prior publications.

3.1 Introduction

One of the primary goals for introductory programming classes is to develop skill in writing code [142, 327, 341]. However, a growing number of studies suggest that writing code can be prohibitively challenging [258] for novice programmers due to various reasons such as the gap between theoretical concepts and practice [273] and high cognitive load [311]. Students spend many frustrating hours trying to figure out why their program fails to compile or does not produce the expected output [22]. Such frustration is one reason for the high drop-out and failure rates in introductory programming courses [322]. To tackle this issue, prior work has investigated different scaffolding approaches to assist students while writing code. One-on-one tutoring has proven to be more effective than typical classroom instruction with one instructor for 30 students [29]. Since it is not feasible to scale one-on-one human tutors in undergraduate introductory programming courses due to the size of these courses and a shortage of course staff, computer-based scaffolding has been studied [236]. One notable approach is timely and personalized programming hints from intelligent tutoring systems (ITS) [10, 240]. Nevertheless, ITS have not been widely adopted in higher education due to the time and cost to develop such systems [7, 136]. To reduce this cost, ITS researchers have explored ways such as using data-driven technologies to generate hints automatically [243, 257]. Despite previous research demonstrating the possibility of employing these automated hints to aid write-code problems [33], there

are still several barriers to scaling the use of those techniques. First, the quality of automatically generated hints varies significantly, and low-quality hints might prevent students from seeking help [240]. Also, existing systems provide help based on the student's current code, which may not be sufficient for novices who have difficulty determining how to begin and require more comprehensive help [181, 258]. In addition, hints generated by one ITS are hard to implement or transfer to other systems [33]. As a result, there is still a huge need for a more scalable, low-cost, easy-to-implement, and high-quality solution that will allow students of various levels to succeed on write-code problems.

Parsons problems, an increasingly popular programming exercise that requires students to place mixed-up code blocks in the correct order [217], could potentially scaffold novices who are struggling while writing code from scratch. Although Parsons problems have been used as practice and summative assessment questions [68], to our knowledge, no prior studies have investigated the use of Parsons problems to scaffold write-code problems. Nevertheless, learners could benefit from Parsons problems in a variety of ways. For instance, Parsons problems can decrease the required cognitive load by constraining the problem space and providing pre-written code pieces, yet still exercise students' higher-level thinking skills [105]. Also, compared to worked examples that only provide passive guidelines, Parsons problems allow more interactivity and most students find them engaging [68, 217]. These all show that Parsons problems have the potential to provide effective scaffolding for write-code problems.

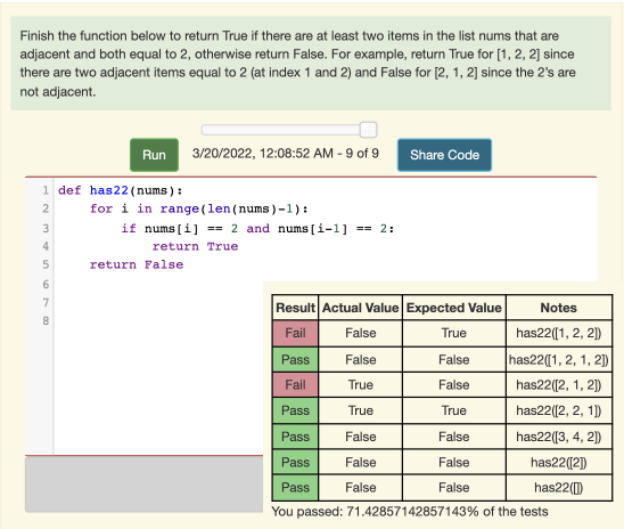


Figure 3.1: Screenshot of a write-code problem with unit test results.

Our research examined the impact of using Parsons problems to scaffold students while they are solving write-code problems. We conducted our studies in Runestone Academy, a free e-book platform that supports different types of code editing and execution in various programming

languages. In Runestone Academy, students can write and execute code and receive immediate feedback from unit test results and error messages (Figure 3.1). Our studies added an equivalent optional Parsons problem to each write-code problem. Students could display and solve the equivalent Parsons problem in a preview window (Figure 3.2), but were still required to solve the write-code problem. Students could not copy and paste the Parsons problem solution to the write-code problem; they had to at least retype the solution. Our research questions were as follows.

- **RQ1**: When, why, and how do students use Parsons problems to scaffold write-code problems? (Study 1)
- **RQ2**: What are the perceived benefits and challenges to using Parsons problems to scaffold write-code problems? (Study 1)
- **RQ3**: What is the effect of using Parsons problems as scaffolding to write-code problems on learning? (Study 2)
- **RQ4**: In the Parsons problems as Scaffolding (PS) condition, how did students rate the usefulness of the Parsons scaffolding and why? (Study 2)

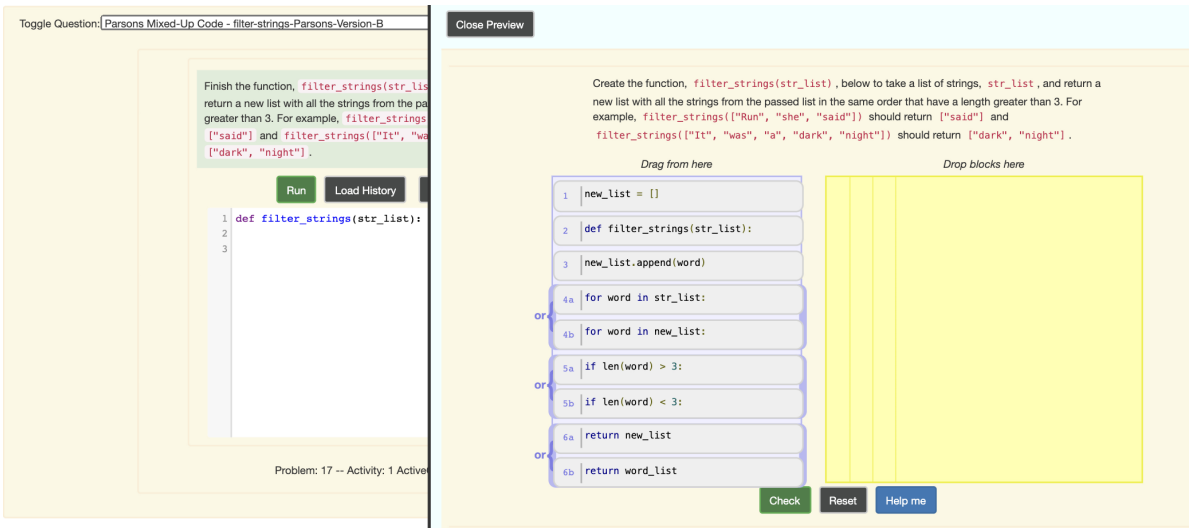


Figure 3.2: Screenshot of using a Parsons problem to scaffold a write-code problem.

3.2 Study 1: Within-subjects think-aloud study

The purpose of this study was to answer RQ1 and RQ2. We conducted a think-aloud study as it enabled information extraction regarding behavior and thoughts during problem completion.

We first conducted a pilot think-aloud study with 7 undergraduate students in summer 2021 on the same six write-code problems we used in Study 1. Students thought Parsons problems were beneficial as a type of scaffold for write-code problems in general. They also provided suggestions on how to improve our experimental materials. Specifically, some students reported that having separate variable names for Parsons problems and write-code problems was confusing, so we adjusted them to be the same. In addition, some students struggled to solve the Parsons problems since they were not adaptive, so we modified the problems to be adaptive (A1 and A2 in Figure 3.3). In fall 2021, we conducted a formal think-aloud study with 11 undergraduate students at a large public research university in the northern United States.

3.2.1 Method

3.2.1.1 Participants

We recruited participants from an introductory Python programming course in a public research university during fall 2021. Students who registered for this course received a recruitment announcement via Slack. Given that we were particularly interested in understanding the impact of using Parsons problems to scaffold novice programmers, we required that participants had less than six months of experience programming in Python. Also, following the IRB requirements, all of the participants had to be at least 18 to enroll in the study. Students who met the criteria and were willing to participate scheduled a one-hour appointment with the researcher. After finishing the study, each participant received a $25 Amazon gift card. Eleven students participated in the study. Ten (91%) of the 11 participants were aged between 18 and 20, and one (9%) was 28. Eight (73%) of the 11 participants were female, and three (27%) were male. All participants had less than six months of Python coding experience, and for 10 (91%) of 11 participants, this introductory Python course was the most advanced programming course they had taken. Only one participant had taken a high-school CS course in C over two years ago.

3.2.1.2 Procedure

Each think-aloud study was conducted virtually through Zoom and lasted 60 minutes. The study started after we checked participants' age and obtained verbal consent to record. During the session, participants started with a sample problem with verbal instructions to make them comfortable with our online environment (Figure 3.2). The instruction covered all of the features needed in later practice and explained the relationship between the write-code problem and Parsons problem. Participants could play around with this sample problem and ask questions. Next, we provided a brief think-aloud training with an example to help participants understand what

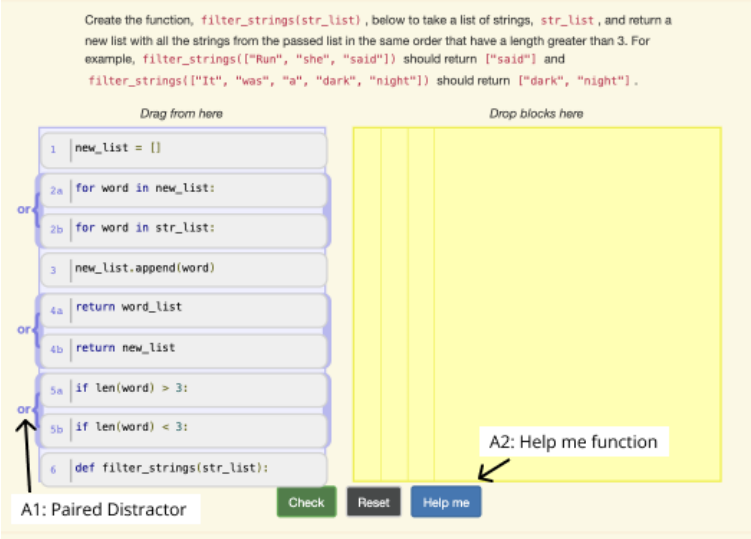


Figure 3.3: Screenshot of the Adaptive Parsons problem interface.

they needed to do during the practice. After finishing the on-boarding tasks, we muted our microphones and let the participants finish the six write-code problems. Each participant had 45 minutes to complete the six problems, and they were allowed to skip one or stop at any time. Following the whole think-aloud session, we conducted a 5-10 minute semi-structured interview to ask about their experience with using Parsons problems to scaffold write-code problems (e.g., "Tell me about a time when you tried to use a Parsons problem to help you solve a write-code problem and found it helpful/not helpful, and why?") and suggestions for future improvement (e.g., "What suggestions do you have for improving the interface?"). At the end of the session, participants filled out a demographic survey about their gender, age, and ethnicity.

3.2.1.3 Materials

This think-aloud study used a web-based environment (Runestone Academy). Since one of the goals of the study was to gather students' opinions on Parsons problems as scaffolding, each participant received six write-code problems with an equivalent two-dimensional adaptive Parsons problem as optional scaffolding.

Write-code Problems We sourced the write-code problems from an intermediate Python programming course at the same public research university. We categorized the problems into three difficulty levels, and chose two problems from each level (easy, medium, and hard). Our purpose was to make sure that participants with different skill levels all had a chance to use a Parsons problem to scaffold a write-code problem.

Parsons problems as scaffolding One recent study found that a Parsons problem with an unusual solution increased students' cognitive load [105]. To address this problem, we generated

Table 3.1: Distractor Locations by Version

Version	Distractor Location
A	Distractors - No Distractors - Distractors - No Distractors - Distractors - No Distractors
B	No Distractors - Distractors - No Distractors - Distractors - No Distractors - Distractors

students' most common solutions by clustering student-written code from previous semesters using Abstract Syntax Tree software, and then used the most common student solution to create the equivalent Parsons problem (Figure 3.3). We also wondered whether it would be better to use distractors or not in the equivalent Parsons problems. Therefore, we inserted paired distractors into three of the six Parsons problems (A1 in Figure 3.3) and placed them interleaved across problems. The distractor blocks were created by experts based on common syntax or logic misunderstandings [217]. Two versions (A and B) were created where the only difference was that if A contained distractors, B did not, and vice versa (Table 3.1). Participants were randomly assigned to either version. Additionally, a "Help Me" button was provided at the bottom of each Parsons problem to assist students who struggled while solving the Parsons problem (A2 in Figure 3.3). This button triggered the intra-problem adaptation which either removed a distractor or combined two blocks into one if the learner had made at least three attempts to solve the problem and had more than three blocks left in the solution.

3.2.2 Data Analysis

We recorded the screen using Zoom for all of the think-aloud sessions and transcribed the recordings. We analyzed our data using *ATLAS.ti*. To answer our research questions, we devised a coding scheme that focused on when, why, and how students used Parsons problems as well as the perceived benefits and challenges. To create the coding scheme for the "when" part, we applied a deductive approach [191] and identified four representative stages of programming development from existing literature: problem understanding, solution planning, solution implementation, and implemented solution evaluation [58, 166]. For other sub-dimensions, we used an inductive method to generate the initial coding scheme. Then two researchers independently coded two transcripts, met to address any disagreements, and refined the coding scheme iteratively. After finalizing the coding scheme, two researchers independently coded four transcripts (36% of the data), and reached an intercoder reliability score of 0.84 (alpha values > 0.80 are considered reliable [139, 323]) using Krippendorff's alpha [140]; then one researcher coded the rest of the transcripts. Relevant codes were grouped into themes to describe our results in relation to the research questions. When reporting our findings, we utilized (explanation) to clarify missing information in student quotes and [behavior] to indicate student behaviors.

3.2.3 Findings

3.2.3.1 RQ1: When and why do students use Parsons problems to scaffold write-code problems?

In this section, we present our findings on when and why students used Parsons problems to scaffold write-code problems. We found that they opened the Parsons problem at three different stages: planning a solution, implementing a solution, and debugging a solution.

All eleven students (100%) in our study solved at least one Parsons problem. Some completed the Parsons problem based on the immediate line-based feedback [105], while others spent more effort. Some students relied on the intra-problem adaptation ("Help Me") to solve the problem, P3 described this process as *"Then at some point, I was like, I don't know how to solve this. So I clicked the Help Me button, and then I got rid of one of them (Parsons blocks)."* In addition, we noticed some gaming behaviors [16] as they experimented with different block combinations to find the correct solution. P10 described how she got the right solution, "[reorganized some blocks] ... *so I guess I'll check this even though I don't think it's right* [clicked the check button and the solution was correct] ... *Oh, okay."*

Planning a solution: After reading the problem description, four students (36%) opened the Parsons problem because they were unsure about how to start. For example, P3 stated *"I don't know how to start here, so Parsons thing ...* [opened the Parsons problem]." And P8 was confounded when trying to tackle specific requirements in the problem description, *"I don't know how to do that adjacent part ...* [opened the Parsons problem]." Students were motivated to use the Parsons problem to help them plan their initial solution. Given that Parsons problems are constructed from a correct answer, they can be a valuable resource to start students on the correct path. Like P8 mentioned, she opened a Parsons problem once to *"check how many lines to expect ... to get an idea of how long it should be."*

Implementing a solution: Seven students (64%) chose to open the Parsons problem while they were implementing a solution. In this case, students often had at least a partial solution in mind, and had already written some code. Five (71%) of the 7 students opened the Parsons problem at this stage because they realized they had taken a wrong path and could not figure out how to correct it. For instance, P8 wrote some code and said *"If this is an even number* [typed some code and then deleted it] ... *well let me just go look at that (Parsons problem). This is ridiculous."* Three (43%) of the 7 students had difficulty translating the solution into source code using their existing Python knowledge [166]. For example, after writing 8 lines of code, and changing line 9 twice, P4 said *"Ok ... I am not sure how to skip this next number so I am gonna look at the Parsons."* Similarly, P10 explained her solution correctly verbally before writing code, but paused while writing and said *"I don't know exactly how to do that. Like, I don't think you can do ... so I'm just*

gonna switch to Parsons just to make sure."

Debugging a solution: After writing a complete solution, ten students (91%) referred to the Parsons problem to correct their unit test errors for at least one problem. All ten students switched to the Parsons problem when they failed to resolve their errors. Some students already knew where their errors were before opening the Parsons problem, but could not solve them. For example, P6 changed a number first in the code while saying *"Maybe minus 1 ... let's try that ...* [checked the code but still failed unit tests] *... no"*, so she opened the Parsons problem to help her debug. Similarly, P4 diagnosed the errors correctly, but she said *"I forgot how to do the index function so I'm gonna check the Parsons."* On the other hand, some students looked at Parsons problems when they had unit test errors but did not know why the code was wrong. For instance, P11 said *"I don't know why it's not working"* when only passing 50% of the unit tests, and opened up the Parsons problem.

3.2.3.2 RQ1: How students used Parsons problems to scaffold write-code problems

Recall that even if students solved the Parsons problem they still had to solve the write-code problem. They could not just copy and paste the Parsons solution, they had to at least retype it. In this section, we will focus on how students used the Parsons problem to help them finish the write-code problem. We wondered if they would just scan the Parsons problem and not try to solve it, would they partially solve the problem until they figured out what they needed and then finish writing the code, would they solve the Parsons problem and just retype the Parsons solution (replacing their original code), or would they solve the Parsons problem and use that solution to modify their code.

Scan Parsons Problem: Because Parsons problems already contain correct code fragments, four students (36%) were able to gain information by scanning the mixed-up blocks without moving any of them. For example, P7 opened up the Parsons problem after diagnosing his syntax error, and said *"Oh I don't remember the method name so check the Parsons code* [opened Parsons problem] *... so let's see ...* [found that method in one block] *... so let's go back ...* [changed his code, ran and passed all unit tests]." In another example, P3 also opened a Parsons problem after getting unit test errors and caught her syntax mistakes when comparing the mixed-up blocks with her own code, *"Oh my god, I was close* [scanned through all the blocks, and pointed to one block] *... oh the word (the variable name) ...* [closed Parsons window, modified append to append(word), ran and passed all the unit tests] *... Oh my god, I did it."* In this situation, students were able to fix their errors by looking over the mixed-up blocks, thus they didn't need to attempt to solve the Parsons problem.

Attempt Parsons Problem: Two students (18%) were able to debug their errors by only moving a few blocks rather than solving the entire Parsons problem. In particular, P6 opened

the Parsons problem twice when planning a solution, and after dragging several blocks, she got a basic idea of how to achieve the solution and started to type in the write-code problem. Similarly, P4 opened the Parsons problem to debug, and after dragging two blocks, she saw one block and said *"I used the wrong function, let's use module ...* [modified code in write-code problem, checked and passed all the unit tests]."

Solve and replace their code: Six students (55%) deleted their own code in the write-code problem and retyped the Parsons solution in several problems. Four students chose to keep the Parsons problem window open while retyping the code to match the Parsons solution. During this process, students might self-explain the Parsons problem solution. For example, after getting a Parsons solution, P8 typed it in the write-code problem and said, *"let me copy it ... What does the minus one do (-1 in range(len(var)-1))?"* To answer her own question, she did some tests in her code such as removing the number or changing to another one. After receiving an error that said "list index out of range", she understood the usage and said "Oh okay, so maybe that's why it was here." Two students wrote the Parsons solution in the write-code problem after closing the Parsons problem window. Before exiting the Parsons problem, they both correctly explained to themselves why the mixed-up blocks were placed together.

Solve and modify their code: While some students replaced their code with the Parsons solution, we found evidence that other students chose to apply what they had learned to refine their existing code. Students tended to use this approach when their primary goal was to debug their solution. Specifically, five students (45%) first examined how the Parsons solution achieved the goal, then went back to their code to refine it based on what they had learned from the Parsons solution. For example, when reflecting on how they used the Parsons problem solution in write-code problems, P5 said *"I had a general idea of how to solve it but I was having an error message, after using Parsons problem I was able to figure out where I messed up and I was able to fix my own code."* In addition, to build up a stronger connection between the solution and the written draft, it was common for students to compare their code with the Parsons solution side by side. For instance, after understanding how the Parsons solution worked, P1 compared it with his own code and explained *"for each number if the number (explaining the Parsons solution) ... oh the one after equals to two (realizing his error) ... so I'll go back to this* [backed to change code in the write-code problem]."

3.2.3.3 RQ2: The benefits of using Parsons problems to scaffold write-code problems

To determine student perceptions of the benefits and challenges of using Parsons problems to scaffold write-code problems, we conducted semi-structured interviews with the 11 undergraduate students at the end of the think-aloud sessions. First we will present the perceived benefits, and we will report the challenges in the next subsection.

Reduce difficulty and completion time: Six students (55%) stated that the Parsons problem aided them in lowering the difficulty of the problem and speeding up the problem-solving process. For example, P5 told us that *"It kind of, because it doesn't just give you the answer right off the bat. They gave me like a good start to it. And then from there, I was able to figure (it) out easily."* Parsons problems also reduced debugging time. For example, P11 stated *"I just used the Parsons problems to check a small mistake I did … And I found that instead of divide, I should do (another method)."* Another student said Parsons problems were preferable to searching on Google as *"there's more than one way to solve some coding problems (so it takes time to find), and it's a little frustrating especially when your teachers want one method."*

Learn problem-solving strategies: Most students (91%) found Parsons problems helpful to gain a clearer understanding of how to create a correct solution. Some students obtained new strategies, for example, P4 explained that *"It got me to think how the these steps are like this, like the reasons behind it and why I should do it like this, it made me think about the process of the steps that I have to take to solve them."* For those who already had a plan for how to solve the problem, working on Parsons problems helped them to develop more effective strategies. Like P2 described, *"When I was using a for loop, and then I saw that the Parsons problem used a while loop. It helps me realize that would be a better solution to that problem."*

Refine and extend existing programming knowledge: A Parsons solution contains information about programming language syntax and semantics. Most students (91%) mentioned that they clarified misunderstandings by using Parsons problems. As P3 said, *"But once I like, looked up the Parsons thing, I knew right away, first of all, I got this number wrong. Like I understood why because it was a simple mistake."* In addition, Parsons solutions serve as good examples to teach new programming language syntax. For instance, P7 explained how the Parsons problem taught him syntax knowledge, *"I don't remember the Python syntax. So I use a Parsons problem to view its syntax, then (I) get a better idea of how the code should be written."*

Prompt to think more deeply: When asked about the helpfulness of Parsons problems as a scaffolding tool, P4 compared it with providing an answer directly, and claimed that *"I really enjoyed it (Parsons problem) because the process made me think, it is going to be useful when I solve a different problem that is similar, rather than just like giving the answers, it really makes me think."* More students (45%) shared something similar when asked about the use of distractor blocks in Parsons problems. Despite the fact that distractors made the problem more challenging, they also pushed students to think and learn more. As P8 stated, *"I feel like I don't want everything to be easy for me because I want to learn and so I think this one (the one with distractors) was good."* Students also gained a deeper understanding by distinguishing between the distractors and correct blocks, like P11 said, *"… it (the one with distractors) did help because it showed how a small difference will not work. And I feel like for other ones (the one without distractors) … then you don't think about*

like if you did it a different way it wouldn't work."

3.2.3.4 RQ2: The challenges of using Parsons problems to scaffold write-code problems

While there is evidence that Parsons problems provided a wide range of help to scaffold students' code writing, some participants encountered challenges during this process.

Difficulty solving the Parsons problem: Given that we used the most common solution to create the Parsons problem, it was possible that students who had a different approach in mind would experience a higher cognitive load. For example, P9 skipped one question and stated, *"I was thinking differently on how to approach it, so it's just the Parsons problem (solution) didn't seem reasonable to me."* Three students (27%) also complained about the use of distractors as they made it easier to get stuck, as P3 explained, *"I prefer the ones without the distractors. Because a couple of times I used the wrong choice, so that wasn't very helpful. And that kind of led me astray a little bit. So it'd be easier just to have one option that is necessary in the solution."*

Help Avoidance: This refers to situations in which students did not seek help when it was most likely to be useful. We encountered two types of help avoidance: avoiding use of the Parsons problem and avoiding use of the intra-problem adaptation (triggered by clicking the "Help Me" button). Only one student (P7) clearly expressed his avoidance of Parsons problems as *"(it) isn't really a hint, it's like the answer."* He would prefer more high-level guidance on the logic instead of a Parsons problem. Two students (18%) avoided using the "Help Me" button due to a desire for independence [241]. As P10 explained, *"I feel like I'd rather try to figure it out myself first. So I didn't really want the answer to be kind of given to me more easily."*

Difficulty understanding the Parsons solution or adaptation process: As previously mentioned, some students gamed (intentionally or unintentionally) the system to achieve the correct solution but did not gain a proper understanding of the solution, and thus felt that they had gained very little from solving the Parsons problem. For instance, when looking at the Parsons solution she created, P8 said, *"What am I doing? I mean, I kind of understood it but not really. Because I don't know what the slash slash (floor division) means."* Sometimes the intra-problem adaptation confused the students as students did not understand the intention or what to do after the adaptation. As P3 explained, *"All it did was eliminate an option, it just felt like a guessing game rather than the actually learning … And that didn't like help me quite understand. I was still stuck in the same problem that I was beforehand."* P6 shared the same feeling, and said, *"I tried to use the 'Help Me' button, but it just kind of confused me more … it took it (a block) away from my code, but it still said I needed more like lines of code."*

3.2.4 Discussion and Design Suggestions

In this study, we investigated the use of Parsons problems to scaffold write-code problems. Our think-aloud study (Study 1) demonstrated that Parsons problems helped students solve the write-code problems, learn new problem-solving strategies, and refine their programming knowledge.

3.2.4.1 Help-seeking behaviors in writing code

We answered the "when" and "why" parts of RQ1 by revealing three representative scenarios where students used Parsons problems to scaffold their code writing in Study 1. In general, those scenarios are consistent with the help-seeking strategies identified in the literature in non-programming domains [6], such as using help when students do not know how to begin and when debugging. Our findings also surfaced specific help-seeking behaviors in write-code problems. For example, our study revealed an important scenario where students asked for help when they had a solution in mind but had difficulty translating it into code [166]. In this case, Parsons problems helped students translate the algorithm from plain English to code.

When it comes to the "how" part of RQ1, we found that students interacted with the Parsons problems in four different ways. First, in "Scan Parsons Problem" and "Attempt Parsons Problem," students referred to Parsons problems to correct syntax errors, consistent with the Glossary mechanism in ITS [6]. This helps students quickly get information about syntax and apply it to the write-code problem. Second, in "Solve and replace their code," Parsons problems were used as a differentiated instruction tool [91] so that beginners with less prior knowledge were not overwhelmed by the open-ended code-writing problem. In addition, in "Solve and modify their code," students finished solving the Parsons problem first, and used it as a worked example to facilitate further understanding [279].

However, we also observed some help avoidance as a small group of students avoided utilizing the "Help Me" function or even the Parsons problem when they were stuck, which has previously been seen in other programming scaffolding research [181, 355]. To address this problem, we plan to use prompts to encourage students to seek help once the system detects many failed attempts.

3.2.4.2 Opportunities for personalized Parsons problems to scaffold writing code

Personalize the Parsons problem based on the student's solution Our studies suggested several benefits of using Parsons problems as scaffolding for students' performance and learning (RQ2). One significant difference between Parsons problems and writing code from scratch is that Parsons problems constrain the problem space [308]. Parsons problems break down a complex write-code problem into more manageable tasks [75], which is one possible explanation for why students believe that they reduce the difficulty of the problem and speed up their problem-solving

progress. Our studies also showed that students may learn alternative problem-solving strategies from Parsons problems. In our study, we utilized the most common solution from student-written code to generate the Parsons problems, which could differ from the solution path the current student is following. Under this circumstance, some students are able to distinguish and examine both approaches and learn alternative problem-solving strategies. This is consistent with a previous study [218] showing that having students view distinct code in parallel helped them achieve better learning of procedural knowledge and develop higher flexibility in problem solving. Nevertheless, this may create a higher cognitive load for other students. For example, some students failed to solve the Parsons problem, or did not find it helpful, as the Parsons problem was using a different strategy than their approach.

This suggests that personalized Parsons problems (e.g., ones that are closest to the student's current problem-solving strategy) or personalized feedback (e.g., explanations on the connections between the Parsons blocks and the student code) are needed to better support students.

Provide additional support on Parsons solution comprehension We found specific scenarios where students used the Parsons problem to refine or extend their knowledge. For example, some students self-explained while solving the Parsons problem, or reflected on the difference between the Parsons problem and their own incorrect solution. These students are more likely to identify inadequacies in their prior knowledge, correct misinterpretation, and demonstrate conceptual change and learning from Parsons problems [166]. Nevertheless, other students found it challenging to understand the Parsons solutions due to a lack of prior knowledge [39]. This result implies that we could provide further support for beginners to comprehend the Parsons problem solution, and reflect on the solution [250]. Our participants provided similar recommendations, as P8 suggested, *"I wish there was like an explanation about what's like [pointed to one block in Parsons solution]."* In addition, recall that the intra-problem adaptation confused some students as it removed or combined blocks without any explanation. Therefore, adding more explanation could increase the effectiveness of the current adaptation process.

Dynamically adjust the difficulty of the Parsons problem Our findings also revealed that some students valued the opportunity to think more deeply by solving the Parsons problems, which is consistent with previous results on the desirable difficulty introduced by adaptive Parsons problems [74]. Instead of receiving a passive "bottom-out" hint, solving Parsons problems and recognizing common mistakes from paired distractors, could improve students' ability to identify errors, which could lead to more robust learning. On the other hand, some students became more perplexed specifically because of the distractors. In light of the aforementioned challenges, we need to increase the adaptivity of Parsons problems to provide more personalized and targeted scaffolding when novices are writing code. In other words, Parsons problems can be constructed dynamically depending on the student's problem-solving status when seeking as-

sistance. Specifically, for those who already have an idea of what to do, skeleton-style Parsons problems can be provided such as those that only contain subgoal labels [75]. In this way, students could compare their existing thinking processes to the program skeleton in the Parsons problem. For students with less experience, we could remove distractors or combine some blocks in advance to provide a simpler Parsons problem.

3.2.5 Limitations and Future work

Study 1 was done with a relatively small sample of students, which made it difficult to cover all the possible scaffolding scenarios. Also, because the majority of students in Study 1 were new to the think-aloud observations, even though we gave them a thorough introduction with examples beforehand, it was possible that they missed some critical parts throughout thought sequences, causing bias in our results. Another limitation is the learning materials we employed. The write-code practice problems were concentrated only on one programming topic in one programming language (Python). Future work needs to explore diverse demographic groups and additional topics and programming languages.

Our goal is to employ Parsons problems as a scalable and effective scaffolding tool for write-code problems for students with different levels of ability and/or knowledge. We are excited to pursue future work based on the design suggestions mentioned above. First, more research is needed to fully understand the effect of using Parsons problems as scaffolding for write-code problems for diverse student groups, contexts, topics, and programming languages. Second, we observe opportunities to provide personalized scaffolding using Parsons problems through iterative design. Third, we plan to collect log data which records how students write code and seek help on the platform to better understand their behaviors.

3.3 Study 2: Between-subjects classroom evaluation study

One IRB-approved study was conducted in the fall semester of 2022 at a large public research university in the northern United States. All participants were enrolled in a data-oriented programming course, which was the second required Python course for the university's information science majors, though other majors take the course as well. This course covered programming concepts including Python basic data structures (list, tuple, and dict), object-oriented programming concepts (`classes, objects, and inheritance`), how to debug, how to use unit testing, basic web scraping, regular expressions, HTML, XML, JSON, working with APIs, working with databases, and Matplotlib.

Table 3.2: Four Write-code Practice Problems

Problem Name	Question Item
1 - *Song*	Write a class `Song` with an `__init__` method that takes a `title` as a string and `len` as a number and initializes these attributes in the current object. Then define the `__str__` method to return the `title, len`. For example, `print(s)` when `Song('Respect', 150)` would print "Respect, 150".
2 - *Cat*	Write a class `Cat` with an `__init__` method that takes `name` as a string and `age` as a number and initializes these attributes in the current object. Next create the `__str__` method that returns "name: name, age: age". For example if `c = Cat('Fluffy', 3)` then `print(c)` should print "name: Fluffy, age: 3". Then define the `make_sound` method to return "Meow".
3 - *Account*	Create a class `Account` with an `__init__` method that takes `id` and `balance` as numbers. Then create a `__str__` method that returns "id, balance". Next create a `deposit` method takes `amount` as a number and adds that to the balance. For example, if `a = Account(32, 100)` and `a.deposit(50)` is executed, `print(a)` should print "32, 150".
4 - *FortuneTeller*	Write a class `FortuneTeller` with an `__init__` method that takes a list of fortunes as strings and saves that as an attribute. Then create a `tell_fortune` method that returns one of the fortunes in the list at random.

3.3.1 System Interface

Runestone allows students to write, execute code, and receive immediate feedback from unit test results (Figure 3.1) [76]. This study added an equivalent optional Parsons problem to each write-code problem to scaffold students' code-writing practice. When students had difficulty solving a write-code problem independently, they could display, interact with, and work on the equivalent Parsons problem in a preview window (Figure 3.2), but they were still required to solve the write-code problem to earn points. Students could not copy and paste the Parsons solution to the write-code problem; they had to retype it. When students failed to pass all the unit tests after three submission attempts, they would receive a pop-up prompt reminding them that help is available ("Help is Available Using the Toggle Question Selector").

3.3.2 Participants and Procedure

The classroom study was conducted during the 80-minute lecture period in week three of the class; students who did not attend the lecture were allowed to finish the study by the end of the day. Students were randomly assigned to one of two conditions: Parsons-Scaffolding condition (PS) and No-Parsons-Scaffolding condition (NP). Students in the PS condition received a text-entry write-code interface with an equivalent two-dimensional adaptive Parsons problem as scaffolding (Figure 3.2), while the NP group only had the write-code interface (Figure 3.1).

Table 3.3: Questions about student general CS self-efficacy level

Question Item
1 - Generally I have felt secure about attempting computer programming problems.
2 - I am sure I could do advanced work in computer science.
3 - I am sure that I can learn programming.
4 - I think I could handle more difficult programming problems.
5 - I can get good grades in computer science.
6 - I have a lot of self-confidence when it comes to programming.

Students first worked with a knowledge tutorial on Python `classes`. It covered three fundamental concepts: defining a new `class`, constructing new `objects`, and writing new `methods`. Each concept included textual instruction and an executable worked example, followed by a task where students were instructed to modify the code (Figure 3.4). After finishing working with the knowledge tutorial, students received an introduction to the types of problems in the system: scaffolded code writing or non-scaffolded. Following that, students were asked to complete a programming self-efficacy survey (6 questions) and a self-evaluation on their knowledge about writing code for `classes` (4 questions). Then, students were given four write-code practice problems in each of the two conditions, with the only difference being that students in the PS condition had the equivalent Parsons problems as scaffolding. Students in the PS condition were explicitly told that they had to enter code in the write-code area to earn points. We generated a random number between 1 and 10 once the student clicked to start the practice. Based on this number, we assigned them to the NP condition if it was odd and the PS condition if it was even. After each practice question, students in the PS condition were given a question, asking them to rate the perceived usefulness of using a Parsons problem to help them solve the write-code problem on a Likert scale from 1-low to 9-high. Students who did not use the Parsons scaffolding were asked to skip the corresponding survey question.

Additionally, following the last practice question, PS students were asked to respond to a brief open-ended question explaining their general perceived usefulness of a Parsons problem as scaffolding while writing code. Students in both conditions then finished the posttest. Each section had no time restrictions, allowing students to progress through the materials at their own pace until the end of the day. A total of 134 students participated in this study. The final sample contained 89 students who completed all of the materials in order (41 in PS and 48 in NP). Only three students chose to finish the materials outside of class time. They were not outliers after we checked their main measures, including practice performance, problem-solving efficiency, and posttest scores.

Table 3.4: Questions and ratings on prior-practice knowledge levels in Python `class`

Question Item	Rate Level
1 - Creating classes like `class Person:` and `objects` like `p = Person("XXX")`	1 - I am unfamiliar with these concepts
2 - Methods like `__init__` and `__str__` not used them in a program	2 - I know what they mean, but have
3 - The use of `self` in `class`	3 - I have used these concepts in a program, but am not confident about my ability to use them
4 - Defining instance variables like `self.color = color`	4 - I am confident in my ability to use these concepts in simple programs
	5 - I am confident in my ability to use these concepts in complex programs

3.3.3 Materials

There were five parts to the materials: basic knowledge instruction on how to create `classes`, `methods`, `objects` in Python; a survey to measure students' CS self-efficacy level and pre-practice knowledge level about writing `class`; four write-code practice problems; the corresponding equivalent Parsons problems; and an immediate posttest to assess their learning performance.

3.3.3.1 Basic knowledge tutorial

The knowledge introduction described how to create a new `class`, how to override the inherited `__init__` and `__str__` methods, how to define a new `method`, and how to create `objects`. For each subtopic, we provided a combination of textual introduction and an interactive worked example that the participant could execute and revise to learn how to handle a specific problem in Python [11]. Students in this study had little prior experience with creating a new `class` in Python, so we believed this basic knowledge tutorial part would provide students with some conceptual and procedural knowledge on this topic (Figure 3.4).

3.3.3.2 Survey Items

To gather students' self-efficacy in computer science learning in general (their general CS self-efficacy level), we used the self-efficacy items developed for the computer science domain [330]. The scale to measure self-efficacy was the first 6 items from a 13-item subscale [328]. Participants were asked to rate these statements on a five-point scale, ranging from "1-strongly disagree" to "5-strongly agree" (Table 3.3). This resulted in a score scale of 6 to 30.

Additionally, to assess whether there was a significant difference between the two groups, we

used a method to collect self-ratings for the pre-practice knowledge level on this topic [69]. Self-evaluation instruments can be used to capture pedagogically useful information about students' prior programming knowledge [69]. To measure students' pre-practice knowledge level, we created a four-question survey around `class` and asked students to rate how well they understood the concepts and could proceed with translating them into actual code (Table 3.4). For the rate levels, given that this instrument was used to measure a new topic, we created five different levels adapted from the original prototype self-evaluation instrument (Table 3.4), and assigned a score of 1-low to 5-high for each level, with a total score range of 4 to 20.

3.3.3.3 Write-code Practice Problems

We sourced the write-code problems from an intermediate Python programming course at a public research university in the United States. We chose four write-code practice problems, each worth 10 points (Table 3.2). The first problem required students to create a class with an `__init__` and `__str__` method. The second and third problems involved implementing a third method in addition to `__init__` and `__str__`. The fourth problem was the hardest one which required performing a random selection from a list, in addition to creating a class and a method. Our goal was to make sure that participants with different skill levels all had a chance to use a Parsons problem to scaffold a write-code problem. During practice, students from the two conditions both received execution-based feedback after each run, which showed compiler errors and the output from running the code including the results from unit tests (Figure 3.1). Each problem was scored out of 10 points based on the percentage of unit tests that passed. There were a total of 40 possible points for the overall practice. In this analysis, we first extracted the practice results provided by the auto-grading system. Then, we manually checked the final code submissions and accounted for any auto-grading errors before conducting the analyses.

3.3.3.4 Parsons Problems as Scaffolding

One recent study of Parsons problems found that a Parsons problem with an unusual solution increased students' cognitive load [105]. To address this problem, we clustered student-written code from previous semesters using the OverCode software [92], and then used the most common student solution cluster to create an equivalent Parsons problem (Figure 3.2). To highlight common misconceptions, we also inserted paired distractors into the Parsons problems. Experts created the distractor blocks based on common syntax or semantic errors [217]. Additionally, a "Help Me" button was provided at the bottom of each Parsons problem to assist students who struggled while solving the Parsons problem. This button triggered intra-problem adaptation, which either removed a distractor or combined two blocks into one if the learner had made at

least three attempts to solve the Parsons problem and there were more than three blocks left in the solution.

3.3.3.5 Posttest Items

Two types of questions were included in the posttest: write-code problems (10 points each, 20 points in total) and fix-code problems (10 points each, 20 points in total). In a write-code problem, the student needs to write the correct code from scratch following the problem description, and in a fix-code problem, the student must fix the errors in the existing buggy solution. These questions covered similar concepts to the write-code practice questions. Every time they ran the code, students would receive execution-based feedback, which showed compiler errors and the output from running the code, usually including results from unit tests. We calculated their posttest scores by using the proportion of passed unit tests in the final submission.

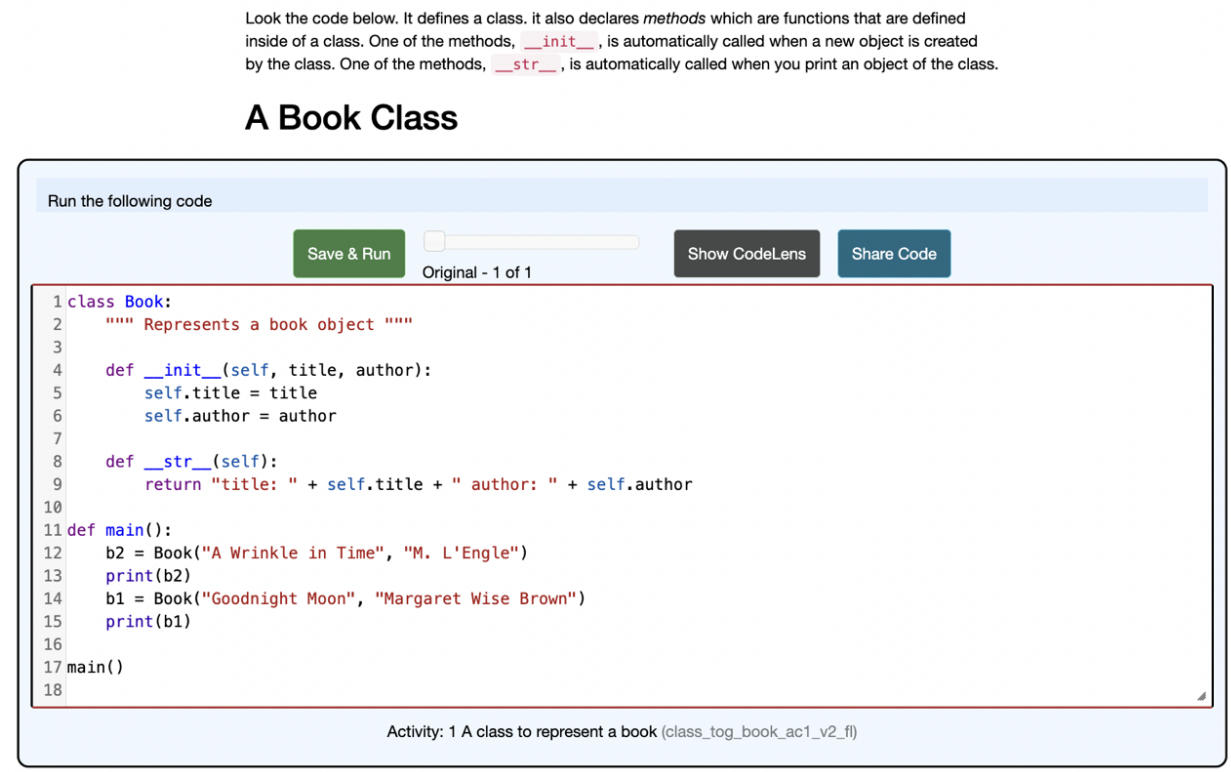


Figure 3.4: Screenshot of the first part of the basic knowledge instruction.

Table 3.5: Pre-practice knowledge level and general CS self-efficacy level by condition, reported in *M (SD)*, *Mdn* (25th percentile - 75th percentile) format

Category (Score range)	**PS** (N=41)	**NP** (N=48)
pre-practice knowledge in `class` (4-20)	9.3 (4.9), 8.0 (4.0-13.0)	8.8 (4.3), 7.5 (5.8-12.0)
general CS self-efficacy level (6-30)	21.1 (4.8), 21.0 (18.0-24.0)	19.7 (4.6), 19.0 (17.0-23.0)

3.3.4 Results

3.3.4.1 RQ3: Are there differences between conditions in terms of practice performance, problem-solving efficiency, and posttest performance?

Descriptive statistics on student responses to the survey items are included in Table 3.5. As shown in Table 3.5, on average, students in both conditions reported a pre-practice knowledge level between "I know what it means, but have not used it in a program" (8 total points) and "I have used this concept in a program, but am not confident about my ability to use it" (12 total points). This demonstrated that, after receiving basic direct instruction with worked examples, these students gained some confidence in their ability to write code around `class`. However, they had not become experts on this topic. This indicated that those participants reached the expected skill level for this study. In addition, as the conditions (NP and PS) were randomly assigned, we expected students in both conditions to have a similar level of self-rated prior knowledge in Python `class` and general CS self-efficacy before they started the practice. Given that their self-rated pre-practice knowledge and general CS self-efficacy were not normally distributed, we applied two Mann-Whitney U tests, and our statistical results indicated that there were no significant differences between PS and NP conditions on their basic pre-practice knowledge (U = 1008.0, p = .846, CLES = .51) or general CS self-efficacy (U = 1146.0, p = .182, CLES = .58), suggesting that the condition groups were comparable. We also used Cronbach's α reliability test to check the internal consistency of the survey questions, resulting in α = 0.82 for the general CS self-efficacy survey and α = 0.80 for the pre-practice knowledge in Python `class` survey, demonstrating that these two surveys had good internal consistency [93].

We calculated each student's in-practice problem-solving efficiency using the likelihood model [109]. A learner would have a larger relative gain and be seen as more efficient if they spent less time while still achieving more problem-solving accuracy [109]. Following this model, for each student, the in-practice problem-solving efficiency was calculated as the ratio of practice score (Max = 40) to practice time (mins). Practice time was calculated as the time used for practice, excluding any periods of inactivity over 5 minutes. The highest problem-solving efficiency was 4.07, achieved by a student who finished all four write-code problems (40 points) in 9.83 minutes.

To examine the effect of Parsons scaffolding while controlling for prior knowledge, three linear

regression models were conducted. The dependent variables were the write-code practice score, problem-solving efficiency, and posttest score. The assigned condition served as the independent variable. Students' tutorial review time and their self-rated prior knowledge were included as covariates. Students' tutorial review time was calculated as the time spent in the knowledge tutorial after subtracting idle time over 5 minutes. Coefficients (coef) are estimates from the regression models. When controlling for covariates and having the NP condition as the baseline, the PS condition showed a statistically significant increase (coef = 9.8) in the write-code practice score compared to the NP condition, p = .003, Cohen's f^2 = 0.11 (small [48]). Similarly, when controlling for covariates and having the NP condition as the baseline, PS students showed a statistically significant increase (coef = 0.56) in the problem-solving efficiency compared to the NP condition, p = .020, Cohen's f^2 = 0.07 (small [48]). However, when controlling for covariates and having the NP condition as the baseline, PS students showed a non-statistically significant increase (coef = 2.56) in the posttest scores, p = .431.

Post hoc Analysis For those who utilized the Parsons scaffolding but still got the write-code practice wrong, we looked deeper into their final write-code submissions and the corresponding final Parsons scaffolding problem status. This resulted in a total of 47 Parsons scaffolding state & write-code state pairs. We classified these pairs into four categories based on the final Parsons scaffolding state, followed by the corresponding final code submission state: (1) *Only scanned Parsons blocks* (26 instances) - twenty-four (92%) of their final write-code submissions did not compile due to syntax errors, type errors, or name errors, one student did not write any code after viewing Parsons blocks, and one student's code compiled successfully but had logic errors and did not pass all the unit tests. (2) *Completed Parsons problems but failed write-code task* (15 instances) - twelve of them had code in the write-code box, and three left the write-code box blank. In all 12 cases, students did the final write-code submission after getting the Parsons solution but still failed to solve it. Since they already had the correct solution in hand (the Parsons solution), we examined their actual code to better understand the specific errors they made. Our results showed that, while these students completed the Parsons scaffolding problem, they still might not have acquired enough skill in writing code on this topic. For example, in six cases, students received a type error by writing `_init_` or `_str_`, which should be `__init__`or `__str__`. And two students had syntax errors in the longest line of *Song*: `return self.title + ", " + str(self.len)`, such as missing the + or miswriting as `str().self.len`. We also found one student incorrectly placed the correct solution after the default test cases for three problems. (3) *Incorrect Parsons completion* (2 instances) - both of them omitted some problem requirements and did not finish the code. For example, one student did not complete the required `def deposit(self,amount)` by leaving this part blank; (4) *Attempted to move Parsons blocks but could not finish* (4 instances) - two final code submissions did not complete the

requirements, and two did not write any code.

3.3.4.2 RQ4: In the Parsons problems as Scaffolding (PS) condition, how did students rate the usefulness of the Parsons scaffolding and why?

To get a general sense of how useful this scaffolding method was, we first looked at students' ratings on the usefulness of the Parsons problems as scaffolding. If students did not apply the scaffolding for the write-code question, they were told to skip the related rating question. The distribution of student ratings for each problem is shown in Figure 3.5. We found that for all four problems, nearly 70% of the ratings were five or more. This indicates that students generally found Parsons problems helpful in solving the write-code practice problems, which target new concepts they were learning. In addition, 71% (29 out of 41) of students in the PS condition completed the open-ended question to explain their ratings. Specifically, 19 (66%) of them explained how Parsons scaffolding helped them, while two of them reported specific challenges they faced, and the rest of the eight answers (28%) explained their ratings by discussing how difficult this practice was for them.

The average usefulness rating for those 19 students was 7.7, indicating that those students found Parsons scaffolding to be helpful. Of those 19 positive explanations, two responses only stated that Parsons problems were helpful without further elaboration. For the rest of the answers, four students (21%) stated that the Parsons problem made it easier for them to solve the write-code problem, but still learn. For example, one student wrote *"The Parsons problem gives me all the necessary elements of creating a* `class` *which I am unfamiliar with, but I still have to figure out an order which is helping me learn, but not too strenuous."* Four students (21%) thought the Parsons problem helped them learn problem-solving strategies, as one student explained, *"I feel like Parsons helped me understand the different pieces of code and how to think about the problems."* A high proportion of students (47%, 9 out of 19 students) valued the benefit of refining and extending existing programming knowledge by using Parsons scaffolding. One student claimed that *"I am still confused about* `class` *,* `method` *, and* `self`*, but these problems definitely helped me understand better,"* another one expressed a similar idea, *"the Parsons problems helped me remember how to create* `methods` *and enter arguments as necessary."* Additionally, one student elaborated on how the Parsons scaffolding helped to extend the existing programming knowledge in a more detailed way, *"I understood structurally what I had to do; I just forgot how to return one of the fortunes in the list at random, so the Parsons problem helped me figure out how to accomplish that."*

However, some participants reported difficulties when using Parsons scaffolding to complete write-code problems, as evidenced by their low usefulness ratings and corresponding negative explanations. For instance, the student who rated *Cat* three and *FortuneTeller* one did not finish

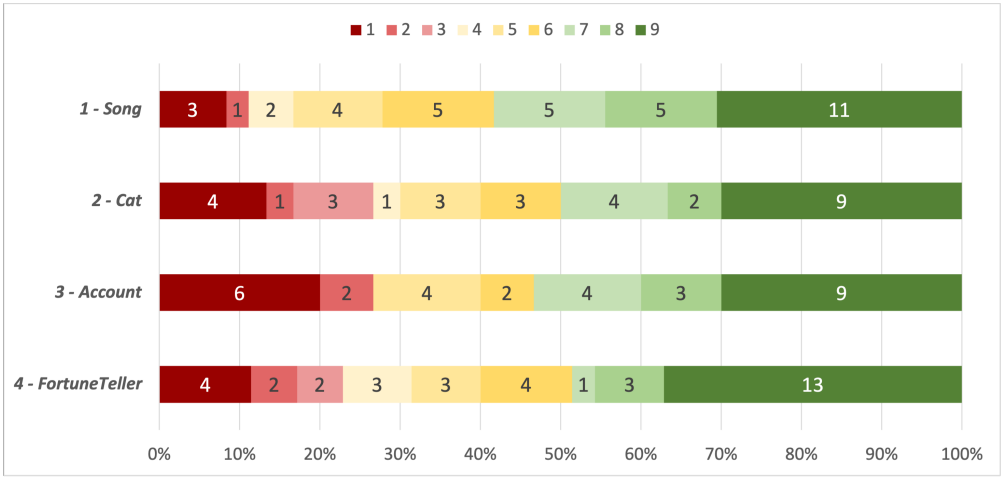


Figure 3.5: Stacked bar chart of Parsons scaffolding usefulness ratings for each practice problem.

those two problems and explained the low ratings further as *"I still do not understand* `class` *and found this exercise extremely frustrating ..."* Similarly, the student who rated *Account* one only scanned the Parsons problem and did not finish the write-code problem. This student emphasized the inherent high difficulty as *"This was really hard to just dive right into."* As mentioned above, we also had two students who explicitly explained why Parsons scaffolding was less useful for them. One of the two students used the Parsons scaffolding four times but was unable to finish any of the associated write-code problems. This student gave an average usefulness rating of 4.8 and provided the following explanation: *"I think it helps me get started, but because this is completely new to me, it merely helps me see what is there, not how to put it together."* Another student received an effective scaffolding rate of 0.5 and reported an average usefulness rating of 4.3. This student explained the rating further as *"... Parsons isn't really that helpful because you're just copying."*

3.3.5 Discussion

3.3.5.1 Parsons scaffolding is beneficial for students in terms of practice performance and in-practice problem-solving efficiency

Our study showed that using scaffolding, such as Parsons problems, can significantly enhance practice performance and efficiency when practicing code writing, while maintaining a similar level of learning performance. This result aligned with the expected outcome of providing scaffolding for practice activities on the skills that the students have not mastered yet [20, 314].

As for posttest performance, we found no significant differences between the two groups. This outcome can be explained by the inherent high difficulty of writing code for a newly learned concept from scratch. Although PS students practiced more successfully than NP students during

practice in general, it may not have been enough for them to become proficient in code writing with `class` because of the small number of practice problems we applied in this study [11]. In the future, we could use student models to track their write-code skill development and determine the number of practice problems they require to master this topic [50].

3.3.5.2 Improve the effectiveness of using Parsons problems to scaffold write-code exercise

Our analysis of student code revealed several ineffective scenarios, which led to design suggestions for different groups of students. Firstly, one possible reason why some students struggle with using Parsons scaffolding is that it may still be too difficult. Although we implemented a "Help Me" button for student-initiated difficulty level adaptation, students still had to complete at least three full attempts to activate it. It was possible that students became overwhelmed by the default multiple Parsons blocks, causing them to give up before even attempting three times. Therefore, an important enhancement of the current scaffolding approach would be providing the Parsons problems with appropriate levels of difficulty based on students' submission histories or incorrect code. In addition, to prevent students from feeling overwhelmed by encountering a new-looking problem (the equivalent Parsons problem) other than the write-code problem, it is important to establish a connection between the Parsons scaffolding and their existing code. One way to achieve this is to personalize the Parsons problem. Instead of providing the most common previous student solution, we could create a Parsons problem that is based on the student's incorrect code. Furthermore, to give students a sense of accomplishment and relatedness between existing written code and Parsons scaffolding, we could make the correct parts of the existing student code static with positive feedback [182].

Secondly, we found that some students finished the Parsons problem as scaffolding but still had errors in their write-code submissions. One potential reason is that, while these students finished the Parsons scaffolding problem, they were unable to localize their errors, such as the double leading and trailing underscores in Python, and could not use the Parsons solution effectively. Alternatively, students might prefer not to follow the Parsons solution, and therefore keep the errors in their own approach. These call for a more personalized Parsons scaffolding approach, which includes using solutions close to the students' current path and adding paired distractors that can highlight errors based on the student's current code state. Besides refining the scaffolding mechanism, incorporating explanations is another way to boost the efficiency of Parsons scaffolding. One direction to achieve this is to provide prompt guidance to make learners' self-explanation on Parsons scaffolding more productive since it helps them concentrate on relevant information [180, 205]. Another direction is adding textual explanations to either Parsons blocks or the finished Parsons solution. A prior study added explanations to next-step hints and

found that novices thought hints with explanations were much more relevant and understandable [183]. They were also better able to relate these hints to their code. We believe this method could improve students' effective scaffolding rate.

Moreover, given that we found some students finished the Parsons scaffolding and left the write-code box blank, it is possible that those students felt it was unnecessary to retype the correct Parsons solution into the write-code box. While we think that the process of retyping might enrich their comprehension and help them to identify some points that they might not have noticed without typing on their own, we could add an "AutoFill" button for those who made considerable progress when utilizing the Parsons scaffolding.

3.3.6 Limitations and Future work

One limitation of this work is that we did not collect subjective ratings or perceptions from participants who did not use Parsons problems as scaffolding. We plan to add some concrete survey questions to learn more about why they chose not to use the scaffolding. In addition, future retrospective interviews are necessary to fully understand the ineffective scaffolding cases. Moreover, we only conducted this study on one topic in a medium-scale classroom at one public university in the United States. With other demographic groups, computing domains, and educational settings, like data science and MOOCs, we might observe other Parsons usage scenarios and scaffolding effects. Furthermore, in order to save class time and reduce pretest cognitive overload for students, we only utilized self-reported measures to assess their pre-practice knowledge level in Python `class`, which may be subject to bias.

Regarding future work, we would like to investigate the impact of Parsons problems as scaffolding with other scaffolding techniques, in other programming languages, and in other educational settings. In addition, we are excited to continue our work based on the design suggestions provided above, such as providing personalized Parsons problems that are based on the student's existing incorrect solution and adapting the Parsons problem scaffolding by setting the starting state of the Parsons problem as the final state of the student's code. In addition, we look forward to reducing the cost of developing Parsons problems by automating the process based on student submissions and large language models.

3.4 Conclusion

In conclusion, this work presents two studies to understand the effectiveness of Parsons puzzles as code-writing scaffolding. Study 1 identified three representative scenarios and four use cases, and reported student perceptions on the benefits and challenges of using Parsons prob-

lems as scaffolding. Our findings showed that this approach is promising in helping students refine and extend existing knowledge, reduce their problem completion time, learn alternative problem-solving strategies, and reflect on their code writing process. Study 2 showed that using scaffolding, such as Parsons problems, can significantly enhance practice performance and efficiency when practicing code writing, while maintaining a similar level of posttest performance. Student feedback also pointed to opportunities for improving the Parsons scaffolding experience, such as providing personalized and adaptive versions of Parsons problems based on students' current problem-solving status.

Chapter 4

Prompted Self-Explanation for Improving Code Understanding

Chapter 3 examined the first theory-informed active-learning mechanism, in which Parsons puzzles scaffold novices as they write code. This chapter presents the second, turning from code writing to the complementary skill of code comprehension, and pairing prompted self-explanation with Parsons puzzle problem-solving to strengthen it. Portions of this chapter draw on material that appears in the 20th International Conference of the Learning Sciences, part of the 2026 Annual Meeting of the International Society of the Learning Sciences [116].

4.1 Introduction

Computer programming is a complex domain requiring strong cognitive and procedural skills, even from novices [261]. This makes the ability to comprehend code a foundational skill [87]. Given a piece of code, students must understand why and how a procedure works to generalize it to a broader range of problems [124]. Due to the rise of AI programming assistants, many instructors have emphasized devoting more time in intro-level courses to code program understanding and critical evaluation [178]. However, as AI can produce code that appears correct, students may even be unaware of the need to understand code, not to mention evaluate it [238].

Therefore, to enhance novices' code understanding, deliberate practice in code comprehension should be integrated with problem-solving from the beginning of CS learning. As introduced in Section 2.2.2, prompted self-explanation is an effective, theory-grounded way to deepen understanding, and it can be incorporated directly into problem-solving tasks. Within CS education, it holds promise for improving students' comprehension of code. One problem-solving activity in intro CS classes is mixed-up code puzzles (Parsons puzzles) [217]. In a Parsons puzzle, students complete code solutions without the burden of writing code from scratch, which delivers a simplified yet authentic problem-solving experience [105, 324]. Integrating prompted SE activities with Parsons puzzles can further enhance solution understanding, thereby advancing our goal of strengthening novices' code comprehension in the early stages of CS learning.

In this work, drawing from prior learning sciences research on SE prompts [242, 339], we first

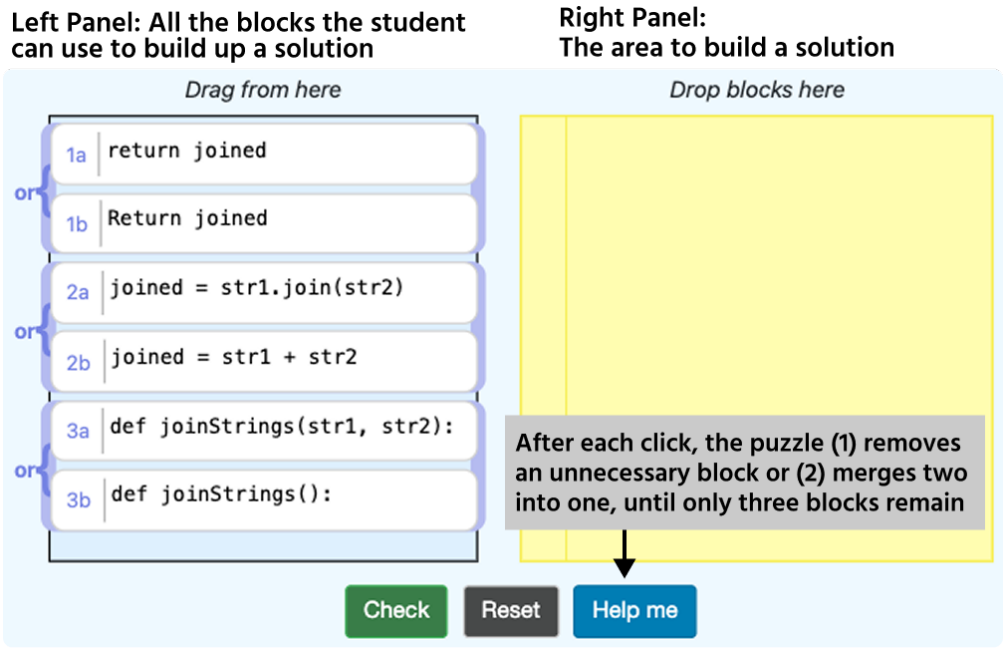


Figure 4.1: Example of an adaptive Parsons puzzle (Step 1).

designed and employed two types of SE prompts: Select and Fill. We then conducted a within-subjects study with 10 undergraduate programming novices, using both types to understand their perceived values, type preferences, and reasons. Students were asked if the SE prompts helped them understand the code, their preferred SE prompt type, and the reasons for that preference. The first two RQs were:

RQ1: Do students value SE prompts with Parsons puzzles for understanding code, and why?

RQ2: Which SE prompt type (Select or Fill) do students prefer after a Parsons puzzle, and why?

Next, we conducted a between-subjects classroom study in an undergraduate intro-level CS course with two conditions: 1) solving a Parsons puzzle followed by an SE prompt versus 2) solving only a Parsons puzzle. This study measured learning gains and students' perceived mental effort during practice:

RQ3: Do students' learning gains from pretest to posttest differ by condition?

RQ4: Does students' perceived in-practice mental effort differ by condition?

4.2 Design of two SE prompt types with Parsons puzzles: Select and Fill

Two types of SE prompts, Select (see Figure 4.2) and Fill (see Figure 4.3), were designed and deployed on an online CS learning platform. The two types were inspired by menu-based and scaffolded SE prompts in learning sciences. In Select, the student selects from a set of choices (similar to menu-based SE prompts), and in Fill, the student must type in an answer (similar to scaffolded SE prompts). As retrospective SE prompts, both were designed to prompt students to explain the code program they had just completed in a Parsons puzzle. In this practice flow, students first attempted to solve a two-dimensional adaptive Parsons puzzle (see Figure 4.1), and then they worked on an SE activity (Step 2) that was either Select or Fill.

Both Select and Fill share a three-layer structure. They begin with a complete puzzle solution at the top. Next, there is a code line summary, followed by a dash, and then the line of code. Below that, there is a detailed explanation of that line of code that may contain one or more SE areas. This structure aims to avoid the split-attention effect [35] and the associated unnecessary mental effort. Also, the line summary draws inspiration from subgoal labels in CS education [179]. In a Select prompt (Figure 4.2), there are multiple clickable areas embedded throughout the detailed, line-by-line explanation. Students need to select the correct option from a list to complete an explanation. In a Fill prompt (Figure 4.3), students need to fill in one or more blanks to complete an explanation. Once students have answered an SE prompt, they can click the "Check Me" button to receive immediate detailed feedback (see Figure 4.2 and Figure 4.3). In a Select prompt, the immediate feedback includes three parts: (1) the detailed corrective feedback; (2) visual highlights about the correct and incorrect selections; and (3) hints in natural language.

In a Fill prompt, the feedback (Figure 4.3) also includes three parts: (1) the detailed corrective feedback for each blank; (2) hints in natural language; and (3) a list of option candidates to narrow down the answer space. To accommodate answer variation, each blank accepts multiple correct answers through regular expressions and a set of predefined synonyms.

4.3 In-depth within-subjects study: Student perceived values and preferences

In the new practice flow, participants first worked on a Parsons puzzle and then answered an SE prompt (Select or Fill). We conducted a within-subjects study with 10 novices to examine their perceptions of this flow and their attitudes toward the two SE prompt types.

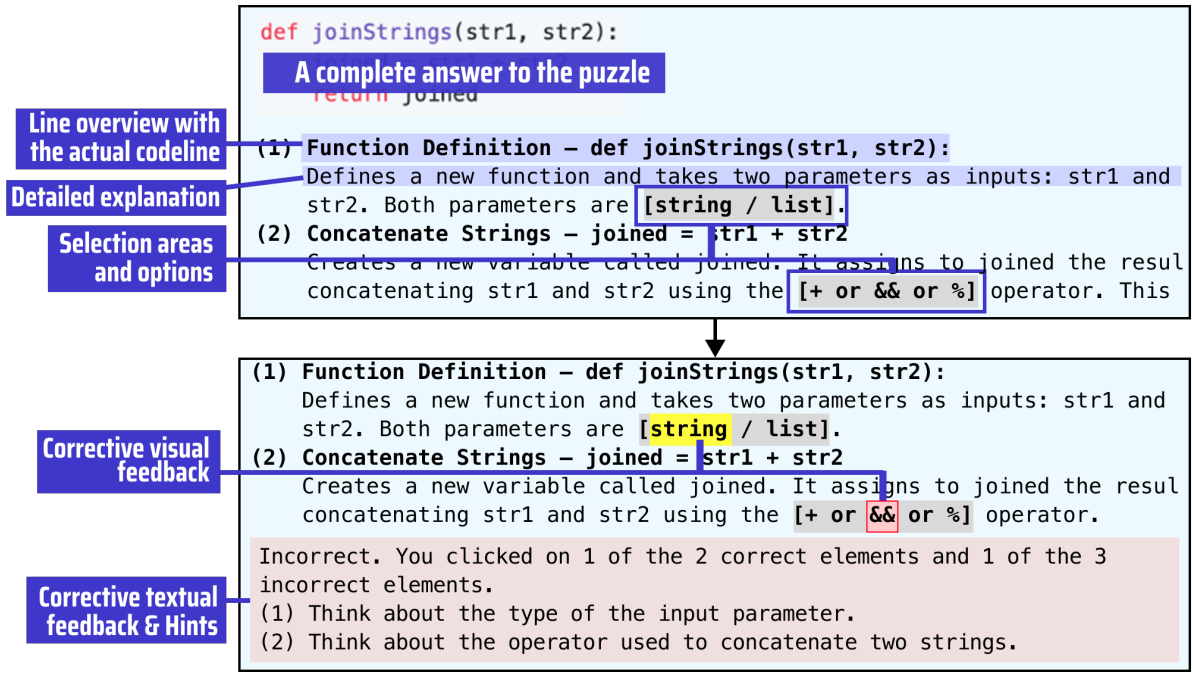


Figure 4.2: Example of a Select prompt with immediate feedback and hints (highlights; not a whole question).

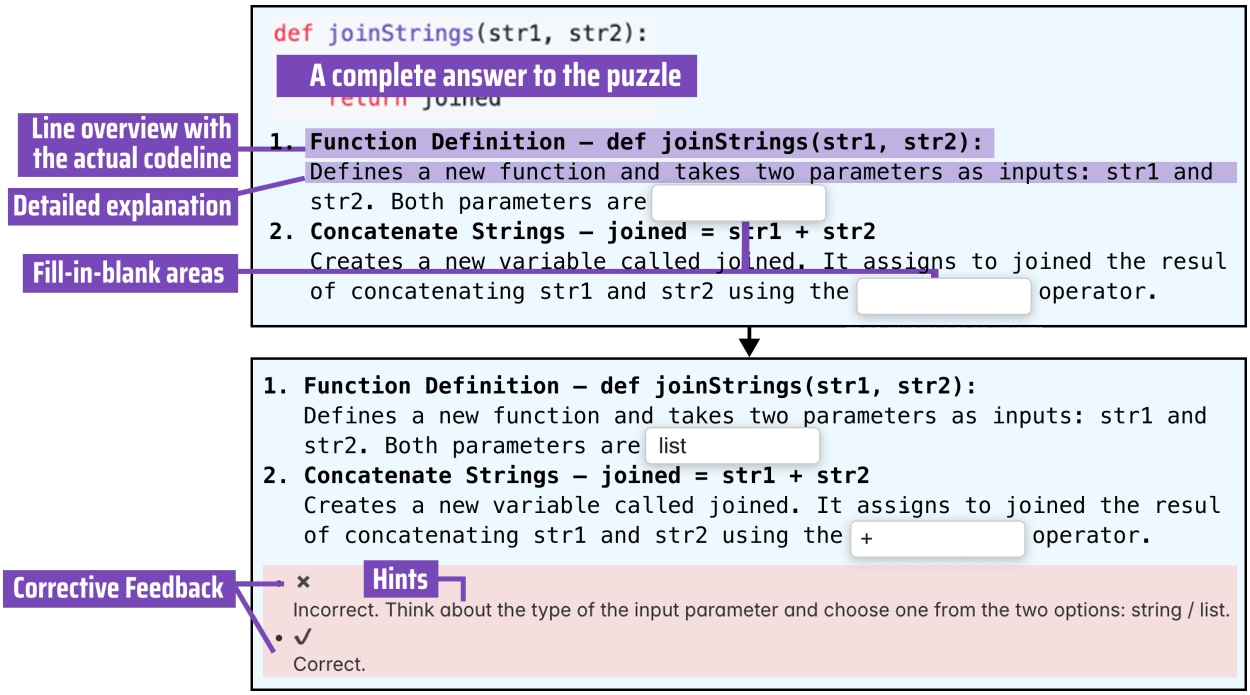


Figure 4.3: Example of a Fill prompt with immediate feedback and hints (highlights; not a whole question).

4.3.1 Method

4.3.1.1 Participants & Procedure

This study received Institutional Review Board approval as exempt. As the study targeted novices with limited CS backgrounds, we distributed a recruitment announcement in summer 2024 at a U.S. institution. We sought Python programming novices, such as students who had taken no more than two Python courses and had no additional Python experience beyond course-related activities. Sessions were conducted remotely through Zoom (about 60-90 minutes). Practice sessions and follow-up interviews were recorded. Each participant received a $25/hour gift card upon completing the study, with the amount depending on the session duration. After confirming eligibility and consent, participants received an ID and an introduction. We asked participants to solve six Python Parsons puzzles. After solving each puzzle, students received one of the two types of paired SE prompts (puzzle+Select or puzzle+Fill). To reduce order effects [230], the order of SE prompt types was counterbalanced across participants. After practice, novices were interviewed about their experience and preferences. Interview questions included *"What did you consider when you decided on your preferred explanation question type?"*

4.3.1.2 Materials

The six Parsons puzzles focused on Python 3 basics (lists and loops). For the corresponding SE prompts, a researcher with Python expertise first used OpenAI's GPT-4o model to create line-by-line code explanations for each solution to a puzzle, serving as the basis of SE prompts. Then the researcher manually refined each explanation and created blanks for Fill and clickable areas for Select. These answer areas focused on the key concepts or flow control structures of a code program. For the two SE prompt types, the same clickable (or blank) locations were used, and a consistent list of potential answers as choices (or in the feedback) was provided. As the length of the code program increased, the number of blanks or clickable areas in the SE prompt increased (e.g., five for easy and seven for hard).

4.3.1.3 Data Analysis

We first counted participants' quantifiable attitudes (whether they valued solving pairs of Parsons puzzles and SE prompts, and their preferred SE type). To better understand the reasoning behind the preference, we conducted a thematic analysis of their transcripts to identify potential insights. Two researchers independently developed initial codes from the materials, then collaboratively developed themes and co-coded one transcript to align understanding. They then independently coded the remaining data, resolving disagreements through discussion.

4.3.2 Findings

4.3.2.1 RQ1: All novices reported that the post-puzzle SE prompts supported their code understanding, even after they had solved the puzzle.

We asked whether novices felt that the SE prompts helped them better understand the code, even after completing the puzzle independently. All 10 novices gave a positive response (100%). They agreed that the SE prompts amplified the instructional effectiveness of solving a puzzle. Engaging with an SE prompt helped them move from surface-level memorization to a deeper understanding of underlying principles, making it particularly valuable. In addition, P1, P8, and P9 specifically valued this retrospective sequence of post-puzzle SE prompts: *"I liked that I could kind of view through something I was already given ... And this took a second to make me think about why I was doing what I was doing"* (P1) and *"it reinforced the puzzle you just solved ... and having that reinforcement is much better than just having the puzzle solved"* (P8).

Eight participants highlighted how answering the SE prompt helped clarify code details, such as illustrating difficult points they did not fully understand: *"it just made it a lot more sense knowing what each step was doing in the greater context of the whole program"* (P5). Similarly, P1 highlighted how the SE activity drew their attention to details by comparing answer options: *"it helped me take into account little details about the puzzle itself, and then also take into account why I didn't do other things."* Three also indicated that it helped prevent misunderstandings. When going through the SE answers, P8 recalled their puzzle-solving process and identified a previous misunderstanding, *"it's not trying to find a 2 that is next to 2, it needs every 2 to be next to the 2. Okay, that's why I was misunderstanding what was being asked."* P6 also mentioned how this process tested their initial explanations based on prior knowledge, and helped them revise misconceptions: *"the actual explanation ended up being different. So it's super helpful to have that explained"* (P6).

4.3.2.2 RQ2: Select was strongly preferred, as Fill added unnecessary, learning-irrelevant mental effort.

For the question, *"Which type do you prefer to combine with the mixed-up puzzle in general?"*, 80% of the novices had a strong preference for the Select type. A main factor influencing their preference was the high efficiency in solving SE prompts. Most said the predefined choices in Select questions sped up the SE prompt answering process: *"I think I completed these a lot faster and more correctly, just because I knew ... what were the options for me"* (P4). Students demonstrated a streamlined process through the contextual options: *"the list ones (Select) ... you kind of read through them, you can piece together what the answer is"* (P5).

The two types also introduced varying ambiguity. For Select, five noted that it provided an additional layer of scaffolding, which prevented clueless guessing. One common challenge for novices is using appropriate programming language terminologies to articulate ideas [152, 245]. Fill demanded this skill directly, but Select provided support for this: *"my difficulty sometimes is translating that accurately in the way it's supposed to show up … I think it's easier for me to see the options, and then choose from the options that align with what I already know versus trying to generate it on my own"* (P10).

In contrast, a primary concern with Fill was that the blanks led to unnecessary guessing that was no longer relevant to the underlying knowledge. P1 even described answering Fill questions as *"trying to read the mind of the test rather than actually knowing the content"* (P1). Also, although the auto-grading system included diverse synonyms, novices could come up with additional ones, which were rejected. This added the burden of evaluating exact wording in Fill, which was learning-irrelevant. By contrast, with Select, as P2 said, *"I don't have to worry about accidentally putting in the wrong synonym"* (P2).

When ranking their preferred practice types for CS learning, two students expressed a strong dislike for the Fill type, ranking it even lower than the puzzle-only option. They saw Fill as a distraction from learning, promoting system-pleasing behavior instead of learning: *"I wasn't actually thinking about why something else was wrong, and I was more just thinking about what the question was looking for from me"* (P1) and *"it was more figuring out how to game the system … how do I give it what it wants, rather than how do I answer the question"* (P8). For the two who preferred Fill, a key feature that enabled their comfort was the list of options in the feedback for wrong answers: *"it was helpful to have the different options listed once I got it wrong, and once I tried for myself"* (P6). Because of this, they valued the potential for deeper learning: *"it allows you to engage more critically with the concepts"* (P9).

4.4 Classroom between-subjects study: Student learning

We then compared learning outcomes in a between-subjects study in an undergraduate intro-level CS class. As revealed in the above section, the current Fill design may have introduced high extraneous cognitive load, potentially burdening students with unnecessary mental effort irrelevant to learning [286]. Fill was also reported to cause high frustration and potentially hinder students' ability to focus on core content [312]. Therefore, to prevent disengagement with SE questions because of frustration or cognitive overload, this classroom study only included Select. The two conditions were solving a puzzle with a Select SE prompt (PZ+SE) versus just solving the puzzle (PZ).

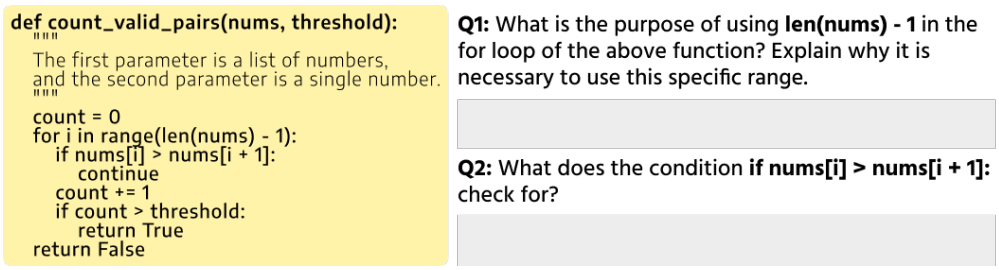


Figure 4.4: Example of two open-ended short-answer test questions.

4.4.1 Methods

4.4.1.1 Participants and Procedure

This study received Institutional Review Board approval as exempt. To align with the regular course schedule, the study was conducted as an in-class activity at the same institution in winter 2025, covering the same topic. Before the study, students were informed that the activity was part of a research study, that usage data might be collected for research purposes, and that they were not required to complete the study materials. Students were provided with researcher contact information for questions or withdrawal requests. In this activity, students completed an introduction to the practice types and a pre-survey, followed by a 10-minute pretest. They were then randomly assigned to one of two practice conditions: PZ+SE (Parsons puzzles with a Select prompt) or PZ (Parsons puzzles only). Each student completed four practice units: four Parsons puzzles in the PZ condition, or four Parsons puzzles paired with Select SE prompts in the PZ+SE condition.

After each unit, students rated their perceived in-practice mental effort. We used a 9-point Likert scale adapted from [211], the same one used in [105]. We chose this for its conciseness, which was more appropriate for our case (students had to fill out the scale four times). For each practice unit, PZ+SE students rated the mental effort required for both puzzle-solving and SE tasks as: *"During the process of solving the puzzle and (if applicable) completing the explanation question, I invested _____,"* while PZ students were asked only about puzzle-solving: *"During the process of solving the puzzle, I invested _____."* After the practice session, students completed a 10-minute posttest with isomorphic questions. In the end, a total of 110 students completed the materials as intended (54 in PZ+SE and 56 in PZ).

4.4.1.2 Materials and Grading

The practice included four adaptive Parsons puzzles. The course team validated them for proper difficulty. The pretest and the isomorphic posttest (a total of 22 points per test) included four

open-ended short-answer questions (two examples are in Figure 4.4) and one Parsons puzzle. Test puzzles included the same key elements as practice puzzles, but were not adaptive. Students did not receive any feedback on short-answer attempts. For each incorrect puzzle attempt, students received feedback that highlighted incorrect blocks. Parsons puzzles were automatically graded using the system's percent-based partial scoring. Two graders manually scored open-ended questions. One grader evaluated all answers and created an initial rubric; both then independently graded about 10% of answers before meeting to refine it. To verify the refined rubric, both graders independently re-graded the same responses, achieving an interrater reliability of 0.72, indicating substantial agreement [189], using Cohen's Kappa [47]. One grader then scored all remaining responses using the refined rubric.

4.4.2 Results

We first checked the pretest scores to see if the two groups were comparable. As the data were normally distributed, we applied a t-test. The result showed no significant condition difference in pretest performance, $t(108) = 1.31$, $p = .194$ (Table 4.1). Next, we answered RQ3 and RQ4.

Table 4.1: Test Performance Statistics by Condition (*Mean*, *SD*)

Condition	Pretest (total 22 points)	Posttest (total 22 points)
PZ+SE	$M = 9.99$, $SD = 5.69$	$M = 15.27$, $SD = 4.81$
PZ	$M = 11.43$, $SD = 5.84$	$M = 14.26$, $SD = 5.60$

4.4.2.1 RQ3: PZ+SE students had significantly higher learning gains from pre- to posttest than PZ students.

Learning gain (posttest - pretest) was used to report improvement while accounting for prior knowledge differences. Students with perfect pretests were evaluated on posttest performance only (5 students). Since the practice time was considered a common factor in SE studies [28], we added it as a covariate. Practice time per student was the sum of time spent on each practice question, where each question's time was computed by summing consecutive interaction intervals within that question, excluding gaps exceeding five minutes as idle or off-task time. We conducted a one-way ANCOVA test, and it indicated a significant condition difference in learning gain, $F(1, 102) = 4.55$, $p = .035$, partial $\eta^2 = .04$ (a small to medium effect size [48]), between PZ+SE and PZ students, while controlling for practice time. For students who achieved perfect pretests (2 PZ+SE students and 3 PZ students), we compared posttest performance by condition, controlling for practice time, and found no difference between conditions, $F(1, 2) = 0.06$, $p = .831$.

4.4.2.2 RQ4: PZ+SE and PZ students showed no significant difference in perceived in-practice mental effort.

We compared students' average perceived in-practice mental effort between conditions. We conducted a one-way ANCOVA test with practice time as a covariate, and no significant difference between conditions was found in perceived in-practice mental effort, $F(1, 107) < 0.001$, $p = .992$. This indicated that PZ students ($M = 5.4$, $Mdn = 5.0$, $SD = 1.3$) and PZ+SE students ($M = 5.5$, $Mdn = 5.6$, $SD = 1.5$) experienced similar levels of perceived in-practice mental effort, even though the PZ+SE group received paired SE prompts after code puzzles.

4.5 Discussion

This work examined whether a retrospective SE prompt after a Parsons puzzle supports novices' code understanding. Consistent with prior learning sciences evidence, all novices found the SE prompt valuable for their understanding (RQ1), echoing findings that novices often lack full comprehension of programs they produce [152]. This work also examined novices' preferences for two SE designs, Select and Fill, as well as the underlying reasons for those preferences (RQ2). Students who aimed to avoid random guessing and prioritize efficiency preferred Select, aligning with the assistance dilemma where greater assistance enables more efficient access to correct information and less frustration [138]. Since programming is complex and cognitively demanding for novices, more help in SE prompts was favored. The current Fill design might have led students to consistently report excessive, learning-irrelevant mental effort [287], as reported in the within-subjects study. One likely explanation is that the current immediate feedback mechanism could not accommodate such a relatively open-ended solution space, which resulted in false negative cases in feedback. This burden conflicted with our goal of keeping the SE prompts lightweight and learning-focused. As for future work, it is worth investigating whether real-time AI feedback [66] can better support the Fill design. In addition, based on the idea of faded scaffolding [324], future research should also examine conditions under which Fill benefits learning more than Select.

Classroom results showed that adding Select SE prompts to Parsons puzzles improved learning gains over puzzles alone (RQ3). A paired retrospective SE prompt provides deliberate practice for code comprehension, requiring students to apply existing knowledge and connect it with new concepts [41]. In our study, students reported reflecting on their understanding when comparing distractor options with the correct answer, which likely led to greater cognitive engagement as they monitored their learning. Upon finishing the SE prompt, students could also go back through the explanation to revise their knowledge and fix misconceptions. Therefore, students

gained more through practicing Parsons puzzles with SE prompts. One concern is that extra prompted SE questions may increase mental effort. However, students rated in-practice mental effort similarly for the puzzle alone and the puzzle with the SE prompt (RQ4). Since students had already invested mental effort when trying to solve the puzzle, paired SE prompts based on the same question likely added minimal additional effort. Also, our question design keeps related information spatially integrated (code and explanations) [250], limiting extra mental effort.

4.6 Limitations and Future work

This work has several limitations. (1) The studies were limited to small samples from a single institution, reducing their generalizability. Future work should explore whether this practice is also effective in other programming languages and student groups. (2) When understanding students' preferences, this work only compared two formats. Others could also be explored. (3) This practice flow was only tested as in-class practice. Future research can look into its effect in other learning contexts, such as homework and exam review, or even as an exam question. (4) Only one level of SE prompt was investigated in this work: a step-by-step explanation with an expert-decided number of incomplete areas. To address specific misconceptions, future work can personalize the prompted SE activities based on students' knowledge level and problem-solving situations. For example, SE prompts can be further personalized by the granularity of the explanations, number of incomplete areas, and locations, as well as the choices in each SE area.

4.7 Conclusion

This work explores pairing Parsons code puzzles with self-explanation prompts to foster novices' code understanding beyond puzzle solving alone. In a within-subjects study, all novices valued the SE prompts even when they had solved the puzzle. Most students preferred Select due to high efficiency and more scaffolding compared to Fill. However, the Fill design caused frustration through unnecessary mental effort that failed to support learning. A between-subjects study showed higher learning gains for students who completed pairs of Parsons puzzles and Select SEs versus puzzles alone. Both conditions reported similar in-practice mental effort, meaning the paired SE prompts did not require additional mental effort. In response to the call for strengthening code understanding in AI-era CS education, we explored various designs and provided insights from both student perceptions as well as learning evidence. Our findings showed how pairing Parsons puzzles with SE prompts can make room for code understanding practice alongside problem-solving in early CS education.

Chapter 5

CodeTailor: LLM-Powered Personalized Parsons Puzzles for Engaging Support While Learning Programming

Chapters 3 and 4 explored two theory-informed active-learning mechanisms designed to deliver engaging learning experiences. While these yielded some positive evidence, the mechanisms were the same for every student, and some students might reject the support when it was not related enough. The rise of AI now makes personalization possible, offering the potential for even more engaging experiences. This chapter begins that personalization branch: I develop CodeTailor and evaluate its effectiveness against (1) a passively delivered solution and (2) a common-solution-based puzzle. Portions of this chapter draw on material that appears in the 2024 ACM Conference on Learning at Scale [117] and the 20th International Conference of the Learning Sciences, part of the 2026 Annual Meeting of the International Society of the Learning Sciences [118].

5.1 Comparing the Effects of Personalized Active Puzzle Support versus Passively Delivered Code Solution

5.1.1 Introduction

Beginners often find learning to program difficult, and they have to invest a significant amount of time to write and debug their code [22, 264]. This process can be frustrating and lead to doubts about their ability to learn to program [96, 153]. Meanwhile, the notable increase in enrollment in intro CS courses makes one-on-one guidance from human experts hard to scale in large classrooms [83, 268]. Recently, large language models (LLMs) have opened up new ways for timely, adaptive, and scalable programming support [237]. For example, they can create complete programs directly from natural language input [133, 232]. However, as LLM products have increased adaptability and ease of use, there are rising concerns about their over-utilization in computing education [148]. Because LLMs can solve most existing CS1 programming problems [61], students may simply copy the problem description to an AI code generator and copy the solution

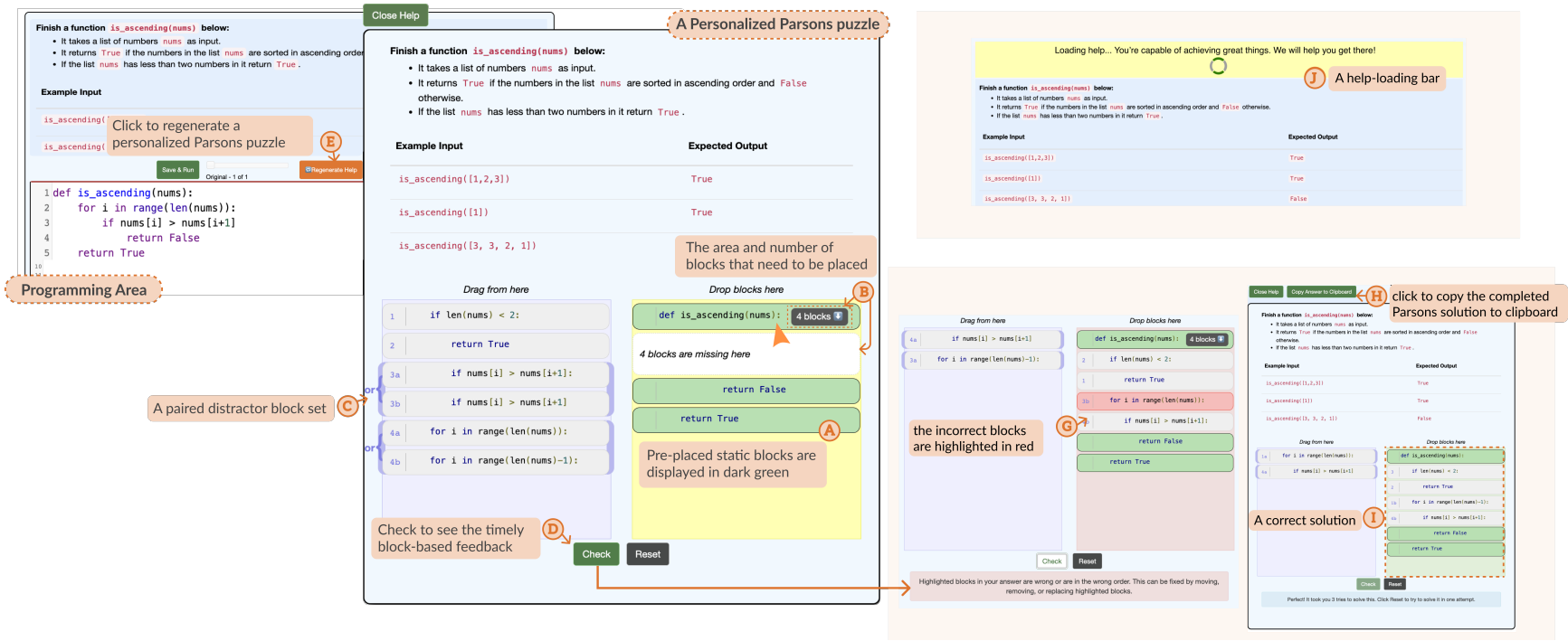


Figure 5.1: The CodeTailor interface contains a programming area on the left and a pop-up personalized Parsons puzzle as support on the right. It provides timely feedback after checking (G) and allows students to copy (H) the finished solution (I).

back to their development environment, without thinking about the AI-generated solution [133]. Another concern is that today's AI systems can generate inaccurate solutions and potentially mislead students [148].

How can we leverage LLMs to support students who struggle while practicing programming without hindering learning? We present *CodeTailor*, **a system that delivers real-time, on-demand, and multi-stage personalized puzzles** to support struggling students while programming (Figure 5.1). CodeTailor distinguishes itself from existing LLM-based products by providing an active learning opportunity where students are expected to *"solve"* the puzzle rather than simply acting as passive consumers by *"reading"* a direct solution [41].

CodeTailor supports students with personalized Parsons puzzles (see Section 2.2.1 for background). In CodeTailor, when students work on programming tasks and get stuck, they can request help, and CodeTailor provides a two-stage personalized Parsons puzzle based on their incorrect code. CodeTailor provides two levels of personalization, namely at the code solution level and at the block setup level. At the code solution level, CodeTailor creates a personalized correct solution tailored to *match* the structure, logic, and variable names in the student's existing unfinished or incorrect code. At the block setup level, the mixed-up blocks in the puzzle are adapted to the student's current problem-solving progress in three dimensions: *pre-placement of correct lines* – students' correctly written lines are pre-placed as static (not movable) blocks in the solution area; *reuse of incorrect lines* – students' incorrectly written lines are used as distractor blocks; and *conditional combining blocks* – students can combine blocks after three unsuccessful attempts on a fully movable puzzle (one in which all blocks are movable).

We conducted two evaluation studies. The technical evaluation assessed the material quality and indicated that CodeTailor can deliver high-quality *(correct, personalized, and concise)* Parsons puzzles to support students at scale without human intervention. A within-subjects think-aloud study was conducted with 18 novice programmers. The baseline condition simulated how students may naturally request help from generative AI products when having difficulty solving programming problems, i.e., asking an LLM to create a solution from the problem description [133]. Results showed that students found CodeTailor more engaging than just receiving an AI-generated solution and most students (88%) preferred using CodeTailor over passively obtaining a direct AI code solution for learning. Students could also apply more newly obtained elements from the supported practice to the posttest when using CodeTailor versus when just receiving an AI-generated solution.

5.1.2 System Design

CodeTailor offers real-time, personalized Parsons puzzles to students as support while they work on short programming tasks, a typical practice type in intro programming learning [300].

5.1.2.1 Overview of CodeTailor

CodeTailor starts with a traditional short programming task with a description, an input programming area, and a "Save & Run" button to execute the code and display the unit test results. It includes a "Help" button to trigger a personalized Parsons puzzle as support. After clicking the "Help" button, students first see a loading bar containing a spinning loader with encouragement (Figure 5.1.J). Once loaded, students see a one-dimensional personalized Parsons puzzle.

This personalized Parsons puzzle is made using a correct solution that is tailored to align with the student's existing incorrect code. When creating the puzzle, CodeTailor first separates this correct code solution into mixed-up code blocks based on the indentation level changes. Then, it offers a fully movable Parsons puzzle or a partially movable one to students, depending on their code progress. In a fully movable Parsons puzzle, students receive a set of **normal movable blocks** that are part of the solution (Figure 5.2A-P1) with a "Combine Blocks" feature (Figure 5.2A-K). This feature can be activated when students have made three failed complete attempts. When activated, it merges two blocks into one to reduce the difficulty of the puzzle. However, this feature is disabled when only three movable blocks are left.

In a partially movable Parsons puzzle, there are three types of code blocks. Aside from normal mixed-up movable blocks for a solution, we also included **static correct blocks** (Figure 5.1.A), which are pre-placed in the solution area to make the puzzle more concise to solve, and **paired distractor sets** (Figure 5.1.C), which are unnecessary blocks that emphasize misconceptions. Static correct blocks are created from the student's correctly written code (Figure 5.2B-P3). When receiving a partially movable Parsons puzzle, these blocks are already placed in the solution area with a dark green background and are not movable (Figure 5.1.A). White areas at the top and bottom of the static correct blocks indicate how many blocks are still required in that area to solve this puzzle. Students can also hover over a static correct block to check the number of blocks missing above and below it in the relative location (Figure 5.1.B).

In a partially movable Parsons puzzle, there are also paired distractor sets. Each pair consists of two blocks connected with a purple edge and an "or" (Figure 5.1.C). For the two movable code blocks in one paired distractor set, only one code block is necessary for the correct solution. In CodeTailor, the distractor code block is either created using the student's incorrectly written code (Figure 5.2B-P2) or generated using an LLM from the student's correctly written code (Figure 5.2C-P4).

When solving the personalized Parsons puzzles in CodeTailor, students can use the "Check" button to check their answer and receive block-based feedback that highlights the incorrect blocks (Figure 5.1.G). After completing a puzzle (Figure 5.1.I), students can copy the solution to the clipboard with the "Copy Answer to Clipboard" button instead of retyping it (Figure 5.1.H). Students can also regenerate a personalized Parsons puzzle (Figure 5.1.E) at any point. This gives students more control when they are unsatisfied with the current personalized Parsons puzzle or when they need updated assistance after changing their code.

5.1.2.2 Personalization Scenarios in CodeTailor

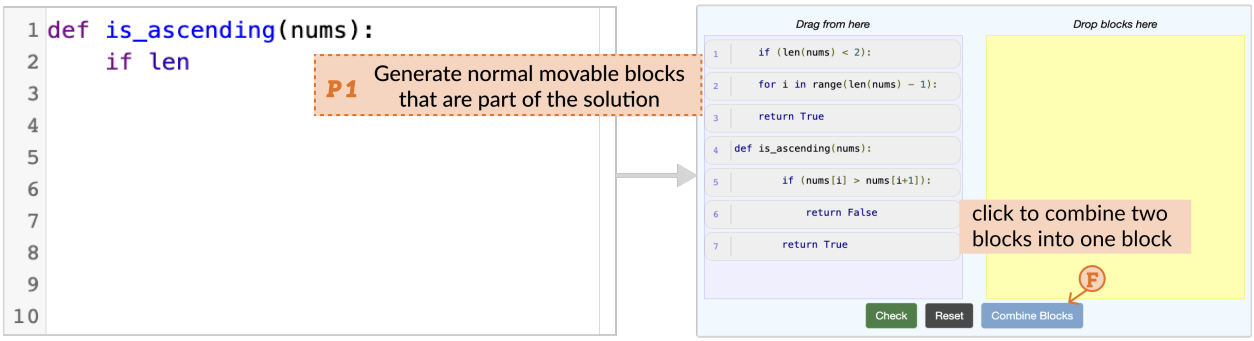
In this section, we use three hypothetical student scenarios to demonstrate CodeTailor's personalization in more detail.

Scenario A: Receive a fully movable Parsons puzzle that includes the "Combine Blocks" feature and no distractors (Figure 5.2A). After learning basic Python concepts, *Rita* practices her programming skills with CodeTailor. She tries to type one line, but gets stuck and requests help. As Rita is in the early problem-solving stages and has not written much code, CodeTailor provides a fully movable Parsons puzzle without distractors (Figure 5.2A). Rita tries to solve the puzzle but still struggles after four tries. She then combines two blocks into one (Figure 5.2A.F). She then completes the puzzle, retypes the puzzle solution, and passes all unit tests.

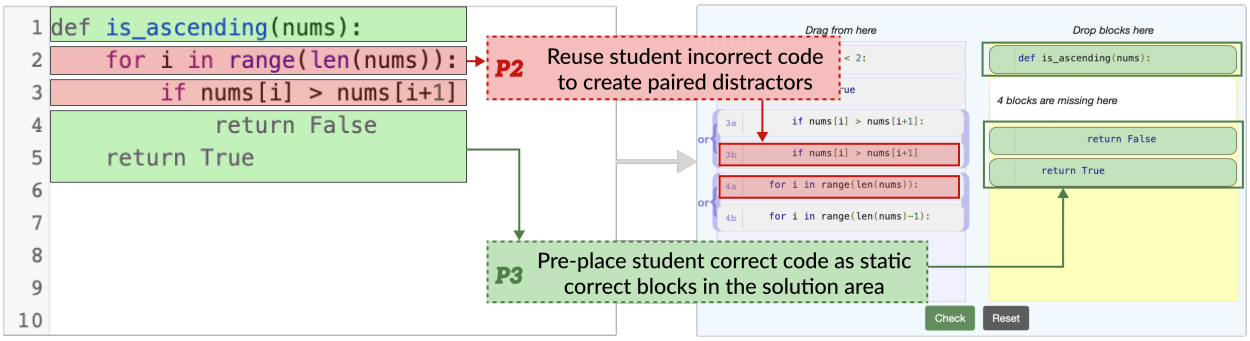
Scenario B: Receive a partially movable Parsons puzzle with paired distractors created from student errors (Figure 5.2B). *Molly* completed an intro-level Python course three months ago and is now practicing with CodeTailor. She asks for help when she needs assistance with her code. CodeTailor provides a partially movable Parsons problem, where her correctly written code lines are set as static correct blocks in the solution area and the remaining blocks (generated by CodeTailor to fix Molly's errors) are movable. Additionally, CodeTailor generates targeted distractors using Molly's incorrectly written code. After seeing a distractor, Molly realizes that she forgot to include a colon on that line. She completes the puzzle, copies the solution, and submits it, passing all unit tests.

Scenario C: Receive a partially movable Parsons puzzle with paired distractors from AI and student errors (Figure 5.2C). *Luna* practices Python programming with CodeTailor to prepare for advanced courses. When she encounters errors after believing she has solved a problem, she asks CodeTailor for help. CodeTailor provides a partially movable Parsons puzzle. Since Luna's code only has errors on one line, CodeTailor generates a distractor set based on that error and also converts one of her correctly written lines into a movable line with an LLM-generated logical error. Luna compares the LLM-generated distractor with the other paired block to reinforce her understanding of "ascending." She then completes the Parsons puzzle, modifies

A: Receive a fully movable Parsons puzzle that includes the combine block feature and no distractors



B: Receive a partially movable Parsons puzzle with paired distractors using student incorrect code.



C: Receive a partially movable Parsons puzzle with paired distractors from both student incorrect code and the LLM.

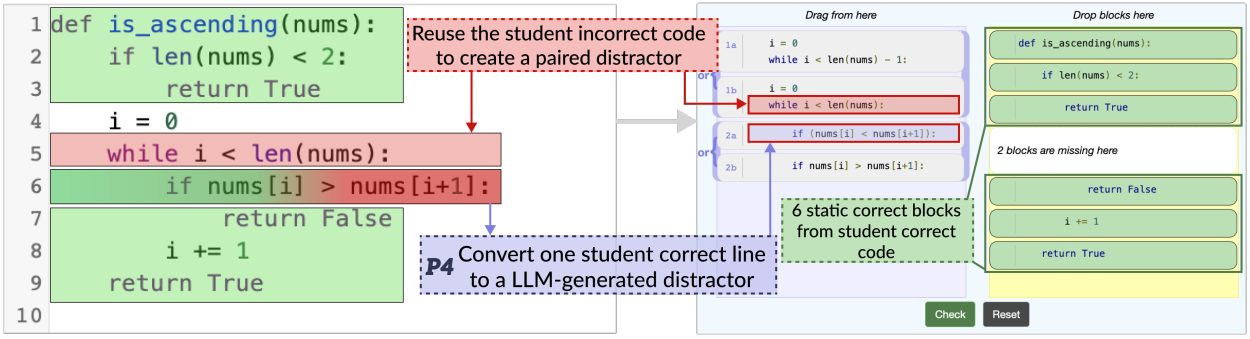


Figure 5.2: Three example scenarios with CodeTailor.

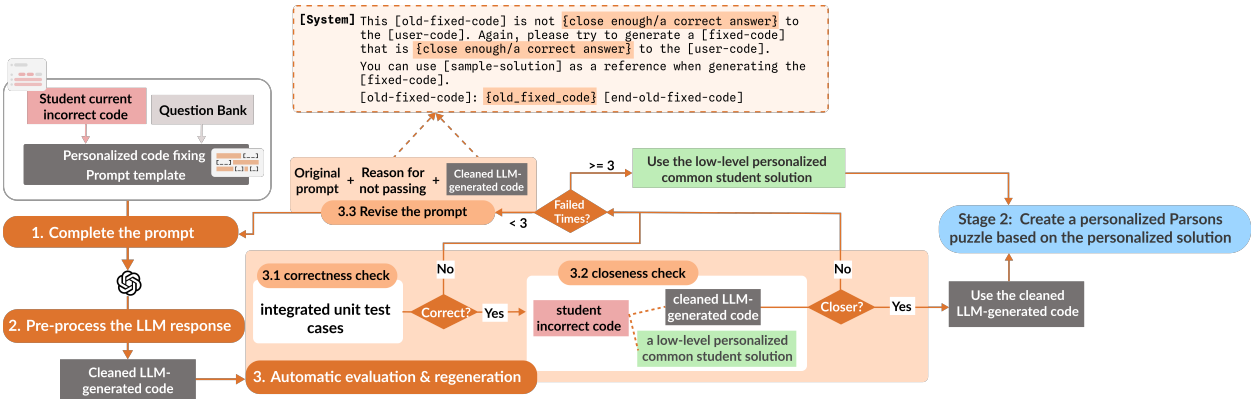


Figure 5.3: Pipeline for generating a personalized correct solution from an incorrect code in CodeTailor (Stage 1).

her code, and passes all the unit tests.

5.1.3 Implementation

CodeTailor first processes the incorrect code and generates a personalized correct solution using OpenAI's GPT-4 model. It then generates various types of code blocks from this personalized correct solution and creates an interactive personalized Parsons puzzle.

5.1.3.1 Stage 1: Generate a personalized correct solution from a student's incorrect code

Pipeline overview First, CodeTailor fills in a prompt template with the student's incorrect code and corresponding problem information (Figure 5.3-1). Then, it sends the finished prompt to the LLM (OpenAI's GPT-4 in our case) to generate a response. After receiving the response, CodeTailor pre-processes the response to only extract the LLM-generated code (Figure 5.3-2) and automatically evaluates this LLM-generated code (Figure 5.3-3) based on its correctness (Figure 5.3-3.1) and closeness to the student's incorrect code (Figure 5.3-3.2). If the LLM-generated code passes both evaluations, it continues to Stage 2 to generate Parsons puzzle blocks (Figure 5.3-Stage 2). If the code does not pass the evaluations, CodeTailor sends a request back to the LLM with an updated prompt (Figure 5.3-3.3) and requests the LLM to retry generating a solution. If the LLM-generated code fails to meet the evaluation criteria after three attempts, CodeTailor creates the Parsons puzzle based on the low-level personalized version of a representative most-common solution.

```
[System] Fix the provided Python [user-code] based on the provided [task-
         description] and [sample-solution] and generate the [fixed-code].
         The [fixed-code] should pass the provided [unittest-code] and be
         more similar to the [user-code] than the [sample-solution].
         When possible, the [fixed-code] should follow the existing solving
         strategy and solution path in [user-code], use the same type of
         [control-structures], use the same variable names as [user-code]
         and require the least amount of edits from the [user-code].
         The [fixed-code] should also follow the Python style guide.

         [task-description]: {question_description} [end-task-description]
         [sample-solution]: {sample_student_solution} [end-solution]
         [unittest-code]: {unittest_code} [end-unittest-code]
         [control-structures]: {control_structures} [end-control-structures]

[User]   [user-code]: {example_student_buggy_code} [end-user-code]
[Assistant][fixed-code]: {example_personalized_fixed_code} [end-fixed-code]
[User]   [user-code]: {actual_student_buggy_code} [end-user-code]
[LLM Output]{fixed_code_output}
```

Figure 5.4: The main prompt template used in CodeTailor.

CodeTailor's LLM prompt structures For each API request, CodeTailor uses a list of messages with three roles (system, assistant, and user) and their corresponding content to construct the prompt, following the OpenAI API reference. The system message includes the problem description, the detected control flow statements in the student's incorrect code (e.g., `if-else, for-range`), a sample student solution for this problem, and unit test cases. Then, CodeTailor applies a few-shot prompting approach, where the first user-assistant message pair is used to provide an input-output example of the desired model behavior. For example, when requesting a corrected solution, the first user message (example user) includes an incorrect code example from a previous student, and the assistant message (example assistant) provides a validated personalized corrected solution written by an expert. In the second user message, CodeTailor provides the actual student's incorrect code and asks for a corrected solution for this code from the LLM. The main prompt structure of the first attempt is shown in Figure 5.4. If the initial LLM-generated code does not meet CodeTailor's evaluation criteria (for correctness and closeness), two more requests are sent to the LLM with an updated prompt. The prompt is updated by adding an attachment to the system message that includes the reason for the failure and the LLM-generated code from the previous attempt, as shown in Figure 5.3-3.3.

Preprocessing and automatic evaluation of the LLM response After getting the LLM response, CodeTailor pre-processes it by only keeping the LLM-generated code (Figure 5.3-2). Next,

it performs an automatic evaluation of the LLM-generated code in two ways (Figure 5.3-3): (1) *correctness*: the code must pass all the unit tests integrated in CodeTailor (Figure 5.3-3.1); (2) *closeness*: the code must be closer to the student's incorrect code than the representative common solution personalized with the student's current variable names (low-level personalization) (Figure 5.3-3.2).

The most common student solution is extracted from previous student-written code for the same question. We clustered those correct solutions by comparing the edit distances based on their Abstract Syntax Tree (AST) structure [256] and selected one representative solution from the largest cluster. The low-level personalization focuses on matching the variable names used in the common solution with those in the student's incorrect solution input. It parses the student's incorrect code using regular expressions to extract variable names declared by assignment (e.g., `var = ...`) and implicit declarations (e.g., `for var in ...` or `while var ...`). It then adjusts the variable names in the solution to match the student's code (Figure 5.3-3.2-Green).

In determining how close two code pieces are, standard methods like AST edit distance [119] may be ineffective here since student-written mistakes often have bugs that cannot be processed with standard similarity methods. Also, converting the code into a vector using a pre-trained model, like CodeBERT [80] or OpenAI embeddings, can be time-consuming. Therefore, CodeTailor compares two Python code fragments by first tokenizing the code, and then calculating the similarity using the ratio between their respective token sequences [1]. It calculates the similarity between two sequences as a float in the range of 0 to 1. A value of 0 indicates no similarity, while 1 indicates that the two code pieces are identical.

5.1.3.2 Stage 2: Create a personalized Parsons puzzle based on the personalized solution

CodeTailor determines the type of Parsons puzzle to create and the corresponding block types based on three factors: overall code similarity, code similarity at the line level, and the number of corrected lines (the erroneous student code line repaired in the Stage 1 solution). CodeTailor first conducts a line-based comparison between the student's incorrect code and the personalized solution from Stage 1. Since a Parsons puzzle typically breaks code into blocks based on code lines, this comparison method allows us to prepare the fragments easily. If there are no identical code lines or the overall code similarity is low, a fully movable Parsons puzzle is provided (Figure 5.2A-P1). If the solution contains identical code lines to the student's incorrect code, a partially movable Parsons puzzle is provided, and the student's correctly written code is pre-placed in the solution area and made static (Figure 5.2B-P3). A threshold of 0.3 was used to indicate sufficient overall code similarity, which was determined through system evaluation and pilot user testing.

When building paired distractor sets, if there are more than three corrected lines, CodeTailor

pairs each corrected line with a highly similar incorrect line from the student's code as the distractor (e.g., above 0.7 in CodeTailor) (Figure 5.2B-P2). Each incorrect line can only be used as a distractor once. If the corrected lines are not sufficiently similar to any of the student's incorrect lines, they will not have paired distractors. To ensure that students with limited blocks to move have an equitable learning opportunity (Figure 5.2C), if the students have fewer than three corrected lines, CodeTailor first still matches these corrected lines with the students' incorrect lines to find potential distractors. However, if those are not available, CodeTailor converts lines with control flow keywords and the longest lines into distractor candidates. Then, CodeTailor generates the needed number of distractor blocks from these candidate lines using the LLM (Figure 5.2C-P4). Once the type of Parsons puzzle is decided and the corresponding blocks are created, the interactive personalized Parsons puzzle is then displayed to students (Figure 5.1, right).

5.1.4 Technical Evaluation: CodeTailor's Material Quality

A high-quality personalized Parsons puzzle should have a correct solution. This solution should also closely align with the student's existing code, as opposed to the most common solution. Also, by personalizing it at the block setup level, the puzzle should be more concise to solve than a fully movable one. We conducted an evaluation using incorrect code data from past students to assess the material quality produced by CodeTailor from these three perspectives.

5.1.4.1 Evaluation Data Preparation

To ensure data validity, we obtained authentic incorrect student code from an intermediate programming course in Python at a public research university in the northern US. We had IRB permission to analyze students' anonymous code. We filtered the write-code problems by topic, difficulty level, diversity of correct and incorrect code clusters, and common error types in the buggy submissions. After filtering, we selected 10 programming problems that covered various programming topics with different difficulty levels, solution strategies, and common error types. We randomly sampled 50 incorrect code submissions from each of the 10 problems, leading to 500 incorrect code inputs. To account for variation at the student level, each sampled code submission came from a unique student. Based on these 500 inputs, we obtained 500 personalized solutions from Stage 1 (Section 5.1.3.1) and puzzles from Stage 2 (Section 5.1.3.2) for evaluation.

5.1.4.2 Evaluation Results

CodeTailor can generate a correct solution from a student's incorrect code The average accuracy rate of the LLM-generated code for all incorrect student code inputs across questions

was 0.98 (*SD* = 0.13). When the LLM-generated code contained errors, CodeTailor used a low-level personalized most-common solution, as mentioned in Section 5.1.3.1. Therefore, the final CodeTailor-generated solutions were always correct.

CodeTailor can generate a correct solution closer to a student's incorrect code than a common student solution. To evaluate the personalization of the CodeTailor-generated code solution, this work used a solution selected from the most common cluster as the baseline [112]. Then, the similarity between the baseline solution and the student's incorrect code was calculated (*incorrect-baseline*), as well as between the CodeTailor-generated personalized solution and the student's incorrect code (*incorrect-personalized*), using the similarity measurement mentioned in Section 5.1.3.1. After obtaining the two similarities, a Wilcoxon signed-rank test [1] was conducted, as the normality assumption was violated. We observed a significant difference between incorrect-personalized similarity and incorrect-baseline similarity: the incorrect-personalized similarity, *M (SD)* = 0.6 (0.2), *Mdn* = 0.7, was significantly higher than the incorrect-baseline similarity, *M (SD)* = 0.5 (0.1), *Mdn* = 0.5, across student incorrect code inputs, $W = 369.0$, $p < .001$, CLES = 0.75. This indicates that CodeTailor can generate a correct solution that is more similar to the student's existing incorrect code than a common student solution.

CodeTailor can offer a more concise Parsons puzzle than a fully movable Parsons puzzle To evaluate the conciseness of the CodeTailor Parsons puzzles, the baseline was set as a fully movable Parsons puzzle with only normal movable blocks. Both puzzles were generated based on the Stage 1 solution and automatically chunked into blocks according to the indentation levels. We compared the number of movable blocks (including all the normal movable blocks and distractor blocks) for the two types of puzzles. We conducted a Wilcoxon signed-rank test and observed significant differences in the number of movable blocks between CodeTailor Parsons puzzles and fully movable Parsons puzzles across all the code inputs. Specifically, even after adding distractor blocks, the CodeTailor Parsons puzzles, *M (SD)* = 6.3 (2.6), *Mdn* = 6.0, had significantly fewer movable blocks compared to the fully movable Parsons puzzles, *M (SD)* = 8.1 (3.0), *Mdn* = 8.0, $W = 13587.0$, $p < .001$, CLES = 0.33, across student incorrect code inputs. By significantly reducing the number of movable blocks to consider, CodeTailor can offer a more concise Parsons puzzle to solve compared to a fully movable Parsons puzzle created from the same code solution.

5.1.5 User Study

Our technical evaluation found that CodeTailor can produce high-quality personalized Parsons puzzles. However, it is also important to understand if students perceive CodeTailor as engaging

[1] https://pingouin-stats.org/build/html/index.html

and beneficial for learning. We conducted a within-subjects user study with 18 students with IRB approval. This study included problems that used two different types of support to help students who struggled while programming: 1) a direct AI-generated code solution that the students could just copy to the programming area; and 2) a personalized Parsons puzzle that students had to solve before copying the solution. The research questions were:

- RQ1: Which type of support do students find more engaging?
- RQ2: How do students' practice performance, posttest performance, and application of supported elements in the posttest differ with two types of programming support?
- RQ3: Do students prefer CodeTailor or an AI-generated code solution to support their programming learning? Why?

5.1.5.1 Method

In this within-subjects study, we asked students to practice by solving four Python programming questions: two on lists and two on dictionaries. Students got the same support for the two practice questions on the same topic. Then, we asked them to complete two posttest programming problems, one for each topic, independently.

Recruitment CodeTailor targets novice students with limited Python programming skills. We sent out the recruitment announcement to both undergraduate and graduate students at the same university. During the screening, potential participants reported their prior programming experience, along with the number and content of Python courses they had taken. Qualified participants had not taken Python courses beyond the introductory level and had no additional Python experience beyond course-related activities in the past six months. The observations were conducted remotely through Zoom, with each session lasting approximately 75 minutes. We recorded their programming sessions, survey responses, and replies to the follow-up interview. Each participant received a 30-40 USD gift card after the study, the amount depending on the actual duration. Eighteen qualified participants completed the study following the instructions and were recorded.

The Baseline Support One goal was to understand whether CodeTailor could provide more engaging and educational scaffolding compared to receiving a direct AI-generated solution, the output most AI code-generation tools produce. Therefore, when the student clicked the "Help" button, the baseline support popped up a solution that was generated by the same process in Section 5.1.3.1.

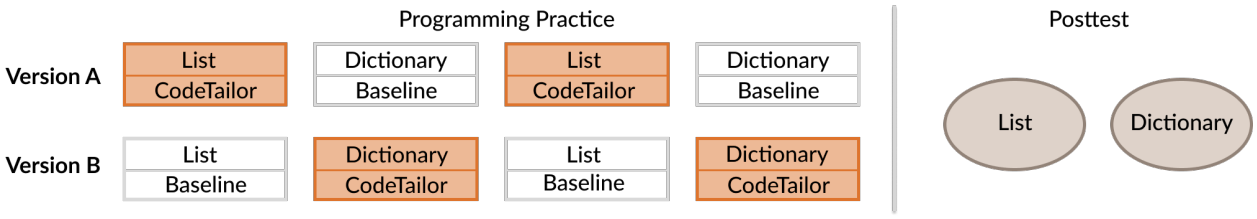


Figure 5.5: Two practice versions: In Version A, students do two list exercises with personalized Parsons puzzles (CodeTailor) and two dictionary exercises with just the AI-generated code solutions. Version B has the reverse setting.

Procedure Participants were assigned an ID based on their study date. Each participant was required to practice by finishing four programming problems with one of the two types of support (CodeTailor or a direct AI-generated code solution). The odd-numbered groups received two types of support in the sequence presented in Version A, while the even-numbered groups received two types in the sequence presented in Version B (Figure 5.5). Participants only received one type of support for the two practice questions on the same topic. This allowed us to compare their practice performance, posttest performance, and application of supported elements in the posttest by support type.

The think-aloud study started after checking ages and obtaining verbal consent to record. To further ensure that participants' Python level was within our study scope, we gave them a three-minute timed skill assessment about Python strings. Students who did not finish this assessment within the time limit moved on to the rest of the study. We provided a 15-minute introduction to the two types of support in the study. Then, to ensure enough practice with the interface, we gave an interactive example for each support type. During practice, participants solved four programming problems and rated their engagement with the two types of support. They answered a survey question, *"I feel engaged when using the above 'Help,' "* on a 5-point Likert scale (1-strongly disagree to 5-strongly agree). After the practice, participants were asked to complete a posttest with two five-minute timed programming questions (one for each topic). No support was provided during the posttest. This allowed us to evaluate students' reapplication of the supported practice elements from the two types of support in the posttest responses. The study ended with a reflective interview comparing the two types of support and asking for suggestions for improvements.

Materials Four programming problems were provided as practice questions: two questions about lists and two about dictionaries. Past student scores indicated that these practice questions had equal difficulty levels. The posttest included two programming problems that were at the near to middle transfer level for the corresponding practice questions [18, 207]. Each posttest

problem consisted of the same key elements that aligned with the practice questions on the same topic in the study.

Data Analysis For RQ1, students' engagement for each support type was calculated as their average self-reported engagement score from the two practice questions with that support. For RQ2, we first calculated the students' practice and posttest performance in each support condition (CodeTailor or baseline) based on the percentage of unit tests passed. For each student, this resulted in a maximum of 20 points for practice and 10 for posttest in each condition. Then, to understand how students retained and applied elements from the practice support to the posttest, we defined a new metric called *scaffolding apply rate*. Since the posttest question shares the same key elements as the practice questions under the same topic, two researchers iteratively developed a key element grading scheme. Each posttest question included five unique key elements corresponding to the practice questions on the same topic. After that, one researcher manually graded student practice and posttest based on this scheme. Example key elements include *"Initialize & return a dictionary," "Create a valid loop to loop through the tuples,"* and *"Check whether a key is already in a dictionary."*

$$\textit{Scaffolding apply rate} = \frac{\textit{Number of scaffolded elements applied}}{\textit{Total number of scaffolded elements}}$$

This metric separates students' level of independent practice from the supported practice. The *number of scaffolded elements applied* refers to cases where a student was initially unable to independently complete a key element during practice before using the support, but did so after using the practice support, and subsequently implemented it independently in the posttest. The *total number of scaffolded elements* refers to all the key elements with which students initially struggled but achieved after using the support during practice.

The value of the scaffolding apply rate is from 0 to 1, where zero means the student was unable to apply any elements they got from the practice support when answering the posttest. This suggests the student did not retain anything from using the support during practice. One means that the student was able to apply all the elements obtained from the practice support in the posttest.

Out of the 18 participants, 14 (77.8%) used the programming support for all four practice problems. Therefore, to balance the data, we only included these 14 participants to answer RQ1 and RQ2. If the data were not normally distributed, we used the Wilcoxon signed-rank test instead of the paired t-test. Seventeen participants (94.4%) reported having sufficient experience with both types of support, so we included them in RQ3. One participant was excluded from the analysis due to a self-reported lack of experience with supports, leading to uncertainty in preference. To

Table 5.1: Descriptive statistics on RQ1 and RQ2 metrics, reported in Mean (SD), Median format

	CodeTailor	Baseline
Perceived Support Engagement	4.5 (0.5), 4.5	3.0 (0.9), 3.0
Practice Performance	16.9 (6.1), 20.0	20.0 (0.0), 20.0
Posttest Performance	1.4 (3.6), 0	0.8 (2.1), 0
Scaffolding Apply Rate	0.3 (0.4), 0.2	0.1 (0.3), 0

further unpack how CodeTailor contributed to the quantitative results, we investigated learners' think-aloud recordings and interviews. This allowed us to gain a better understanding of students' interactions with the programming support during practice. We interpreted the transcripts and identified themes based on participants' responses and observations of their behaviors. When reporting the findings, we used (explanation) to clarify missing information in quotes and [behavior] to indicate student behaviors.

5.1.5.2 Results and Findings

RQ1: CodeTailor was perceived to provide more engaging support than an AI-generated solution. We analyzed the students' survey responses about engagement, their verbal explanations of these responses, and their corresponding behaviors. Students reported feeling significantly more engaged when using CodeTailor to solve the write-code problems, compared to receiving the baseline support, $W = 3.0$, $p < .001$, CLES = 0.89 (Table 5.1).

According to students, moving the blocks by themselves contributed to this high level of engagement. Specifically, P16 explained it as *"I felt really engaged because the drag and drop was asking me to do some stuff."* In addition, a high level of engagement also resulted from being prompted to think through the question. As P18 explained, *"because it makes me kind of think through the solution rather than just copy and paste it."* Furthermore, P7 reported, *"I could visualize where all the different parts of the problem are going to go, and it made me think more about the problem."* Conversely, participants reported feeling disengaged when using the baseline support of just receiving an AI-generated solution. For example, P1 reported, *"I didn't feel like I did anything, so it kind of feels like cheating."* P8 even directly expressed the desire to receive more engaging help, as *"there could be something like more engaging than just reading the code."*

RQ2: Students applied more supported elements on the posttest when using CodeTailor as practice support than when receiving an AI-generated solution. All the students got a full score (20 of 20) in the practice when using the baseline support. This was expected, as they could just copy the correct solution and submit it. Ten of the 14 students got full marks for the practice when using CodeTailor. Only one student (P3) gave up completing the personalized

Parsons puzzles in CodeTailor and got zero in the corresponding write-code practice. Students achieved an equal level of posttest score when using CodeTailor and the baseline support, W = 2.0, p = .789, CLES = 0.51 (Table 5.1).

We observed significant differences between CodeTailor and the baseline support in terms of students' ability to apply supported elements from the scaffolded practice to the posttest (*scaffolding apply rate*), W = 3.0, p = .041, CLES = 0.66 (Table 5.1). In other words, with CodeTailor, students applied more elements from the support during the posttest than when they simply received an AI-generated code solution. For example, one participant struggled with dictionary keys during practice. After successfully solving two practice problems with CodeTailor's support, she was then able to apply this to the posttest. She said during the posttest, *"this is something like the quantity previously (referring to a corresponding practice question)."* Similarly, another participant incorrectly compared the list length during practice, and after seeing her own buggy code that was paired with a correct code block in CodeTailor, she pointed to the correct block and said, *"I'm trying to remember if that's the way you do length. I think this might be the right one."* She then answered this practice problem correctly and reapplied this key element in the posttest.

RQ3: Most students preferred to use CodeTailor to support learning than just receiving an AI-generated solution. We asked the participants to compare their experiences with CodeTailor versus the direct AI-generated solution and their preferences. In general, 15 (88%) out of the 17 students preferred to use CodeTailor for learning. Below, we will further explore the reasons why CodeTailor was preferred to support learning.

CodeTailor provided a hands-on and engaging way for students to co-create a correct solution with appropriate effort. Most participants preferred CodeTailor because it is more interactive and allows them to invest the right amount of effort to achieve a solution. CodeTailor had already reduced the solution space and pre-placed their correctly written lines, but still gave learners enough freedom to explore. For example, P7 expressed it as, *"It not exactly like just giving you the answer key. It's still having you do a bit of work on your part to figure out what you should be doing correctly. And I think that's helpful for learning what exactly to do."* P5 also appreciated the timely feedback in CodeTailor during the exploration, *"I like the mixed-up one (CodeTailor) because it allows for me to engage and make my own personal guesses, but it also provides feedback."* P12 specifically appreciated the "Combine Blocks" feature when he was unable to identify the issue with his puzzle after 15 failed attempts. P5 utilized the "Combine Blocks" feature to obtain the correct solution with less frustration.

CodeTailor encouraged students to think about the construction of the solution. Because CodeTailor included a correct answer in the mixed-up order, students were more encouraged to think when ordering the blocks. Their main focus shifted to understanding the program

flow. Participants pointed out that CodeTailor helped them think better about this process compared to just showing them a direct code solution. For instance, P8 highlighted the learning of block relationships as *"it helps you visualize the relationships between different blocks of code more easily."* In addition, P8 also mentioned the contribution of distractors to support thinking as *"it helps for like narrowing down what you should do and helps you compare different potential ways to solve it."* P11 described it as *"(CodeTailor) helps me to better think through ... it provided context clues and helped me to learn a little bit better."*

CodeTailor fostered continuity in learning by building on past efforts without revealing the final solution. CodeTailor provides a personalized mixed-up puzzle by continuing from where the students left off. It applies students' strategies and variables in the blocks and also pre-places students' correctly written lines. As P10 mentioned, *"I think it helps you work through things the way that your brain originally thinks through them."* Also, CodeTailor allows students to correct one piece of error without revealing the entire final solution. This is particularly helpful when students get stuck on a specific part but still want to complete the rest on their own. As P16 pointed out, *"If I would have needed that (the help) earlier in the process, I think I would have been frustrated if the whole solution was already there for me."*

Personalized distractor blocks helped students diagnose errors and promoted metacognitive reflection. CodeTailor creates distractors from the student's incorrect code, which allows for more targeted debugging.

P9 emphasized its value in locating errors when debugging, saying *"Use (of) my wrong line of code to generate this multiple choice (paired distractor set) is helpful in terms of helping me to identify or locate where the bug is."* Furthermore, personalized distractors help students reflect on their thought processes, which promotes self-regulation. As P6 stated, *"it allowed me to see basically what my thought process was, and it made me think about and compare the differences between the two blocks."*

CodeTailor boosted students' confidence during problem-solving. Since CodeTailor generates the correct solution based on the student's existing code, it boosts learners' confidence during practice. As P10 said, *"I think it definitely increases your confidence, especially when you're on the right track."* Seeing CodeTailor correctly pre-place correct lines also inspired the students. For instance, when P17 saw that CodeTailor indicated that two lines were correct, she was happy and said *"I basically sort of did that piece right, which actually feels great to me."* P11 shared a similar feeling, *"It kind of helped me to confirm the correct things that I did."*

Although learners generally found CodeTailor easy to use and helpful, some reported challenges.

CodeTailor lacks sufficient explanation to facilitate understanding of difficult details. Thanks to the block-arranging activity in CodeTailor, students can understand the overall struc-

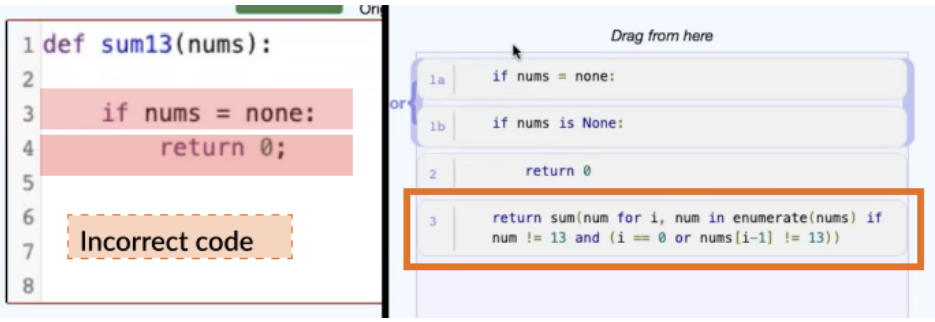


Figure 5.6: A complex shorthand one-line block in CodeTailor.

ture and main solution logic thoroughly. However, some participants had difficulty comprehending the details within individual blocks, which prevented them from successfully transferring the correct code to the posttest. For example, P17 completed the personalized Parsons puzzle in CodeTailor but had difficulty understanding some parts of the for loop. As a result, when she finished the posttest, she could not reuse the component from CodeTailor.

CodeTailor occasionally generated complicated solutions that exceeded learners' current knowledge. One student (P11) faced a situation in which CodeTailor produced a complex solution that surpassed P11's current knowledge level. Specifically, CodeTailor generated a one-line block that was too difficult for novices to digest. While the puzzle seemed easy to solve with only four blocks in total, P11 had difficulty understanding it (Figure 5.6).

Getting a direct solution (the baseline) is still preferred for quick problem-solving. Two students (P3 and P9, 12%) favored the baseline support of just receiving an AI-generated code solution. P9 liked it because of the quick error correction and side-by-side code comparison. P3 preferred receiving the AI-generated code, as P3 felt discouraged after multiple unsuccessful attempts to solve the personalized Parsons puzzle. In addition, while most participants mentioned CodeTailor's learning benefits, some of them (P1, P2, P15, and P17) also said that they might prefer getting a personalized correct solution directly if they just wanted to finish the problem quickly. For example, P1 expressed it as *"I personally like the drag-and-drop thing (CodeTailor) just because it was helping me learn. But if I wanted something really fast, then I would do the complete solution (baseline help)."* Similarly, P15 stated her preference would vary by situation, explaining, *"I think if I just want a quick way to solve that question, I'll go to the regular one (baseline)."*

5.1.6 Discussion and Future work

In this section, we discuss how CodeTailor addressed the rising concerns when applying generative AI in educational contexts, future steps to tackle challenges in CodeTailor, and implications for broader AI-based support design.

5.1.6.1 Address the concerns of LLMs in education

Two growing concerns about using generative AI in education are (1) students using AI to generate answers and finish the activity without learning (over-reliance), and (2) AI mistakes that can mislead novices [133, 148]. Motivated by these concerns, CodeTailor harnesses LLMs to generate correct, engaging, and personalized scaffolding to support development of basic programming skills at scale.

CodeTailor tries to address the concern of *over-reliance* through several aspects. First, it provides a middle-stage product (a personalized puzzle) as scaffolding, which still requires students to put in cognitive effort to arrive at a final solution. This makes the AI output indirect, allows students to apply existing knowledge [41], and prevents them from being passively engaged or even disengaged by only reading the materials [38, 290]. In addition, deeper cognitive engagement strategies could be triggered in Scenarios B and C through the selected-response activity with distractor blocks (Figure 5.2). Students can monitor and reflect on their understanding by comparing distractor blocks based on their errors with the correct ones. Second, the block-setting customization and the "Combine Blocks" feature allow the system to support a wide range of abilities. This could prevent students from feeling challenged by CodeTailor's task, which may lead them to switch to using AI tools to generate a direct solution. In addition, by chunking the solution into several blocks, CodeTailor prevents mental overload from reading a full solution at once.

Regarding the concern that AI models may be *incorrect*, CodeTailor includes an automated evaluation pipeline to check the raw LLM output, with guardrails in place to make sure the materials shown to students are correct [131]. As code solutions can be evaluated automatically through unit test cases, our pipeline is capable of being scaled to large classrooms and other scenarios. However, future research should investigate how to develop a reliable automatic evaluation pipeline for more open-ended LLM-generated educational materials, such as explanations.

5.1.6.2 Enhance the instructional effectiveness

In the student evaluation, we found some students struggled to understand the code blocks. Previous scholars found that students are unable to understand the purpose of code without a proper explanation [144, 280]. Since learners tend to read through the Parsons solution to understand each block after solving the problem, one potential way to enhance the instructional power is by adding explanations to each completed block or between blocks. Future work is needed to investigate what types of explanations are useful in Parsons puzzles as support [39].

Furthermore, although we prompted the LLM with an example novice solution, CodeTailor would occasionally generate an overly complex solution. It could be because the LLM was at-

tempting to meet another requirement, like aligning with the problem-solving strategy. Future work can explore automatic detection of over-complicated code in the pipeline, such as using keyword selection based on code styles [65, 263]. In addition, not all the students were able to complete the Parsons puzzle in CodeTailor. Future work should consider providing additional support in these scenarios. Adjusting the difficulty levels of the puzzles based on the student's current progress could also address this.

What's more, in CodeTailor, students have to wait an average of 10 seconds for the real-time LLM-based help to load due to potential multiple requests being sent to the LLM after failed code evaluation (Figure 5.3-3.3). While some students found it relatively fast, others considered it to be slow, but were not bothered because they could think while waiting. Only one mentioned that the waiting time was "noticeably long." However, students reported that the help loading page in CodeTailor with an active spinner and an encouraging sentence made the waiting time less painful. Future work should investigate better ways to reduce the delay and optimize the waiting processes to improve the learning experience.

5.1.6.3 Potential beyond programming support

Support higher-order computational skills in the AI era. With the adoption of AI programming assistants, recent work claimed that the focus for programming courses may shift from writing correct code to developing algorithmic thinking [148, 232].

By asking students to select and place code in order, CodeTailor offers a focused learning opportunity that encourages *algorithmic thinking* because students have to look at the structure and dependencies within the code [301, 333]. In this process, students also practice their *code comprehension* skills, as they must read and understand new code in the corrected solution while rearranging the code blocks. An algorithm-to-code activity could further enhance CodeTailor's ability to practice algorithmic thinking. Specifically, students could write the algorithm (processing steps of the program) in natural language. Then, if the student's algorithm was correct, CodeTailor could generate a personalized Parsons puzzle based on their algorithm. This activity would leverage AI's strength in code generation but train students' skills in high-level program design and problem decomposition [252]. By focusing on the broader aspects of program structure and logic, this activity could prepare students for the complexities of real-world computational scenarios while still harnessing the power of generative AI.

LLM-based sequencing activities in other educational contexts. The concept of sequencing pieces of a solution can be applied to other learning contexts, such as reading, foreign language, and math education [73]. For example, in math learning, sequencing items are commonly required for proofs [233]. In language learning, activities such as sequencing stories, where students rearrange mixed-up paragraphs or sentences from a story, can help them understand the

sequential flow of a narrative. This combination of story and sequencing activities can also be applied to teach K-12 students AI literacy skills, such as recognizing plagiarism with generative AI tools in different learning contexts. CodeTailor can be adapted to support these activities by simply changing the content.

5.1.7 Limitations

This work has several limitations. (1) The user study is a small-scale think-aloud study; it was not conducted in a real educational setting where students might behave differently with different preferences. (2) The effect of CodeTailor on students' long-term knowledge retention and far-transfer learning is still unclear. (3) We obtained limited feedback on the quality of LLM-produced Parsons distractors, as few cases fell into this personalization scenario. (4) We only compared CodeTailor with receiving a direct AI-generated code solution; future work could explore its effectiveness in comparison to other formats or when combined with other support features. (5) CodeTailor integrates an LLM in real-time; hence, the costs could be high for large classroom settings; future work should explore alternative methods to reduce costs.

5.2 Do Students Recognize Personalization? Comparing the Effects of Personalized versus Common Solution-Based Puzzle Support

5.2.1 Introduction

Over the past few years, researchers have advanced generative AI to provide personalized learning materials and activities [89]. Large language models (LLMs) enable the real-time generation of personalized Parsons puzzles from students' code to provide personalized scaffolding during problem-solving [117]. Section 5.1 compared personalized Parsons puzzles with directly shown correct solutions and found that students were more engaged when they received active puzzles as programming support.

However, the specific impact of this form of personalization in puzzles, compared to standard common-solution versions, is not well understood. Compared to the traditional use of common-solution puzzles [112, 114], do students recognize this personalization? How does personalization influence students' perceived alignment between their own approach and the puzzle solution, and their understanding of the scaffolded solution? These fine-grained insights are crucial to designing future AI-powered programming tools that better support personalized learning. Therefore, this work aims to address this gap in a study with two conditions: one condition had access to

LLM-powered personalized solution-based Parsons puzzles (PP), while the other always received common solution-based Parsons puzzles (CP) as write-code scaffolding. The RQs were:

RQ1: How do personalized vs. common-solution puzzles impact students' perceptions of the alignment of the puzzle solution with their own approach?

RQ2: How do personalized vs. common-solution puzzles impact students' perceived understanding of the puzzle solution?

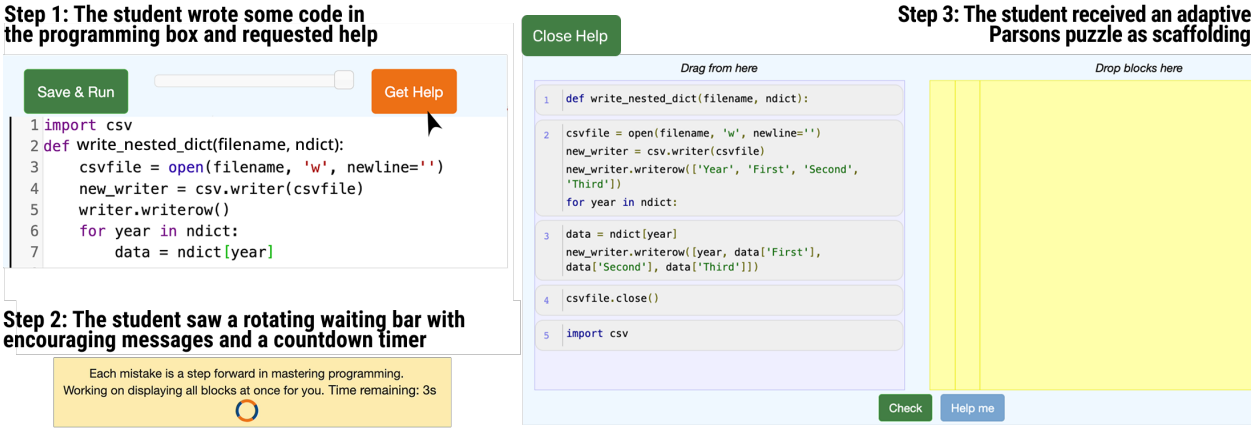


Figure 5.7: The student writes code (Step 1), requests help (Step 2), and receives a Parsons puzzle (Step 3).

5.2.2 Methods & Materials

This IRB-approved study was conducted as an in-class practice in an undergraduate Python course on CSV Readers and Writers. After a 15-minute introduction, students completed a knowledge review, pre-survey, and 12-minute pretest. They were then randomly assigned to PP or CP conditions and completed a condition-specific tutorial, which included a short write-code example problem. Students then worked on four write-code practice problems with Parsons puzzle scaffolding available on request; PP students received puzzles based on their current code, while CP students always received common-solution-based puzzles. Students were reminded to request help after 5 minutes on a problem.

After each practice problem, students completed a perception survey, from which we analyzed two items related to our RQs (Table 5.2). The first question (Q1 - Perceived Solution Alignment) asked students to rate *"How closely does the provided solution in 'Get Help' align with your initial idea for a solution?"* on a scale from 1 (very low) to 5 (very high alignment), with option 6 for those without an initial solution idea before getting help. The second question (Q2 - Scaffolding Puzzle Solution Understanding) asked students to rate *"How well did you understand the provided*

Table 5.2: Survey questions on students' perceived solution alignment and scaffolded solution understanding

Type	Question	Selection Options
Perceived Solution Alignment	Select an option from 1 to 5, or click 6 if applicable. **How closely does the provided solution in 'Get Help' align with your initial idea for a solution?**	Very low alignment (1) to Very high alignment (5) I did not have a solution idea before getting help. (6)
Scaffolded Solution Understanding	Select an option from 1 to 5, or click 6 if applicable. **How well did you understand the provided solution in 'Get Help' ?**	Not at all understood (1) to Fully understood (5) I did not arrive at a solution for the puzzle in 'Get Help'. (6)

solution in 'Get Help'?", rated from 1 (not at all understood) to 5 (fully understood), with option 6 for those who did not arrive at a solution for the puzzle. The study ended with an isomorphic posttest. AI tools were used to assist with the classroom deployment. The final sample included 85 students (37 PP, 48 CP) who completed materials without violating the intended task order. Practice consisted of four problems of varying complexity with unit test feedback. Both tests consisted of three problems at varying difficulty levels (18 points total), graded by two raters.

An LLM-driven puzzle generation pipeline was designed and deployed with integrated safeguards to ensure appropriate classroom use. Potential latency in this LLM use case could come from two main aspects: (1) response regeneration due to LLM errors or low-quality outputs, and (2) the need to fully prepare all puzzle blocks before display, since puzzles cannot be progressively streamed like text. This work dealt with these latency concerns in three ways: (1) loading screen design: displaying rotating encouraging messages with explicit time-remaining indicators to improve students' waiting experience (see Figure 5.7-Step 2); (2) generation route: to improve generation efficiency, an updated safeguard mechanism limited regeneration attempts and defaulted to a common solution if the generation exceeded a time threshold or retry limit. This is designed to maintain responsiveness and quality; (3) model: OpenAI's GPT-4o-mini model was used to meet the fast response-time needs.

The knowledge review included a description of the concepts with code examples such as *how to read a CSV file with a header row.* The practice consisted of four write-code problems of varying complexity targeting the concepts. Each question appeared on a single page. After each code execution attempt, students received detailed feedback from unit tests with error messages. Both the pretest and isomorphic posttest had three write-code problems at varied difficulty levels. They targeted the same concepts covered in the practice. The first question (missing-line question) had students write two missing lines without any feedback. The second question provided a partially

completed function and asked students to complete it; and the third question asked students to write a full function from scratch, but students could begin by outlining the main steps of the solution as detailed comments. Both tests had a total score of 18 points.

Students' test answers were graded following the process below: (1) missing-line questions were graded manually based on correctness; (2) write-code questions were first graded using unit tests; (3) for failing answers, one grader initially evaluated and came up with a grading rubric; (4) two graders independently coded around 20% of the failed answers, met to discuss differences, refined the rubric, and calibrated the grading independently, reaching a high agreement in inter-rater reliability (Cohen's Kappa = 0.87 [189]); (5) one grader finalized the rest of the grading using the refined rubric.

5.2.3 Results

We first checked students' pretest scores. As the data were not normally distributed, Mann-Whitney U tests were used. No significant differences were found on the pretest between PP students (M = 8.7 of 18, SD = 6.3, Mdn = 7.0) and CP students (M = 8.0 of 18, SD = 6.3, Mdn = 7.0), U = 953.5, p = .563. It indicated that groups were comparable at baseline, with average pretest scores near 50%, indicating basic but not yet competent knowledge [9]. Next, to outline how personalization worked in the PP condition, we calculated its personalization and safeguard rates across all problems. Each PP student received five problems with the scaffolding: one tutorial problem followed by four practice problems in the study. The four practice problems used a common student solution as a safeguard; the tutorial example problem used an expert-written one. Scaffolding was triggered 150 times across all problems in PP. Of those requests, for those who requested help with written code, the majority (81.3%) received a personalized puzzle as scaffolding, while 18.7% fell back to the safeguard solution due to invalid personalization results.

A Wilcoxon signed-rank test revealed significant pretest-to-posttest improvement across all students (W = 228.0, $p < .001$, CLES = 0.37), indicating learning in both conditions. Learning gains (posttest - pretest) were comparable between conditions (U = 747.0, p = .528). PP students (M = 4.0, SD = 5.0, Mdn = 3.3) and CP students (M = 3.4, SD = 4.5, Mdn = 3.0) showed similar improvement. For the 10 students with perfect pretests, their posttests remained high with no differences (U = 14.0, p = .813).

RQ1: Students perceived that the solution in the personalized puzzle was more aligned with their original approach than the common solution puzzle. We examined the extent to which students could perceive the "personalization" in the scaffolding they received. Following each practice problem, the student rated the alignment between their own approach and the solution in the puzzle. Since each student rated multiple times, i.e., completing the survey

Table 5.3: Results of the Mixed Linear Model (Perceived Solution Alignment as the dependent variable)

	Coef.	Std. Err.	$P > \|z\|$	[0.025	0.975]
(Intercept)	3.34	0.15	< 0.001	3.06	3.62
Scaffolding Puzzle Type (Personalized)	0.62***	0.18	< 0.001	0.27	0.97
Condition (PP)	0.06	0.22	0.781	-0.37	0.5
Group Var	0.69	0.27			

*p $<$ .05, **p $<$ .01, ***p $<$ .001. Stars are shown for predictors. The (value) indicates the treatment level.

questions for all the cases in which they requested scaffolding, the alignment rating became a "repeated measure". To understand how students rated the alignment between their approach and the solution provided in the puzzle as scaffolding (Q1), we built a linear mixed-effects model with students' perceived solution alignment as the dependent variable. We used their assigned condition and the type of scaffolding puzzle as independent variables. A random intercept was used to account for student-level differences. We excluded cases where students reported seeking scaffolding without an initial solution idea (Option 6, Q1), resulting in 230 help-seeking cases from 72 student respondents.

The results (Table 5.3) indicated that the type of scaffolding puzzle provided (personalized or not) was a significant predictor of students' perceived alignment between the solution used in the puzzle and their initial problem-solving idea ($p < .001$). In particular, the perceived solution alignment ratings increased by 0.62 units on average (a 16% increase relative to the scale range) when they received a personalized solution-based puzzle as scaffolding. Students perceived the solutions as more closely aligned with their initial problem-solving approaches when they received a personalized scaffolding puzzle, compared to a common-solution puzzle.

RQ2: Early evidence suggests personalized puzzle scaffolding improves scaffolded solution understanding via increased perceived solution alignment. A similar linear mixed-effects model was conducted with understanding of the solution in the scaffolding puzzle as the dependent variable. We excluded cases where students did not reach a scaffolded solution (Option 6, Q2), resulting in 255 help-seeking cases from 75 student respondents. However, puzzle personalization (Coef. = 0.07, p = .692) and assigned condition (Coef. = 0.27, p = .191) did not predict students' understanding of the scaffolded solution.

We then explored whether students' perceived alignment between their own problem-solving ideas and the puzzles they received (Perceived Solution Alignment) influenced their understanding of the scaffolding puzzle solution (Scaffolding Puzzle Solution Understanding). We built a linear mixed-effects model predicting scaffolding puzzle solution understanding from perceived

Table 5.4: Results of the Mixed Linear Model (Scaffolding Puzzle Solution Understanding as the dependent variable)

	Coef.	Std. Err.	$P > \|z\|$	[0.025	0.975]
(Intercept)	2.40	0.22	< 0.001	1.97	2.83
Perceived Solution Alignment	0.43***	0.06	< 0.001	0.31	0.54
Scaffolding Puzzle Type (Personalized)	-0.19	0.16	0.225	-0.51	0.12
Condition (PP)	0.22	0.17	0.211	-0.12	0.56
Group Var	0.39	0.19			

*p $<$ .05, **p $<$ .01, ***p $<$ .001. Stars are shown for predictors. The (value) indicates the treatment level.

solution alignment, puzzle type, and assigned condition, with random intercepts per student. Results indicated that the perceived solution alignment significantly predicted scaffolding puzzle solution understanding, whereas puzzle type was not a significant predictor (Table 5.4). Together, we found that receiving personalized puzzles significantly predicted higher perceived solution alignment (RQ1), which in turn was associated with greater solution understanding (exploratory analysis result), while personalization did not directly predict solution understanding.

This indicates a potential underlying mechanism [224] that the personalization of the scaffolding puzzle influenced students' understanding of the puzzle solution through the perceived alignment between the provided solution and their own problem-solving ideas. To explore it statistically, we applied a bootstrapping analysis using a 95% confidence interval (CI) [175, 176, 239]. The mediated relationship (indirect effect = 0.27, 95% CI [0.12, 0.42]) was considered statistically significant, as the CI did not include 0 [239]. This suggests a mediated pathway where personalized puzzles based on student code enhance solution understanding through higher perceived solution alignment, though this finding is exploratory and warrants further investigation.

5.2.4 Discussion, Future work, and Limitations

First, both conditions showed significant learning gains, consistent with the goal of cognitive scaffolding [21], which is enhancing students' knowledge and skill on a topic. Learning gains did not significantly differ across conditions, possibly because not all PP students were fully exposed to personalization. Personalization was based on students' incorrect code, capturing their problem-solving approaches. However, students could request scaffolding before writing any code, limiting personalization opportunities. Prior work suggests students may need Parsons puzzles as scaffolding early when unsure how to begin [112]. This design choice was informed by this need. Future work should balance required student input against personalization precision.

In addition, this work highlighted the contribution of student-code-based personalization in

Parsons puzzles as scaffolding versus the common-solution approach. These findings align with prior research showing the benefits of personalizing tutoring systems to match student strategies [8]. The personalization in this work was based on students' existing incorrect code. Students could recognize such personalization, which was reflected in the greater alignment between the solution in the puzzle and their own initial problem-solving approach. With such higher perceived alignment, they could continue their own thinking process, and did not have to interpret a new approach. This reduced the required comprehension effort. Also, high solution alignment could provide a correctness confirmation and increase their perceived competence, which might also improve intrinsic motivation to put more effort into fully understanding the puzzle solution. To make such personalization approachable, future systems could ask students to express their ideas in pseudocode, plans, or sub-goal labels, enabling personalization even before any code is written. This work has several limitations. First, this study was conducted in a single session in one undergraduate programming classroom, limiting its generalizability across contexts, demographics, and programming languages. Additionally, self-reported solution understanding and alignment may contain subjective biases. Future work should explore more behavioral measures. Moreover, while the exploratory analysis showed statistically significant effects, the limited sample size (over 500 samples have been recommended [150]) affected the stability of these exploratory results. Future work should explore whether this effect can be replicated with larger samples and in other environments.

5.3 Conclusion

In this work, we introduced CodeTailor, a novel system that delivers LLM-powered personalized Parsons puzzles to support students who struggle while programming. CodeTailor can tailor the code solution provided in the puzzle blocks to match the student's latest code, pre-place the correct written lines in the solution area, reuse the erroneous lines as distractor blocks, and combine the movable blocks on request. Technical evaluation showed CodeTailor could reliably deliver high-quality (correct, personalized, and concise) Parsons puzzles. A user study showed that students found CodeTailor more engaging and could apply significantly more supported elements from the scaffolded practice to the posttest after using CodeTailor than after just getting the correct solution. Overall, most students preferred CodeTailor for learning over just receiving an AI-generated code solution. The classroom study showed that students who received a personalized puzzle rated the solution's alignment with their own approach as 16% higher than with a common-solution puzzle. Preliminary evidence also suggests a mediated pathway through which personalized puzzles improved scaffolding puzzle solution understanding via higher perceived solution alignment.

Chapter 6

Exploring a Four-Agent Architecture with Two Student-Facing and Two Behind-the-Scenes Agents for Engaging Assessment

Chapter 5 found that personalized active-learning mechanisms can improve students' engagement and that students recognize such personalization. While a single LLM supported personalization in this case, more complex educational tasks may call for different designs, such as coordinating more than one AI. This chapter investigates the use of multiple LLM agents to extend personalization to more complex experiences composed of multiple sub-tasks, each of which a single AI may struggle to handle alone. To focus this work, the chapter examines a multi-agent approach to delivering engaging assessment while preserving enough evidence for humans to intervene and diagnose. Portions of this chapter draw on material published in the 26th International Conference on Artificial Intelligence in Education [115].

6.1 Introduction

Assessment plays a critical role in the process of learning [331], as it helps diverse educational stakeholders understand student learning progress, improve their teaching methods, and enhance educational outcomes [299]. One key prerequisite for effective test adaptation and feedback is students' sufficient effort and cognitive engagement with the assessment materials [345]. However, students often display non-effortful test-taking behaviors when completing formative assessments [4, 304, 335, 344, 346, 348], which may cause them to miss many meaningful chances to improve learning before reaching standardized tests [104, 123]. As introduced in Section 2.2.3, conversation-based assessments evaluate students through interactive dialogue [345, 347] and are typically implemented within the evidence-centered design framework to include the necessary foundational aspects of assessments [151, 349, 351]. Evidence-centered design [192, 193] is a framework for building educational assessments around evidentiary reasoning.

Nevertheless, challenges have been reported such as the difficulty of delivering relevant responses to unpredictable student input and managing diverse student interactions [97, 345]. Another high expense in creating CBAs is that multi-disciplinary development teams need to pre-

define every discourse move and map them to assessment constructs with complex NLP methods. Such limitations have hindered the widespread adoption of CBAs in real educational settings. Recent advancements in LLMs have showcased their capabilities in information understanding, generation, and reasoning [209]. Early explorations of using commercial LLM-powered chatbots to facilitate CBA agents have shown promise in terms of conversation quality and evidence produced while significantly reducing expenses [86, 350]. However, it is challenging for a single agent to manage all components of the CBA process, including dialogue facilitation, evidence collection, real-time analysis, and assessment turn control.

In this work, we propose a carefully-designed LLM-based multi-agent architecture that applies multiple LLM-enhanced AI entities to collaborate during a student's interaction with a CBA. The architecture is designed with the goals of: (1) conducting human-like natural dialogues to engage students during assessment; (2) incorporating expert-created rubrics to guide conversation moves; (3) collecting and evaluating evidence statements at different levels of detail during the assessment. By leveraging this LLM-based multi-agent architecture, we aim for cost-effective assessment that maintains engagement while emphasizing its reliability, accuracy, and validity.

6.2 An LLM-powered Multi-Agent Architecture for Conversation-Based Assessment

The overview of this multi-agent architecture is shown in Figure 6.1. It includes four LLM agents that serve different predefined roles: ***Expert Agent***, ***Peer Agent***, ***Formative Assessor***, and ***Summative Assessor***, and uses a ***Non-LLM Watcher*** to supervise and coordinate the four agents. Expert Agent and Peer Agent are the two visible artificial agents that interact with human students during the CBA; the Formative Assessor and the Summative Assessor are the two non-student-facing agents with supporting roles to operate behind the scenes. The non-LLM Watcher oversees the assessment process and manages the agent team. It is responsible for calling different agents based on different situations and making decisions to end conversations after a number of turns. We applied an organization-issued Azure OpenAI GPT model to power all the agents and the LangChain framework [2] to handle the agent prompting.

6.2.1 Agent Roles and Prompt Structures

6.2.1.1 Student-facing: Expert Agent.

This student-facing pedagogical agent serves as the primary facilitator of instructional dialogue. This agent asks different leading questions to students and provides targeted follow-up questions when needed. It needs to initiate the starting turn, maintain the middle conversation turns, and

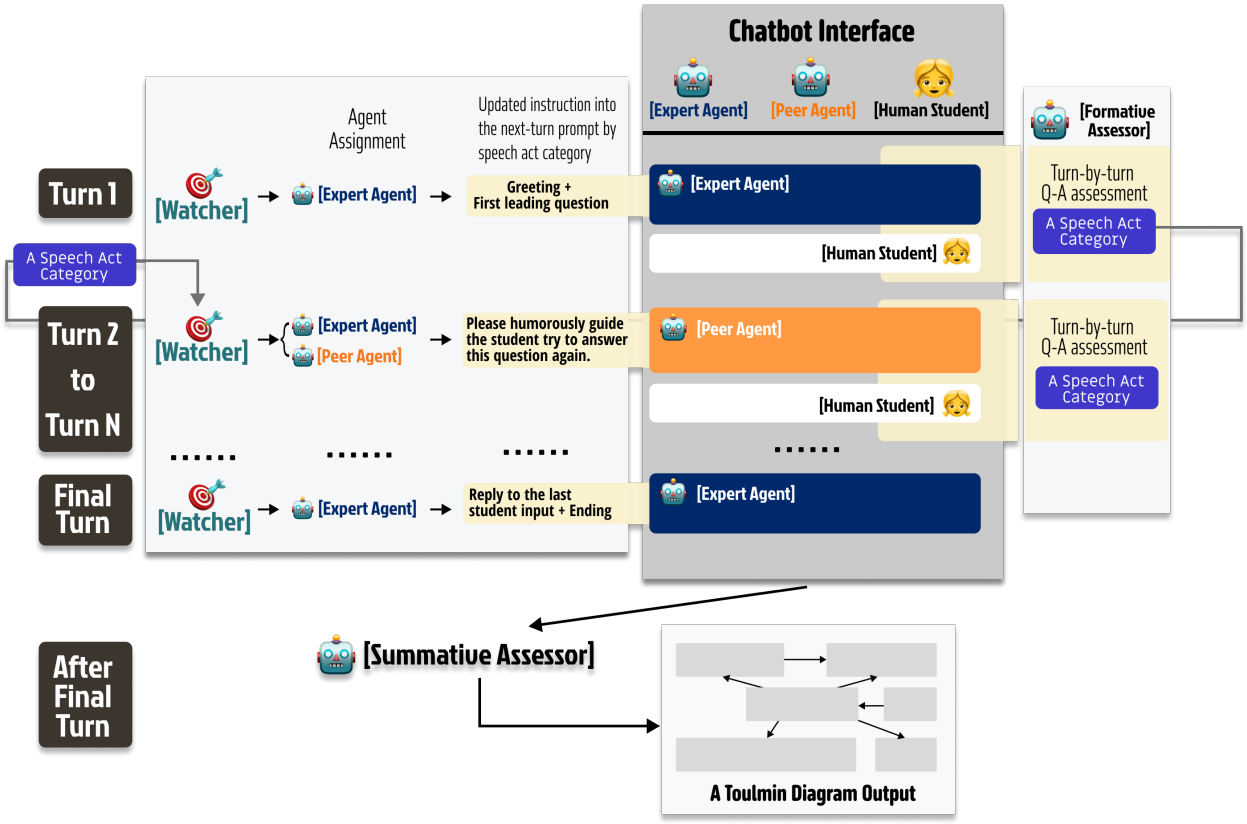


Figure 6.1: This architecture includes four LLM agents that serve different roles: Expert Agent, Peer Agent, Formative Assessor, and Summative Assessor. A Watcher is used to supervise and manage this CBA flow.

conduct the final wrap-up. Its prompt structures vary based on categorization of student input and the conversation progress. The prompt for the Expert Agent's starting turn contains these main parts: 1) a high-level introduction of the agent role, including its knowledge level (e.g., *"You are a knowledgeable friend of <student>*[1]*"*) and its main responsibility; 2) a task context overview covering the human student's knowledge level (e.g., *"You can assume that <student> know some basic concepts about weather"*), the targeted assessment level, and the specific topics that should be covered in this assessment; 3) the main conversational schema, including the application of Socratic questioning, conversation style, misconception handling, and length requirements; 4) a set of negations to prevent trial-and-error responses and help narrow the scope of the prompt; 5) a reminder of its actionable steps; and 6) the retrieved relevant domain information when applicable. For mid-conversation exchanges, more turn-level detailed instructions are updated based on Formative Assessor's speech act categorization, as shown in Table 6.2. Such instruction begins after *"Here is what you should target at when replying to the student. This instruction is based on the student's previous answer."* The final turn's prompt excludes parts 3 through 5 and instead contains ending instructions.

6.2.1.2 Student-facing: Peer Agent.

This agent's main goal is to handle specific response types in middle turns to keep students engaged in answering the current question. For example, when students' prior responses suggest they do not know the answer, this student-facing agent needs to encourage them to think more deeply and answer questions. Its prompt structure has these main parts: 1) a high-level introduction of the agent role, such as the knowledge level (e.g., *"You as <student>'s fellow student, although you sometimes make mistakes, you are much better than <student>"*) and its main responsibility (*"patient tutoring"*); 2) a task context overview including the student's knowledge level and the targeted assessment level; 3) the retrieved relevant domain information when applicable; and 4) detailed turn-level action instructions based on the student's previous response (Table 6.2). The turn-level action instructions begin with this guide: *"This is what you need to do when replying to the student."*

6.2.1.3 Behind-the-scenes: Formative Assessor.

This agent performs turn-level response type assessment by doing speech act classification for each agent-student conversation turn (see Table 6.1). These speech acts are common response types provided by students in tutorial dialogue in AutoTutor (for review see [97]), and are also useful for detecting student input in CBA. Based on the speech act category determined by the

[1] <> indicates a placeholder for the actual information, such as the human student's name in this case

Table 6.1: Guidelines used for classifying student responses into speech act categories

Speech Act Category	Guidelines for Classifying Response Types
CORRECT	Responses target the related dimensions, answer the leading question correctly, and provide enough amount of evidence towards the topic selected.
PARTIAL_CORRECT	Responses target the related dimensions, answer the leading question correctly without further elaboration.
INCORRECT	Responses target the related dimensions but answer it incorrectly.
METACOMMUNICATIVE	Responses ask for repeating the last leading question.
METACOGNITIVE	Responses indicate that the student does not know the answer to the leading question.
IRRELEVANT	Responses indicate that the student wants to talk about something that is not related to dimensions.
OTHER	Responses target at anything student respond cannot be labeled in the above categories.
INCOMPLETE	No specific guidelines

Dimensions refer to the relevant components within the selected standards.

Formative Assessor (Table 6.1), the Watcher assigns agent roles (either Expert Agent or Peer Agent) for the subsequent turn and provides additional actionable turn-level instructions to the next agent (Table 6.2). Its prompt structure includes these parts: 1) an introduction of the agent role and its main responsibility (*"label each response as one of the labels according to the following guidelines"*); 2) a set of human expert-created rubrics to classify student responses; and 3) the relevant domain-specific education standards.

6.2.1.4 Behind-the-scenes: Summative Assessor.

After the conversation, this agent generates Toulmin diagrams [192] that show how student responses to questions serve as evidence supporting arguments about concept mastery. Its prompt structure includes two main parts: 1) an introduction of the agent role and its main responsibility (e.g., *"Given <student>'s responses with <Expert Agent> in the context of <domain>, please create Toulmin diagrams showing how responses to question is used as evidence supporting assessment arguments on mastery of the concepts addressed"*); and 2) the relevant domain-specific education standards. In this architecture, those Toulmin diagrams are evaluated by human experts afterwards.

Table 6.2: Watcher's agent assignment rules and instructions for different categories (<student>, <answer>, <question>, and <domain>are placeholders)

Category	Next Agent	Turn-Level Updated Agent Instruction
CORRECT	Expert Agent	<student> answers the question (<question>) correctly, provide <student> with positive feedback first and move to the next topic about <domain>.
PARTIAL_CORRECT	Expert Agent	Please make sure you don't let <student> pass, ask <student> to provide a satisfactory explanation or elaboration on the answer <answer>.
INCORRECT	Expert Agent	Please humorously guide <student> try to answer this question (<question>) again.
METACOMMUNICATIVE	Expert Agent	Please rephrase this question only: <question>.
METACOGNITIVE	Peer Agent	Please don't give <student> the answer or provide sources or links, please acknowledge it and motivate <student> to try to answer the question.
IRRELEVANT	Peer Agent	Please humorously guide <student> back to the task (<question>).
OTHER	Expert Agent	Please humorously guide <student> back to the task (<question>).
INCOMPLETE	Expert Agent	Please provide more clues but do not provide the answer, and it's best to correct any misunderstandings by making <student> have a sudden realization.

6.2.2 Hypothetical Student Use Cases

In this section, we present two hypothetical scenarios (Student A and Student B) to briefly demonstrate how this architecture functions when students either follow or deviate from the expected CBA path. The task is assessing science inquiry skills in the context of thunderstorm formation prediction. The student-facing agents consist of "Dr. Garcia" (the Expert Agent) and "Art" (the Peer Agent). The conversation ends after 7 turns, which is predetermined by humans.

Scenario A: A student is on track to answer CBA questions. Student A got into the CBA and the Expert Agent started the conversation with "Hello Student A! This is Dr. Garcia, your friendly neighborhood weather expert. Now, let's dive into the world of thunderstorms. Can you explain how stability in weather patterns might influence the formation of a thunderstorm? Remember, we're looking for more than a simple yes or no here." Student A replied, "Stable air resists; unstable air enhances thunderstorm formation through convection." This reply was categorized as "PARTIAL_CORRECT" by the Formative Assessor. Based on this, the Watcher selected the Expert Agent as the next agent and provided actionable turn-level instruction as "Please make sure you don't let Student A pass; ask Student A to provide a satisfactory explanation or elaboration on the answer <the previous answer>." After turn 7, the Expert Agent ended this CBA by responding to the final student input with a farewell message.

Scenario B: A student tries to stay off-topic during the CBA Student B did not want to an-

swer the topic-related questions. After Expert Agent asked a leading question, Student B replied: "Who knows, I just want to enjoy the party." This response was categorized as "IRRELEVANT" by the Formative Assessor. So the Watcher assigned the next turn's agent as Peer Agent and added this instruction: "Please humorously guide Student B back to the task <the actual question>." In the next turn, Peer Agent asked "Hey Student B, it's Art. I get it, parties are great! But let's put on our science hats for a sec." Student B replied to this question as "No I don't know what you are asking." This response was categorized as "METACOGNITIVE" by the Formative Assessor. The Watcher assigned the Peer Agent for the next turn and added this instruction: "Please don't give Student B the answer or provide sources or links, please acknowledge it and motivate Student B to try to answer the question."

6.3 Student Usability Evaluation Study

To understand how this multi-agent architecture supports students in real-world assessment contexts, we conducted an IRB-approved study with 37 secondary-level students in the United States in 2024. This evaluation was conducted in the context of assessing students' science inquiry skills in thunderstorm formation prediction. In this work, we evaluated a CBA of storm formation knowledge after participants watched a short video on the topic (Figure 6.2). Each participating student completed a predetermined number of turns in this CBA, with the first leading question being "Based on the video, can you explain how the warm air at the Earth's surface moved upward to form a storm?" The Expert Agent was named "Dr. Garcia" and the Peer Agent was named "Art." We analyzed the real-world conversation data in three aspects: (1) the formative assessor's ability to categorize speech acts, (2) the follow-up turns provided by the student-facing agents based on those speech acts, and (3) whether we were able to collect evidence aligned with ECD from the Summative Assessor.

6.3.1 Data Analysis

No student answers were categorized as "INCOMPLETE" in this CBA, so the analyses focused on the remaining seven categories. To understand how the Formative Assessor tagged the student input with relevant speech acts, two domain experts developed a rubric that contained the correct answers to the main aspects in this conversation assessment. After the training period for coding, they updated the rubric to add example student responses for each question that would correspond to "CORRECT" and "PARTIAL_CORRECT" responses to help discern the nuances between the two categories. Two raters used three students' conversations for training and then double coded conversations for nine students. The raters met and reconciled all disagreements. For the

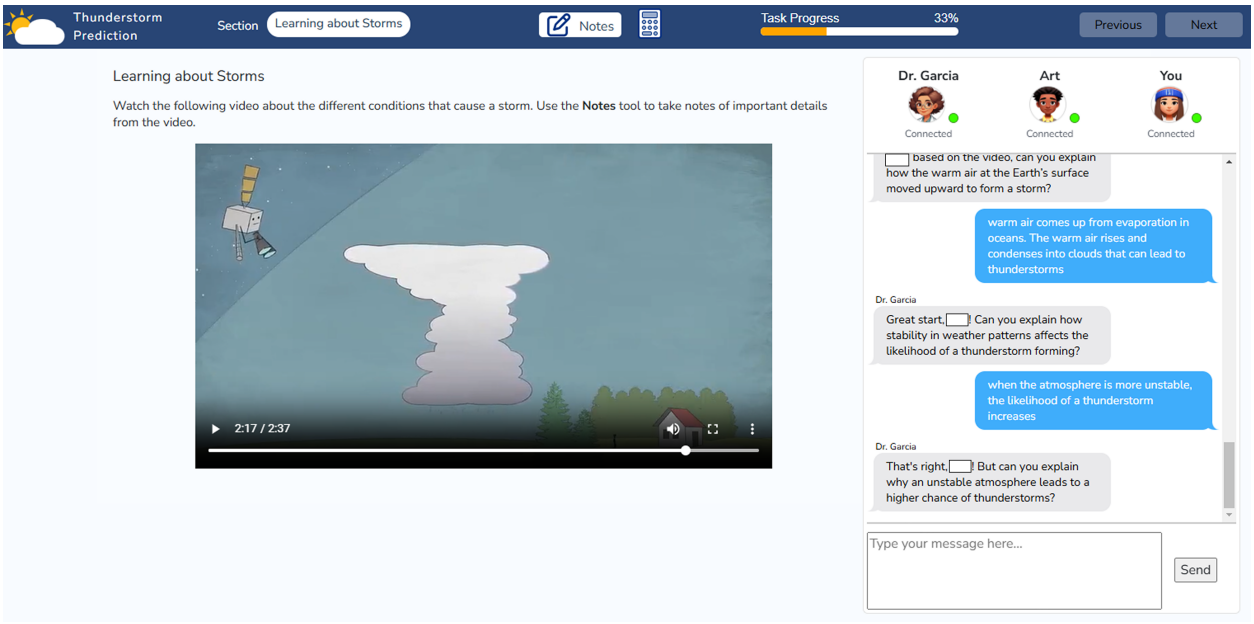


Figure 6.2: Secondary-level students (N=37) completed a thunderstorm forecasting task where they watched videos, engaged with CBAs, completed concept maps and other activities about storm formation. This work focuses on one CBA.

reliability stage of coding, the raters coded 20% of the students' data (8 students). The agreement was good (Kappa = .62). The raters then met to reconcile any disagreements to determine final codes. Disagreements were mainly driven by differences in labeling student responses as correct vs. partial correct. For the reliability coding, 80% of the disagreements were between labels of "CORRECT" and "PARTIAL_CORRECT." One rater independently coded all remaining data.

One metric that can help evaluate whether a conversational agent responded appropriately to a student input is cohesion, a metric for determining how aligned two discourse moves or essays are with each other [85, 146, 190]. This metric has previously been successfully implemented in CBAs to provide valuable insights [85]. To gain a cohesion measure of student answer input and artificial agent follow-up pairs, we used TextEvaluator [276]. TextEvaluator is a computational linguistic tool comprising several principal components derived from over 100 features and aligned with Common Core Text Complexity metrics [276]. We conducted the cohesion analysis and answer tag comparisons with 31 students who had turn-level Formative Assessor results logged successfully for this CBA.

6.3.2 Findings

6.3.2.1 Formative Assessor: Some classifications matched with human evaluation exactly, but fine-grained variations appeared in certain criteria

We compared the Formative Assessor's categorization with human raters' classification. Results indicated that the speech act tags provided by the Formative Assessor and the human coding were in agreement for 52.2% of the 155 speech acts collected from all 7 speech act categories (Table 6.1). We reported the percent agreement between the Formative Assessor and human coding because the tagged categories were imbalanced, and some ("PARTIAL_CORRECT" and "CORRECT") appeared much more frequently than others. Of the 74 unmatched cases, the main discrepancy types were as follows: 43 cases (58.1%) resulted from the discrepancies between "PARTIAL_CORRECT" and "CORRECT" classifications by human raters versus the Formative Assessor. Also, 19 cases (25.7%) came from the difference between "PARTIAL_CORRECT" and "INCORRECT" classifications. There were only 5 cases showing a mismatch between "CORRECT" and "INCORRECT" classifications, indicating that most of the disagreements came from more nuanced differences. In general, the current Formative Assessor categorized responses as "PARTIAL_CORRECT" more frequently (53.5% of all cases), while human evaluators tended to make more definitive categorizations (with only 41.3% of cases marked as "PARTIAL_CORRECT").

In addition, the Formative Assessor categorized 4 cases (5.4%) as "IRRELEVANT," but the human raters did not categorize any cases under this category. There was also one unmatched case where the student said "I am not really sure" after providing an answer. The human coder emphasized the answer part and evaluated it as "CORRECT," but the Formative Assessor emphasized the last part and categorized it as "METACOGNITIVE."

6.3.2.2 Summative Assessor: After completing the conversation assessment, it successfully generated summative Toulmin diagrams for students

When the CBA ended, the Summative Assessor was able to generate summative Toulmin diagrams based on the involved sub-topics and science standards for all the students in this CBA. For instance, for Student 04, under the main topic, the Summative Assessor was able to generate two separate Toulmin diagrams for each sub-topic including *"Toulmin Diagram for Stability and Thunderstorm Formation"* and *"Toulmin Diagram for the Role of Moisture in Thunderstorm Formation."*

Speech Act Category	Average Cohesion Score	An Example of Student Answer-Follow-up Agent Interaction
CORRECT	**45.9**	**Student:** Storm clouds are made up of a build up of moisture in the air, when the moisture is pushed up towards the sky it enlarges storm clouds until their breaking point. **Agent:** Great start! You nailed the basic idea. Now, can you explain how moisture in the air contributes to the formation of thunderstorms?
OTHER	**45.0**	**Student:** I have already responded to that question. **Agent:** Great start, <student>! Can you explain how stability in weather patterns influences the formation of thunderstorms?
PARTIAL_CORRECT	**43.4**	**Student:** warm moisture. **Agent:** Great start! But why is warm moisture important in predicting a thunderstorm? Can you explain the role it plays in the development of a storm?
META COMMUNICATIVE	**43.0**	**Student:** ? **Agent:** Hey <student>! Are you trying to figure out if a storm is about to occur? Let's dive into some science inquiry skills together! What signs or data would you look for to determine if a storm is approaching? Give it a shot!
INCORRECT	**42.8**	**Student:** thunderstorm. **Agent:** Great starting point! Let's dive deeper. How does the stability in weather patterns influence the formation of thunderstorms?
METACOGNITIVE	**39.1**	**Student:** I am not sure to be honest. **Agent:** No worries, <student>! It's totally okay to feel unsure. Let's figure this out together. Can you think of any signs or indicators that might suggest a storm is approaching?
IRRELEVANT	**38.2**	**Student:** the sky becomes darker. **Agent:** Hey <student>, you're right on track! When the sky becomes darker, it can definitely be a sign that a storm is brewing. It's like the sky's way of saying, "Get ready, something big is coming!" But remember, we're not just cloud-watching here; we need to think about other signs too. What else might you look for to confirm a storm is on the way?

Figure 6.3: Standardized average cohesion scores and an example of Student Answer-Follow-up Agent interaction for each speech act category.

6.3.2.3 Student-facing Agents: Agents were able to identify different student response types and tried to maintain topic-focused conversations

After detecting different student response types, these agents should maintain natural conversation flow and provide targeted responses to students with the goal of continuing the assessment. Results of the cohesion analysis indicated that a standardized score of cohesion (0-100) aligned with student speech acts as expected (Figure 6.3). This provides preliminary evidence that the agents' follow-up moves aligned with the student speech acts tagged by the Formative Assessor, and that the agents tried to keep conversations on topic.

In particular, for a correct student answer, the cohesion score was the highest (Figure 6.3), and the lowest score was given to irrelevant student replies. This demonstrates that agents provided different responses to students based on their varying inputs. When a student provided a correct answer, the agent's response was expected to be in a similar context to the student's input, focusing on the question being answered and preparing to move on. However, if a student said something irrelevant to the assessed topic, such as "the sky becomes darker," the agents were expected to respond with a discourse move bringing the student back on topic, which would naturally have less contextual overlap with the student's utterance. An example agent reply is "when the sky becomes darker, it can definitely be a sign that a storm is brewing...What else might you look for to confirm a storm is on the way?" (Figure 6.3). Follow-up analyses applied a linear mixed effects model to see whether the standardized cohesion scores were different based on different Formative Assessor's speech act categorization (Table 6.3). Results showed that the cohesion scores with Irrelevant, Metacognitive, Partial_Correct, and Incorrect categories (see Table 6.1 for more detailed category descriptions) were significantly or marginally significantly lower than the cohesion score of the Correct category (p = .011, p = .003, p = .026, p = .073, respectively).

Table 6.3: Results of the Mixed Linear Model (The reference level is CORRECT)

Variable	**Coef.**	**Std. Error**	**z**	**P > z**	**[0.025, 0.975]**
Intercept	45.859	0.874	52.472	<.001	44.146, 47.572
INCORRECT	-3.058	1.704	-1.795	0.073	-6.397, 0.281
IRRELEVANT	-7.606	2.984	-2.549	0.011	-13.453, -1.758
METACOGNITIVE	-6.718	2.298	-2.923	0.003	-11.223, -2.214
METACOMMUNICATIVE	-2.858	5.704	-0.501	0.616	-14.038, 8.322
OTHER	-0.856	4.101	-0.209	0.835	-8.894, 7.183
PARTIAL_CORRECT	-2.412	1.083	-2.228	0.026	-4.534, -0.290
Group Var: Student	0.017	0.281			

6.4 Discussion and Future work

6.4.1 LLM-based multi-agent architecture to give humans more control in the AIED systems for assessment purposes

Given that AI models behave like black boxes, their inherent randomness and limited controllability are hard to avoid. However, assessment in education is a unique context that demands high validity, clear evidence throughout the process, and human control. This multi-agent setup opens up multiple entry points for human experts to design, decide, and review throughout the assessment process. It also collects assessment evidence of different granularity to allow future iterations on both the systems and human evaluation. As an initial step, this architecture leverages LLMs for its adaptivity and natural dialogues while minimizing potential harmful impacts in CBAs.

6.4.2 Coordinating multiple agents introduces challenges that require fine-grained solutions

Since this mechanism involves multiple agents with distinct capabilities and goals, key challenges arise, including information sharing through communications, individual agents' memory capacity [106], and potential inconsistency among agents. One issue with Formative Assessor's current turn-level assessment is that it mainly evaluates each Q-A turn in isolation, rather than considering multiple answer parts together as in the context of expectation-misconception tailored dialogue. As a result, it often over-categorizes responses as "PARTIAL_CORRECT" when they could be marked as "CORRECT" if several related student replies were evaluated as a whole. In addition, there were some cases when Expert Agent repeated its question even when the student already answered it in a decent way. One possible reason is that the Watcher failed to properly process the information within the agent team, so the follow-up turn did not update accordingly. Future iterations should try to develop more robust error-handling procedures to better coordinate the information delivery among agents, and enhance the capability of those LLM-based agents.

6.5 Limitations

This work has several limitations. As an initial exploration of applying a multi-agent mechanism in CBA, we only conducted a small-scale evaluation in science education. The selected task domain made it impossible to isolate the effectiveness of the multi-agent mechanism itself. Additionally, the current evaluation was powered by an older GPT model, which affected its

performance. Also, this evaluation lacked baseline comparisons. Moving forward, we will refine this architecture and the prompts used to establish agents, followed by more comprehensive studies examining outcomes and student perceptions. Some immediate next steps include embedding a learner model and loading whole chat histories to all agents. We also plan to enhance the capability of the involved specialized LLM-based agents, such as assessment abilities and decision-making skills. These can be improved by using more specialized datasets to fine-tune the underlying models and exploring other prompting strategies to enhance agents' utilization.

6.6 Conclusion

This work presents an LLM-based multi-agent architecture for conversation-based assessment. It applies LLMs' unique strengths to provide engaging assessment through more natural dialogues and also allows for control on the human side with different levels of assessment evidence collected. A preliminary evaluation showed that student-facing agents can identify different student response types and maintain natural conversation flow; some of the Formative Assessor's classifications matched the human evaluators exactly, but fine-grained variations appeared; and the Summative Assessor was able to generate summative Toulmin diagrams after the conversation assessment ended. These first steps show promise for a multi-agent computing system for CBA that aligns with ECD, working toward an adaptive, engaging, reliable, and valid test-less assessment experience.

Chapter 7

Exploring a Dual-Agent Design for Personalized Non-Code Planning Puzzle Generation and Evaluation

Chapter 6 showed the potential of coordinating multiple AI agents, while also revealing coordination costs and the difficulty of inter-agent communication. This chapter applies a dual-agent design, adapting the mixed-up puzzle format from Chapters 3 and 5 to non-code planning steps. Together, these support the practice of explicit planning in problem-solving, an increasingly important skill in computing education. The material in this chapter is in preparation for future publication. Any resulting publication will draw on the work presented here.

7.1 Introduction

The landscape of computing education has undergone significant change with the development of AI in programming. AI-powered code assistants are now widely available to students at all levels, and are capable of generating complete code solutions with minimal effort [237]. This new reality, while convenient for producing syntactically correct code, exposes emerging needs in the field of computing education [88]. In terms of pedagogy, there is a growing demand for purposefully designed practice opportunities that reinforce the essential problem-solving steps central to building computational skills. For example, as generative AI began entering computing education classrooms, calls emerged for students to shift their focus toward "higher-level algorithms" during learning [19] and for assignments to move toward "algorithm design" [148].

Some efforts have begun to explore new skills and practice opportunities that are essential in this new age. One line of work addresses the new set of skills that AI has introduced to computing education. For example, as effective prompting emerges as a valued skill for high-quality code generation, researchers have explored prompt problems [63], a new type of exercise focused on formulating natural language prompts for code-generating LLMs. Recent work has also highlighted requirement articulation as an emerging skill, with efforts targeting either the articulation of requirements in prompts [173] or the use of underspecified coding tasks to train students to deal

with ambiguity [62, 219]. A separate line of work targets skills that have long been fundamental to computational problem-solving [197, 223, 253, 333] but underexplored under traditionally syntax-heavy programming instruction, including exercises focused on code evaluation [177, 246] and problem decomposition [102, 174]. However, practice opportunities that explicitly target the solution planning process remain scarce, with only a few studies investigating providing textual feedback on students' solution plans [60, 255].

To address this gap, this work introduces *LadderPuzzle*, a new exercise type that applies the Parsons puzzle (Section 2.2.1) format to non-code contexts. *LadderPuzzle* keeps this mixed-up block format, replacing code blocks with individual planning steps that lead to a correct algorithmic solution. The steps are personalized based on each student's initially proposed plan ideas, reflecting their own problem-solving approach. This activity is designed to actively engage students in the solution planning process prior to code implementation, underscoring the value of explicit planning as a deliberate step in problem-solving. This work explores the following research questions (RQs) through a within-subjects study with 13 novice programming learners. The within-subjects user study compares LadderPuzzle against an open-ended planning question designed to support the solution planning process:

- **RQ1**: How have novice programming learners engaged with solution planning in their prior programming learning?
- **RQ2**: How do students perceive the usefulness of LadderPuzzles compared to open-ended planning questions for solving programming problems?
- **RQ3**: How do LadderPuzzles and open-ended planning questions impact students' perceived readiness and actual code writing performance?
- **RQ4**: What are students' preferences and recommendations regarding the use of LadderPuzzles and open-ended planning questions in future programming learning?

7.2 LadderPuzzle: Personalized Parsons Puzzles Powered by Multi-Agent LLMs for Practicing Solution Planning

7.2.1 Overview of LadderPuzzle

LadderPuzzle has been integrated into a traditional short programming problem interface. The interface consists of a problem description, an input area, and a "Save & Run" button for executing and displaying results. When solving a short programming problem with *LadderPuzzle*, upon reading the problem description, students are first asked to write an initial plan in the input area (Figure 7.1-Left), outlining their preliminary approach or ideas. They then click "Get Ladder

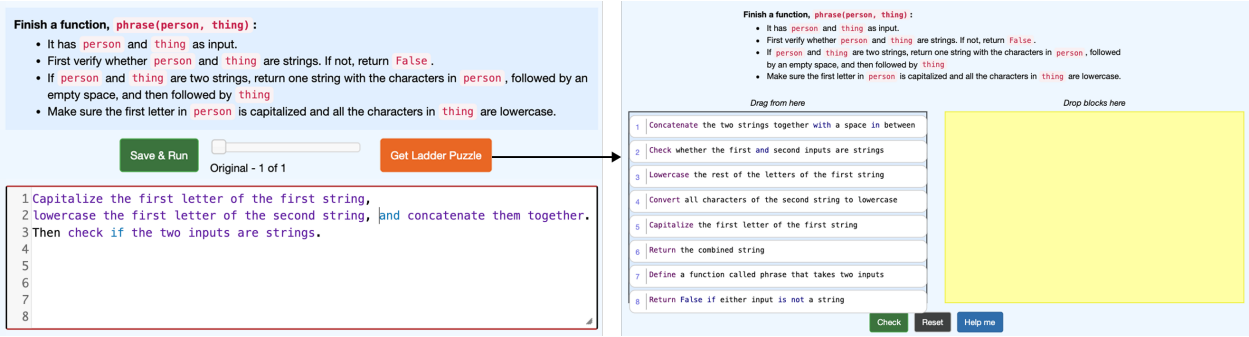


Figure 7.1: Screenshot of a LadderPuzzle: a student's initial proposed plan (left); the corresponding personalized mixed-up planning puzzle (right).

Puzzle" to receive a personalized mixed-up planning puzzle generated based on their written plan. The puzzle presents natural language planning steps in a mixed-up order, which students drag and arrange into a correct algorithmic solution (Figure 7.1-Right). In the current design, students may interact with the LadderPuzzle in any way they choose and are not required to solve it. Once they have engaged with the planning step, they may choose to clear or comment out their initial plan from the input area and proceed to write their code solution. Students may click "See Ladder Puzzle Again" to revisit the generated LadderPuzzle at any time during the problem-solving session.

To illustrate the LadderPuzzle workflow, we present an example walkthrough of a novice programming learner, Alice. Alice is tasked with solving the following problem: *"Finish the function has999(nums) below to return True if there are at least three adjacent items in the list nums that are all equal to 9, otherwise return False."* Alice begins by writing her initial plan in the input area: *"Define has999 and it takes in nums, which is a list of items. A for loop to check each item if it's a number and if equal to 9. if false return false. if true, continue to next item in nums list. store each of the item in a local list."* Upon clicking "Get Ladder Puzzle," Alice receives a personalized mixed-up planning puzzle generated from her initial plan, reflecting her own solution approach, language, and style. Alice arranges the mixed-up planning steps, completing the puzzle in three attempts, and arrives at a clear algorithmic solution. She then writes her code solution in the input area, clicks "Save & Run," and successfully passes all test cases.

7.2.2 Implementation

7.2.2.1 Stages 1-2: Generate and validate a personalized plan

LadderPuzzle adopts a two-LLM agent architecture to generate personalized planning puzzles from each student's written proposed plan. A *generation* agent produces a personalized step-

by-step plan based on the student's proposed plan (Stage 1), while a *student evaluation* agent, simulated as an introductory-level programming student, verifies whether the generated plan can be implemented as a correct code solution that passes the associated inherent unit tests (Stage 2). If the plan can be successfully implemented, it is forwarded to Stage 3 to be presented as a mixed-up planning puzzle; otherwise, the generation agent is prompted to generate a new plan.

The generation agent employs a few-shot prompting structure with two targeted examples: one illustrating the completion of a plan with missing steps, and another illustrating the correction of a plan with logical errors, both resulting in a workable personalized plan. The idea of chain-of-thought reasoning is also applied to enhance the agent's reasoning process. OpenAI's GPT-5-mini model was selected for this agent based on its balance of cost-effectiveness, reasoning capability, and response speed. The evaluation agent simulates an introductory-level programming student, who is instructed to implement a given natural language plan as code that solves the problem. Given its primary focus on code generation, the GPT-5.1-codex-mini model was selected for its specialized coding capability.

7.2.2.2 Stage 3: Generate an adaptive planning puzzle

Upon receiving a validated personalized plan from Stage 2, LadderPuzzle converts it into a one-dimensional puzzle in which each planning step is presented as a separate block. Since the current planning steps carry little logical nesting, indentation is not required. The interface also includes a "Help Me" button, adapted from the original adaptive Parsons puzzle design, which allows students to merge two blocks into one after three incorrect full arrangement attempts.

7.3 Evaluation Study

An IRB-approved within-subjects user study was designed to understand students' perceptions of *LadderPuzzles* as a new practice exercise. The study had two conditions: solving short programming tasks with *LadderPuzzle* and solving short programming tasks with an open-ended planning question as the baseline condition. In the baseline condition, students were presented with an open-ended planning question with a separate answering box: *"Before you write code, what steps will you take to approach this problem? You can write your plan in the space below."* (see Figure 7.2 for an example question, see Table 7.1 for example student answers for this question). This open-ended planning question mirrors a direct practice form of prompting students to plan their solution before coding.

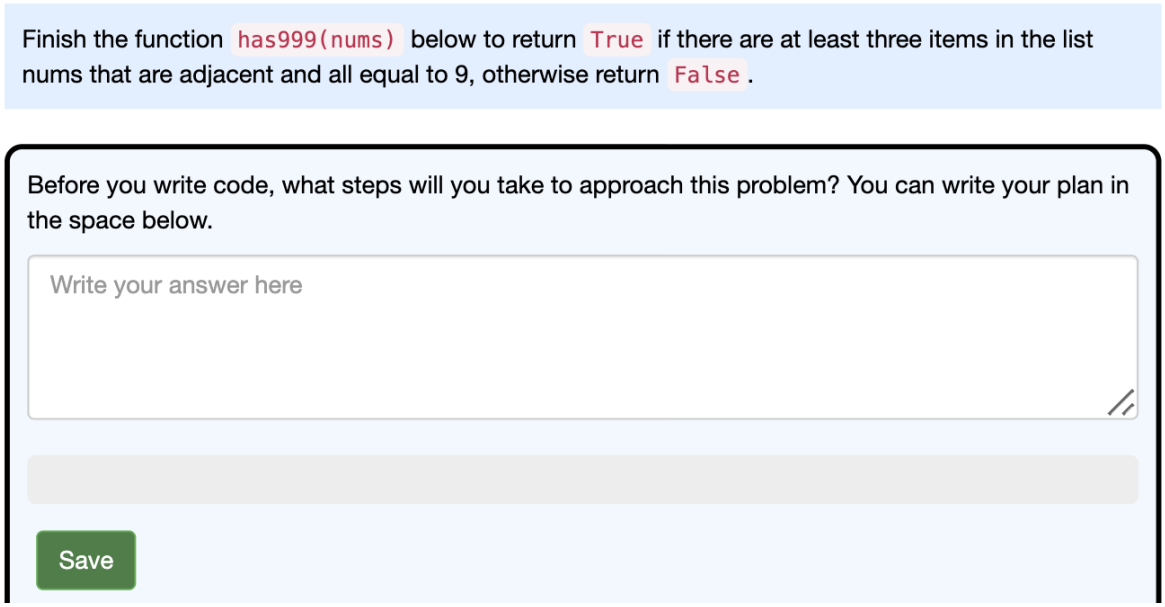


Figure 7.2: An open-ended planning question.

7.3.1 Methods

7.3.1.1 Recruitment and Participants

LadderPuzzles are intended for novice programming learners with limited programming skills. Therefore, participants were recruited from low-level Python programming courses and associated Slack channels at a large public research university. The eligibility criteria required participants to be at least 18 years old, currently enrolled at the university, have less than 12 months of active programming learning experience, have completed no more than three college-level programming courses, and have no extensive programming experience beyond course-related activities in the past six months. The studies were conducted remotely through Zoom and recorded, with each session lasting approximately 60-90 minutes. Each participant received a $40 gift card per hour after completion. A pilot study was conducted with one participant to verify the clarity and flow of the study procedure, resulting in minor refinements. A total of 13 qualified participants completed the study.

7.3.1.2 Procedure

Participants were assigned a study ID based on their study date, which determined their counterbalanced condition order (A or B, as shown in Figure 7.3). Each participant completed four short programming practice problems (Table 7.2), with two in each condition (either a LadderPuzzle or the open-ended planning question). All programming problems focused on the topics of lists

Table 7.1: Example student responses to an open-ended planning question.

	Problem ID	Open-ended planning response
P1	Practice-1	first define the function, then do a for loop to check if len list > = 3, if statement to check if num == 9 , return true, else return false.
P5	Practice-1	Using if/elif/thenstatements in the function. If there are at least 3 items, return True, if not, then: return false. Return the function at the end.
P7	Practice-1	Define the function has999 with the list nums as an input Check whether there are 3 9s in a row using a loop Using list indexes eg. i vs i+1 vs i+2 If so, return True If not, return False

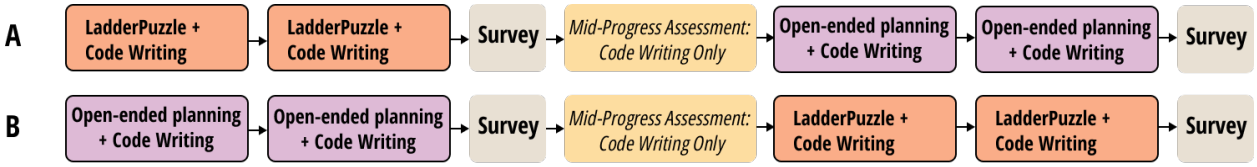


Figure 7.3: Overview of the Study Procedure.

and loops. The practice problems were adapted from regular practice problems in an intermediate course, with the difficulty increased to account for students who may have already taken such courses. The mid-progress assessment was a question combining key elements from the first two practice questions. Final question versions were refined with AI assistance. To respect participants' time, an optional five-minute timer was provided per practice problem, allowing participants to proceed to the next problem or continue at their discretion. Upon completing each condition, participants completed a survey assessing their experience and perceptions (Table 7.3), followed by one short programming problem without explicit planning steps as a mid-progress assessment task before transitioning to the next counterbalanced condition. After completing the second condition, participants completed the same survey (Table 7.3). Since students had experienced both planning practice types by this point, no follow-up assessment was conducted, as it could not isolate the effect of either type. The study concluded with a reflective semi-structured interview in which participants reflected on their prior planning experiences, compared the two planning activities, and offered suggestions for improvements.

7.3.1.3 Data Analysis

Students' perceived usefulness, perceived preparedness, and perceived improvement in solution quality were measured using self-reported survey scores for each condition. Wilcoxon signed-

Table 7.2: Problems used in the practice and mid-progress assessment

Problem ID	Problem Description
Practice-1	Finish the function has999(nums) below to return True if there are at least three items in the list nums that are adjacent and all equal to 9, otherwise return False.
Practice-2	Finish the function sum7(nums) to return the sum of all numbers in the list nums that are not 7 and not immediately after a 7. Return 0 for an empty list or if there is no qualifying number after a 7. There can be multiple 7s in the list.
Mid-Progress Assessment	Finish a function sum_skip2(nums) that takes a list of integers and returns their sum, but whenever two adjacent 2s are found, skip both of them and the number immediately after the pair. Return the sum of the numbers that are not skipped.
Practice-3	Finish the function is_ascending(nums) that returns True if the list contains at least three consecutive numbers that are in ascending order (where equal adjacent values do not count as ascending), and False otherwise. If the list has fewer than 3 numbers in it, return False.
Practice-4	Write a total45(nums) function that takes a list of integers and returns the sum of all items in the list, excluding any numbers that fall within a 4-to-5 sequence (inclusive). If present, there will always be exactly one 4 and one 5, and the 4 will always appear before the 5. A 4-to-5 sequence starts at a 4 and ends at the next 5, and all numbers between them (including the 4 and 5 themselves) are excluded from the sum.

rank tests were applied due to the small sample size. For the mid-progress assessment, we captured code submissions (if any) at the five-minute mark and compared student performance between conditions using a Mann-Whitney U test on the unit test pass rates. To gain a deeper qualitative understanding of students' perceptions, two researchers conducted a reflexive thematic analysis [31]. Each researcher analyzed the interview transcripts and, where necessary, the session videos, interpreting responses and developing codes guided by the research questions. The researchers subsequently came together to compare their codes and interpretations, discussing any differences to arrive at a shared understanding. For example, when exploring what made LadderPuzzles more helpful to students, one researcher initially identified seven points, while the other identified four broader, composite reasons. Following reflexive discussion, one researcher proposed a set of unified themes. The second researcher reviewed the proposed themes and raised no disagreements, resulting in the final set of themes. Students' preferences and attitudes regarding the two types of planning activities were extracted from interview responses and reported accordingly.

Table 7.3: Survey questions used in the LadderPuzzle evaluation study (7-point Likert scale: 1 = "strongly disagree" and 7 = "strongly agree")

Survey Question ID	Question Description
1	The planning activity was useful for completing the corresponding programming problems.
2	I feel prepared to write code after this planning activity.
3	The planning activity improved the quality of my solutions.
4	I would like to do the above planning activity again for future programming problems.

7.3.2 Findings

7.3.2.1 RQ1: Most students rarely planned explicitly, consistent with prior research; when they did, it was typically for harder or larger problems.

When asked about their planning habits prior to this study, more than half of students reported that they rarely or never engaged in explicit planning before writing code (54%). This is consistent with prior research comparing novices and experts, which found that novices spend very little time planning [260]. A smaller group indicated that their planning behavior depended on the difficulty of the problem (23%). For example, P7 described the decision factor as, *"I think, to me, it really depends on what the code is and what the instructions are. So, for example, if it's a longer form project where there's multiple functions, multiple variables, then I think I'd try to plan out."* Only two students reported planning often (15%). One student (P12) did not explicitly discuss this in the interview. These responses suggest that most students had not regularly practiced explicit planning before writing code. Three main themes emerged from the qualitative analysis to explain why students did not engage in explicit planning prior to this study.

Directly coding felt more authentic and instinctive. Four students reported a tendency to view typing code as the only primary problem-solving process. As P4 explained, *"I feel like planning, it feels like coding, but it's not coding. It's like, I really just want to write. I don't want to think about… I don't want to think about this. I just want to actually do it. It (planning) feels like fake coding, almost, I guess."* Such perception led them to *"just want to dive into the problem rather than taking time to think about the exact steps for it"* (P1) and *"just start getting stuff done and seeing what works and what doesn't work"* (P13).

Planning is perceived as time-consuming and impractical. To some students, planning was seen as an inefficient use of time, even though it may in fact reduce the overall time spent on the problem. As P9 explained, *"I just wanted to try something right away instead of take the*

extra time to plan it out. Sometimes coding takes a while … but I think maybe … I would spend less time actually coding if I did put in a little bit of time planning first." Three students acknowledged prioritizing assignment completion over spending time planning. For instance, P7 identified the key deciding factor as *"it's quite dependent on my time."* P2 echoed this, *"it was out of, laziness … (for) mostly assignments, I just wanted to get through them as soon as possible."* For two other students, planning felt difficult to engage with in practice, either when *"not understanding what it's asking of me"* (P11) or when using the coding platform *"I don't really have a lot of comments or anything, so I think it's just a lack of space or organization to write things out, or, I don't necessarily have a notepad next to me"* (P8).

Unawareness of planning as a distinct problem-solving step. Four students indicated limited awareness of solution planning as a distinct and deliberate step in the programming process, with two explicitly noting that it had never been taught in their prior programming courses. Although P3 had completed two programming courses, planning before coding remained an unfamiliar practice, *"in class we generally don't spend a lot of time planning, we just kind of get straight into coding, and that's when I do trial and error. For me, that's the most comfortable for me."* Similarly, for P5, planning had never been introduced as a practice in their programming courses, *"I didn't know planning code was a thing. I guess I wasn't taught in class to plan. I guess it wasn't an exercise that we practiced or we talked about, and I feel like that would have been helpful."*

7.3.2.2 RQ2: LadderPuzzles were rated as significantly more useful and more effective at improving solution quality than the open-ended planning questions.

Students rated the usefulness of each planning activity for solving programming problems on a 7-point Likert scale (1 = strongly disagree, 7 = strongly agree). LadderPuzzles were rated as significantly more useful for solving programming problems than open-ended planning questions ($W = 0.0$[1], $p = .015$, $r = 1.0$). In addition, students found that LadderPuzzles significantly improved the quality of their solutions ($W = 0.0$, $p = .006$, $r = 1.0$). From the qualitative analysis of interview responses, three themes emerged explaining why students found LadderPuzzles more helpful than open-ended planning questions.

Provide a clear sense of direction toward a solution. Nearly half of the students noted that LadderPuzzles were helpful in providing an assurance of being on the right track. As P10 noted, *"I feel like it's much more clear, I feel more assured using this because there's something I know … I'll be going in the right direction with this."* P6 described this sense of certainty as *"it explicitly tells you what you need to do, so you know to make the code exactly towards that."* Beyond confirming

[1] every single difference went in the same direction

direction, LadderPuzzles also helped students verify their initial approach, as P4 explicitly noted, *"I guess that was nice to know that I'm on the right track, or if I'm on the wrong track."*

In contrast, while the open-ended planning prompt helped students *"slow down and kind of understand what I have to do for the program"* (P1), it did not provide the immediate feedback that *LadderPuzzles* offered, which helped students to *"double-check that everything seemed to be in the right order, and fix things that weren't in the right order"* (P1). In addition, although the open-ended planning questions helped students *"organize my own thoughts better"* (P13), it did not bring them closer to a correct solution, with some students still feeling *"in the same spot"* (P13). As a result, students felt the open-ended planning prompts did little to improve their solution quality.

Establish an algorithmic structure to guide code implementation. Although participants were not required to complete the puzzle, almost all (12 of 13 students, 92%) chose to solve the LadderPuzzle to get a correct algorithmic plan. Through this process, students produced a clear algorithmic solution to guide their code implementation. P1 explained such visualized guidance as, *"especially when it's kind of top-to-bottom order, it definitely helps me visualize it in my head more of what I need to do. So I think the order kind of … helped me know what order to put things in."* A documented algorithmic structure provides a considerably more straightforward reference point than the problem description alone and could serve as *"memory anchors for syntax"* (P2), *"it helps you plan it out, and it's easier to refer back to what you wrote rather than rereading the whole question multiple times"* (P8). In addition, this exercise encouraged algorithmic thinking while preserving space for students to work out the implementation themselves. For example, P13 shared, *"(The LadderPuzzle) first got me thinking about it, and then sort of … help me organize what I actually had to do. It didn't just give me the answers, though. I like that it didn't just tell me, I still had to figure it out."*

Manage cognitive demand during problem-solving for novices. Given that solving programming problems remains cognitively challenging for novices, students valued LadderPuzzle for reducing cognitive demand in two main ways. First, because the output of LadderPuzzle is a complete algorithmic solution, it helped students narrow the error-checking space to syntax when writing code. This freed students from shifting between the logic level and the syntax level. As P5 explained, *"It just gave me a lot more confidence or it gave me a lot more comfort, because when I ran my code and something was wrong, it wasn't me thinking, okay, is it my code, or is it my planning? But I know for sure it's my code, because I know the planning can't be wrong."* Second, it broke the complex coding process into manageable chunks at both the algorithmic and syntax levels. As P9 described, *"as long as I can get a good structure of the function, and (even) I don't have the perfect syntax, I think it would still help beginners kind of learn how to think through coding*

problems, because obviously, when you're solving functions, it's a lot of logic and problem solving."

7.3.2.3 RQ3: Students reported comparable preparedness to write code after both planning activities, with students with LadderPuzzle achieving higher mid-progress programming scores, though not significantly.

Survey responses indicated that students reported comparable levels of preparedness to write code after the planning activity in both LadderPuzzle (M = 4.9, Mdn = 5.0, SD = 1.3) and Baseline (M = 5.0, Mdn = 5.0, SD = 1.3), W = 12.0, p = .407, r = 0.33. This is further supported by the statistically comparable performance on the mid-progress assessment task. Specifically, while students who completed LadderPuzzle as the planning activity during practice achieved a higher mean unit test pass rate (M = 25.0%, Mdn = 0.0%, SD = 41.8%) compared to those who completed the Baseline planning question (M = 3.6%, Mdn = 0.0%, SD = 9.5%), the difference did not reach statistical significance (U = 26.0, p = .384). One theme from the qualitative analysis helps explain this result. Of the 12 students who responded to *"What additional support might help you translate a plan into working code?"*, nine (75%) expressed a need for additional syntax support.

Specifically, even when the planning activity helped students arrive at a clear algorithmic solution, syntax remained a bottleneck in producing a complete code solution. The biggest problem was remembering the syntax details. As P3 explained, *"I guess, what to write, or how to format things. I think I really don't remember, I just cannot remember how to write length and stuff."* Similarly, P2 echoed this, *"My biggest problem was that I don't remember almost any of the syntax. So, if they were available in addition to each of these blocks that would have been really helpful."* Another problem was finding the syntax errors when the code was wrong; therefore, P8 wanted a debugger or syntax checker, *"like Grammarly going through and editing, like, oh, you add an extra colon and you didn't realize those kinds of things."*

7.3.2.4 RQ4: Students expressed high willingness to use both activities again and varied in their recommendations, with some favoring one over the other and others suggesting both.

Survey results showed that students expressed high willingness to engage in both planning activities again for future programming problems, with LadderPuzzle receiving slightly higher ratings (M = 6.2, Mdn = 7.0, SD = 1.3) than the Baseline (M = 5.8, Mdn = 6.0, SD = 1.2), though the difference did not reach statistical significance (W = 8.0, p = .164).

Six of the 12 responding students (50%) recommended using LadderPuzzles only, due to their high level of structured and guided support. For example, P2 explained it as *"I would suggest the LadderPuzzle because it is much more faster to think and also easier to plan."* P10 shared a similar

perspective, *"I feel like I'd recommend the LadderPuzzle. It's definitely much more guided (when) having the backbone there."* Four students (33%) suggested a progressive sequence from LadderPuzzle to the open-ended planning prompt as students' expertise level increases. As P13 described this difference, *"Honestly, both of them will be good. Maybe you could just do the ladder puzzles on the earlier questions, so you sort of get warmed up, and then later on, it's just the planning (open-ended planning question)."* P8 highlighted LadderPuzzle's particular value for beginner students, *"I think in the beginning, trying to approach a problem would be kind of difficult. I think having the LadderPuzzle first (can) help you."*

7.4 Discussion

This work proposes a new practice type, LadderPuzzle, that focuses on solution planning for early computing learning. It responds to calls for greater emphasis on higher-order problem-solving skills. The user study showed that LadderPuzzles were rated significantly more useful and effective at improving solution quality than open-ended planning questions. However, students still shared a need for syntax support. In addition, students expressed high willingness to use planning activities, though their recommendations on type sometimes varied. In this section, we discuss these results and pedagogical implications based on our key findings.

7.4.1 More dedicated practice opportunities for higher-order computational skills

Prior to the widespread availability of generative AI tools, traditional introductory programming courses focused heavily on syntax instruction, with practice centered around code writing. Therefore, students often did not have deliberate practice opportunities for higher-order computational skills. Our results (**RQ1**) align with prior research indicating that students tend to bypass the planning process [260], often diving directly into code rather than planning explicitly [165]. AI tools are increasingly capable of handling low-level code generation [19]. As a result, developing practice opportunities that explicitly target higher-order computational skills becomes even more important. This gap may partly explain why both types of planning practice employed in this work were well-received, with students expressing a high willingness to engage with both activities in the future (**RQ4**).

The finding that students perceived the *LadderPuzzles* as significantly more useful and more effective at improving solution quality than open-ended planning questions (**RQ2**) is consistent with prior research highlighting the benefits of visualizing learners' thoughts through structured components [163, 174, 188]. In addition, by embedding planning as a dedicated practice check-

point within the multi-step programming problem-solving process, LadderPuzzle aligns with the segmentation principle [186]. Students' sequential recommendation of starting with LadderPuzzle, transitioning to the open-ended planning question, and eventually progressing to independent problem-solving aligns with the guidance-fading principle [288, 290], which shows that highly structured support is essential for novices but becomes redundant and imposes unnecessary cognitive load as expertise develops. Building on this, future work should investigate the conditions under which LadderPuzzles are most effective, the appropriate point at which students should transition to the open-ended planning question, and when the planning practice should be faded as students move beyond the novice stage.

7.4.2 Connecting higher-order problem-solving to code implementation

While the primary focus of this work is on solution planning, the user study revealed that planning alone did not fully prepare students to write code (**RQ3**). Within the computing education community, the rise of AI has led to ongoing debates around whether and to what extent low-level syntax should continue to be taught [19, 147, 306]. One design implication is to position LadderPuzzle as an algorithm-to-code exercise [117]. In this formulation, upon completing the algorithmic plan, the system automatically provides students with the corresponding code solution, removing the burden of independently implementing the solution. However, recent studies also revealed that computer science achievement can predict vibe coding proficiency [297]. Building on this insight, another design implication is to incorporate dedicated syntax support and instructional guidance within the programming exercise, better equipping students to translate their algorithmic plan into working code.

7.5 Limitations and Future work

This study has several limitations. First, due to time constraints, each participant completed only a single practice session with two LadderPuzzle problems, whereas building familiarity typically requires sustained practice over time. Future work should examine how LadderPuzzles foster more independent and fluent code writing over time, with a longer-term goal of improving students' programming self-efficacy and supporting the development of more robust computational mental models. Second, this work evaluated LadderPuzzles on only a single topic and with students from a single institution. Future studies should examine their effectiveness across a broader range of programming topics and skill levels. Third, only one mid-progress assessment was included in this user study. Although average unit test pass rates suggested better independent

code-writing performance after working with LadderPuzzles than open-ended planning questions, the limited number of participants and assessment questions was insufficient to draw any definitive conclusions. Future work should expand the number of assessment tasks and explore testing LadderPuzzles in a large-scale classroom deployment, enabling longitudinal studies of usage patterns and learning outcomes over time.

7.6 Conclusion

This work proposes LadderPuzzle, a new type of practice that explicitly trains students in solution planning during computing problem-solving. LadderPuzzle was created and validated using a dual-agent LLM framework consisting of a generator and a validator. A within-subjects evaluation study with 13 students showed that most participants had little prior experience with planning before coding. In the evaluation study, comparing LadderPuzzle with an open-ended planning question, results showed that LadderPuzzle was rated as significantly more useful and more effective at improving solution quality than the open-ended planning question. Students also expressed a clear willingness to engage with both types of planning activities in the future, with some type recommendations varying by expertise level: LadderPuzzle for less experienced students and open-ended planning questions for more advanced ones. These findings provide evidence for the value of personalized, puzzle-based planning exercises in early computing education.

Chapter 8

Conclusions and Future Directions

Technology-enhanced learning environments hold strong potential to foster deeper personalization and engagement in student learning. As new technologies continue to develop, exploring how best to harness them toward positive outcomes has become more important than ever. This dissertation investigated this space through a series of studies ranging from traditional pedagogical mechanisms to advanced AI-based architectures. This work drew on established learning sciences theories, explored the technical opportunities afforded by emerging AI, and responded to the new educational needs arising from an increasingly AI-driven learning landscape. This chapter summarizes the contributions of each project and outlines directions for future work.

8.1 Summary of Contributions and Insights

This dissertation advances the design of personalized, engaging learning experiences in technology-enhanced environments. It contributes primarily to the literature in Artificial Intelligence in Education, Computing Education, Human-AI Interaction, and Learning Sciences. I summarize the contributions and insights below.

- **This dissertation contributes theory-informed mechanisms for supporting students as they build programming skills.** Through a series of evaluations examining students' perceptions, interactions, and learning effectiveness, we showed that these mechanisms can make early programming practice more effective and engaging. These investigations apply existing learning sciences theories and pedagogical approaches to computing education, contributing new empirical evidence and practical insights for this learning context. Specifically, in Chapter 3, the finding that students practiced more efficiently with Parsons scaffolding while maintaining posttest performance aligns with cognitive load theory, since offloading syntax generation lowered extraneous cognitive load. In Chapter 4, the finding that students learned more with Select without added mental effort supports

the self-explanation effect. Their preference for Select over Fill aligns with cognitive load theory and the assistance dilemma, as the Fill format likely imposed extraneous load that led to overload. However, some students found the support mechanisms excessive while others found them insufficient, suggesting a single type of mechanism did not always match each student's zone of proximal development.

- **This dissertation contributes designs, techniques, and insights showing how AI can make learning more learner-centered, emphasizing that thoughtful integration is needed for AI to promote learning rather than serve as a shortcut (Chapter 5 and Chapter 7).** Early evidence, including one of the first studies of authentic novice use of AI code generators in intro-level programming [133], showed that when novices are given direct access to AI-generated solutions, they frequently over-rely on them. CodeTailor offers a design that discourages such shortcut use [133] and promotes active engagement [117] by making AI-generated learning artifacts interactive and movable rather than static and plain. In addition, advances in AI for programming have transformed computing education, shifting the focus from lower-order (Chapters 3, 4, and 5) to higher-order computing skills. However, targeted active practice for these higher-order skills remains limited. In Chapter 7, I explored a new practice type (LadderPuzzle) designed to engage students in explicit planning, a higher-order skill, during computing problem-solving. It uses AI to generate and validate planning puzzles to deliver personalized planning practice for each student. Evaluation showed that novices, who rarely planned before coding, valued explicit planning practice and rated the LadderPuzzle as significantly more useful and effective for improving solution quality than an open-ended planning question.

- **As a step toward real-world use, CodeTailor (Chapter 5) has been deployed as a product feature within Runestone Academy[1]. This demonstrates the technique's practical value and its ability to transfer into an authentic platform, and it also reveals the trade-offs involved in moving from research to real-world practice.** In our evaluation studies, we generated personalized puzzles with a common-solution guardrail built from previous students' code, but this approach is not feasible for every problem on an established platform, where such historical data may not exist. Deployment therefore required alternative pathways to preserve puzzle quality without relying on per-problem student data. It also required practical engineering: we developed support for instructors to supply their own API keys, allowing the feature to scale independently to large groups of students without centralized cost or setup. We leave to future work the study of CodeTailor's usability and effectiveness once released to larger populations and extended to

[1] https://landing.runestone.academy/

additional programming languages and diverse learning contexts.

- **This dissertation demonstrates two cases where multiple coordinated AI agents, rather than a single AI, are used as a design choice to deliver engaging, personalized educational experiences.** In the first case, we proposed a four-agent architecture for conversation-based assessment. This design was motivated by the difficulty a single AI faces in playing multiple student-facing roles, such as an expert and a peer, while simultaneously carrying out assessment and collecting evidence in real time. The initial evaluation showed some positive evidence that the agents could deliver personalized assessment to students while maintaining enough evidence for human review. However, it also raised questions for future work about the cost of coordination, such as whether some agents could take on multiple tasks so that the benefits outweigh the risks during complex coordination. The second example used AI agents only on the generation side (behind the scenes). Unlike the assessment setting, these tasks were more straightforward and required less context or history to be shared between agents, which lowered both the risk and cost of coordination.

- **This dissertation responds to a shift in computing education, in which advances in AI for programming have moved the focus from lower-order skills (Chapters 3, 4, and 5) to newly emphasized ones.** LadderPuzzle (Chapter 7) is an example of such a pedagogical innovation, providing targeted practice for these emerging skills and adding to this body of literature. Planning is a commonly recognized skill in problem-solving, but early courses rarely teach it explicitly. One reason is that traditional intro-level courses are already dense with language syntax and concept understanding. Introducing a dedicated planning stage before code writing, and giving students practice at it, can help them arrive at an algorithmic solution before they implement it. My work contributes to this shifting view of what computing education should teach, providing a new, dedicated form of practice.

8.2 Future Directions

Building on the insights of this dissertation, I outline two kinds of future work in this section. The first consists of near-term steps that emerge directly from the studies presented here. The second comprises the broader directions I am excited to explore in the coming years.

8.2.1 Near-Term Directions

Below, I list near-term directions that build on the dissertation as a whole, complementing the future work discussed within each chapter.

- **Characterize the errors students make to guide design loop adaptation.** While this dissertation explored how students engaged with different mechanisms and the learning experiences that resulted, the errors students make when interacting with these mechanisms, and how often those errors recur across students, remain uncharacterized. Analyzing such errors using log data (Chapters 4, 5, 6, and 7) could yield clearer insights for design loop adaptation [8] and guide future personalization efforts tailored not only to what students get right, but also to the errors they make.

- **Understand how time on task relates to learning and engagement.** One direct outcome of adding structured mechanisms and learning steps, or of making students more engaged, is that students might spend more time on the overall learning task. This dissertation (Chapters 3, 4, 5, and 7) did not focus on examining how that time functions as a mediating or moderating factor in learning and perceived experience. It would be valuable to treat time on task as an outcome in its own right and to build models that clarify how it relates to these different measures.

- **Explore the trade-offs of dividing work across multiple agents versus consolidating it across different task types.** While dividing a task among more specialized agents can keep each agent focused, it also risks losing information and increasing the cost of communication among agents. The resulting isolation can be either a drawback, as in the assessment setting (Chapter 6), or a deliberate design feature, as in the planning puzzle generation setting (Chapter 7). Future work should characterize this trade-off more systematically, identifying which properties of a task predict the optimal number of agents and the right degree of coordination between them.

- **Examine the impact of immediate elaborated feedback and of different feedback types on learning and engagement.** One advantage of digital learning over a traditional classroom, with only one teacher for many students, is its ability to deliver immediate feedback. However, when investigating different learning mechanisms, the studies in this dissertation mainly explored a single type of feedback (Chapters 3, 4, 5, and 7), rather than other types such as the feedback used in cognitive tutors. In addition, the effect of feedback itself could be isolated to better understand its role in learning. Finally, the incorporation of feedback in technology-enhanced environments has influenced revisions to the original

ICAP framework's engagement levels. While this dissertation did not focus on this question, future work should consider how established learning frameworks can be updated as AI takes on educational roles with real-time interaction, roles that used to be filled only by human teachers and peers.

- **Investigate why some skills traditionally received less emphasis in instruction.** This dissertation worked on two skills that have traditionally received less attention in classroom instruction: program comprehension and problem-solving planning. However, it did not systematically identify why students lack these skills, which aspects are most difficult for them, or what prevents them from developing those skills. A key next step is to identify these difficulties directly, for example by comparing matched problems in which a plan is provided versus withheld, and to understand what drives students to engage or disengage during learning.

8.2.2 Broader Directions for Future Research

Rapidly advancing technologies, many not initially built for education, pose complex questions for the field. My long-term goal is to build solutions that prepare motivated and independent learners in a transforming world. Building on the insights of this dissertation, I envision several broader research directions to pursue.

8.2.2.1 Novel Tools, Pedagogy, and Professional Training for Emerging Next-Generation Skills

In this increasingly digital world, everyone (students, educators, and workplace professionals alike) needs computing skills, AI literacy, and self-regulation to become creators and evaluators of AI rather than passive users. However, pedagogical tools that explicitly promote these skills are still lacking. My previous work on personalized and active AI-powered learning tools [115, 117, 118] and insights from students' self-directed learning practices [133] have provided me with novel ideas to promote these next-generation skills. I will continue by identifying emerging skills and designing AI tools for active, personalized, and social learning across diverse student populations. In addition, people often hold a narrow view of learning achievement and might avoid the higher-order processes needed for deeper learning. I will examine factors influencing their intentions to develop these skills and explore better practices. By researching the expected new skills, I aim to foster people's mastery of skills and empower them to become confident creators in today's technological world.

8.2.2.2 Human-Centered AI for Data-Driven Learning Decisions that Keep Humans in Control

AI holds the potential to provide valuable insights that enhance decision-making for educational stakeholders. However, its well-known risks, such as bias, discrimination, and lack of transparency, can undermine that potential. I would like to explore the use of interpretable, intuitive data visualizations to communicate AI-driven insights to educational stakeholders. Specifically, I plan to introduce interpretable AI methods to provide explanations and feedback for AI-powered educational decisions based on student data. I will also design visualizations that help students, instructors, and administrators interpret this information and use it to support collective decision-making. These information delivery interfaces will be iteratively refined through human-centered design, tailoring granularity to human progress and engagement data. This line of work aims to strengthen human-AI communication while prioritizing and preserving human agency.

8.2.2.3 Designing and Delivering Multifaceted Personalized Education to Engage Every Student

In addition, pedagogical AI systems, like Google's LearnLM, have the potential to be used as purpose-built AI models for education. On the model side, refining AI models and mechanisms through student interaction and learning science theories can yield actionable improvements. For example, my previous work explored the use of speech-act diagnoses to guide student-AI conversations [115]. I would like to extend this line of work to explore how AI can be refined to serve education better. On the pedagogical mechanism design side, AI can deliver multiple levels of personalization, from student interests to problem-solving strategies. While spanning diverse domains, my work has focused on micro-level personalization [115, 117]. In the future, I plan to explore personalized learning through broader, macro-level dimensions, such as student interests and learning pace. In addition, personalization should be a student-centered, collective effort involving educators, designers, parents, and communities. I aim to make AI-powered personalized learning a shared commitment among stakeholders, not just a singular technical act.

Bibliography

[1] Python module difflib. https://docs.python.org/3/library/difflib.html, 2023. Accessed: 2023-08.

[2] Langchain, 2025.

[3] Siti-Soraya Abdul-Rahman and Benedict du Boulay. Learning programming via worked-examples: Relation of learning styles to cognitive load. Computers in Human Behavior, 30:286–298, 2014.

[4] Hanif Akhtar and Retno Firdiyanti. Test-taking motivation and performance: Do self-report and time-based measures of effort reflect the same aspects of test-taking motivation? Learning and Individual Differences, 2023.

[5] Vincent Aleven and Kenneth R Koedinger. An effective metacognitive strategy: Learning by doing and explaining with a computer-based cognitive tutor. Cognitive science, 26(2):147–179, 2002.

[6] Vincent Aleven, Bruce McLaren, Ido Roll, and Kenneth Koedinger. Toward meta-cognitive tutoring: A model of help seeking with a cognitive tutor. International Journal of Artificial Intelligence in Education, 16(2):101–128, 2006.

[7] Vincent Aleven, Bruce M Mclaren, Jonathan Sewall, and Kenneth R Koedinger. A new paradigm for intelligent tutoring systems: Example-tracing tutors. International Journal of Artificial Intelligence in Education, 19(2):105–154, 2009.

[8] Vincent Aleven, Elizabeth A McLaughlin, R Amos Glenn, and Kenneth R Koedinger. Instruction based on adaptive learning technologies. Handbook of research on learning and instruction, 2, 2016.

[9] Patricia A Alexander. The development of expertise: The journey from acclimation to proficiency. Educational researcher, 32(8):10–14, 2003.

[10] Paolo Antonucci, Christian Estler, Durica Nikolić, Marco Piccioni, and Bertrand Meyer. An incremental hint system for automated programming assignments. In Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education, pages 320–325, 2015.

[11] Robert K Atkinson, Sharon J Derry, Alexander Renkl, and Donald Wortham. Learning from examples: Instructional principles from the worked examples research. Review of educational research, 70(2):181–214, 2000.

[12] Robert K Atkinson, Alexander Renkl, and Mary Margaret Merrill. Transitioning from studying examples to solving problems: Effects of self-explanation prompts and fading worked-out steps. Journal of educational psychology, 95, 2003.

[13] Phil Bagge and Shuchi Grover. Before you program, plan! In Shuchi Grover, editor, Computer Science in K–12: An A to Z Handbook on Teaching Programming. Edfinity, 2020.

[14] David Baidoo-Anu and Leticia Owusu Ansah. Education in the era of generative artificial intelligence (AI): Understanding the potential benefits of chatgpt in promoting teaching and learning. Journal of AI, 7(1):52–62, 2023.

[15] Frederick T Bail, Shuqiang Zhang, and Gary T Tachiyama. Effects of a self-regulated learning course on the academic performance and graduation rate of college students in an academic support program. Journal of college reading and learning, 39(1):54–73, 2008.

[16] Ryan SJD Baker, AMJA De Carvalho, Jay Raspat, Vincent Aleven, Albert T Corbett, and Kenneth R Koedinger. Educational software features that encourage and discourage "gaming the system". In Proceedings of the 14th international conference on artificial intelligence in education, pages 475–482, 2009.

[17] Albert Bandura. Self-efficacy: toward a unifying theory of behavioral change. Psychological review, 84(2), 1977.

[18] Susan M Barnett and Stephen J Ceci. When and where do we apply what we learn?: A taxonomy for far transfer. Psychological bulletin, 128(4), 2002.

[19] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 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, pages 500–506, 2023.

[20] Brian R Belland. Scaffolding: Definition, current debates, and future directions. In Handbook of research on educational communications and technology, pages 505–518. Springer, 2013.

[21] Brian R Belland, ChanMin Kim, and Michael J Hannafin. A framework for designing scaffolds that improve motivation and cognition. Educational psychologist, 48(4):243–270, 2013.

[22] Klara Benda, Amy Bruckman, and Mark Guzdial. When life and learning do not fit: Challenges of workload and communication in introductory computer science online. ACM Transactions on Computing Education (TOCE), 12(4):1–38, 2012.

[23] Matthew L Bernacki, Meghan J Greene, and Nikki G Lobczowski. A systematic review of research on personalized learning: Personalized by whom, to what, how, and for what purpose (s)? Educational Psychology Review, 33(4):1675–1715, 2021.

[24] João Henrique Berssanette and Antonio Carlos de Francisco. Cognitive load theory in the context of teaching and learning computer programming: A systematic literature review. IEEE Transactions on Education, 65(3):440–449, 2022.

[25] Kirsten Berthold, Tessa HS Eysink, and Alexander Renkl. Assisting self-explanation prompts are more effective than open prompts when learning with multiple representations. Instructional Science, 37:345–363, 2009.

[26] Marei Beukman, Björn Rudzewitz, Daniela Verratti Souto, Mingjing Zhu, and Xiaobin Chen. Developing an AI-powered system that promotes interactive learning and engagement. In International Conference on Higher Education Learning Methodologies and Technologies Online, pages 74–87. Springer, 2024.

[27] John B Biggs and Kevin F Collis. Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome). Academic Press, 2014.

[28] Kiran Bisra, Qing Liu, John C Nesbit, Farimah Salimi, and Philip H Winne. Inducing self-explanation: A meta-analysis. Educational Psychology Review, 30:703–725, 2018.

[29] Benjamin S Bloom. The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. Educational researcher, 13(6):4–16, 1984.

[30] Charles C Bonwell and James A Eison. Active learning: Creating excitement in the classroom. 1991 ASHE-ERIC higher education reports. ERIC, 1991.

[31] Virginia Braun and Victoria Clarke. Reflecting on reflexive thematic analysis. Qualitative research in sport, exercise and health, 11(4):589–597, 2019.

[32] Joachim C Brunstein and Cornelia Glaser. Testing a path-analytic mediation model of how self-regulated writing strategies improve fourth graders' composition skills: A randomized controlled trial. Journal of educational psychology, 103(4), 2011.

[33] Trong Hieu Bui and S. M. F. D. Syed Mustapha. Automated data-driven hint generation in intelligent tutoring systems for code-writing: On the road of future research. International Journal of Emerging Technologies in Learning (iJET), 3(9):174–189, 2018.

[34] Chi-Min Chan, Weize Chen, Yusheng Su, Jianxuan Yu, Wei Xue, Shanghang Zhang, Jie Fu, and Zhiyuan Liu. Chateval: Towards better llm-based evaluators through multi-agent debate. In International conference on learning representations, volume 2024, pages 9079–9093, 2024.

[35] Paul Chandler and John Sweller. The split-attention effect as a factor in the design of instruction. British Journal of Educational Psychology, 62(2):233–246, 1992.

[36] Ching-Huei Chen. An adaptive scaffolding e-learning system for middle school students' physics learning. Australasian Journal of Educational Technology, 30(3), 2014.

[37] Eason Chen, Ray Huang, Han-Shin Chen, Yuen-Hsien Tseng, and Liang-Yi Li. Gptutor: a chatgpt-powered programming tool for code explanation. In International Conference on Artificial Intelligence in Education, pages 321–327. Springer, 2023.

[38] Michelene TH Chi, Joshua Adams, Emily B Bogusch, Christiana Bruchok, Seokmin Kang, Matthew Lancaster, Roy Levy, Na Li, Katherine L McEldoon, Glenda S Stump, et al. Translating the ICAP theory of cognitive engagement into practice. Cognitive science, 42(6):1777–1832, 2018.

[39] Michelene TH Chi, Miriam Bassok, Matthew W Lewis, Peter Reimann, and Robert Glaser. Self-explanations: How students study and use examples in learning to solve problems. Cognitive science, 13(2):145–182, 1989.

[40] Michelene TH Chi and Nicole S Boucher. Applying the ICAP framework to improve classroom learning. In their own words: What scholars and teachers want you to know about why and how to apply the science of learning in your academic setting, pages 94–110, 2023.

[41] Michelene TH Chi and Ruth Wylie. The ICAP framework: Linking cognitive engagement to active learning outcomes. Educational psychologist, 49(4):219–243, 2014.

[42] Veronica Chiarelli, Nadia Markova, and Kasia Muldner. Evaluating the utility of notional machine representations to help novices learn to code trace. In Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1, pages 314–328, 2023.

[43] Thomas KF Chiu, Zubair Ahmad, Murod Ismailov, and Ismaila Temitayo Sanusi. What are artificial intelligence literacy and competency? A comprehensive framework to support them. Computers and Education Open, 6, 2024.

[44] Sasithorn Chookaew, Patcharin Panjaburee, Dechawut Wanichsan, and Parames Laosinchai. A personalized e-learning environment to promote student's conceptual learning on basic computer programming. Procedia-Social and Behavioral Sciences, 116:815–819, 2014.

[45] Irene-Angelica Chounta, Bruce M McLaren, Patricia L Albacete, Pamela W Jordan, and Sandra Katz. Modeling the zone of proximal development with a computational approach. EDM, 2017:56–57, 2017.

[46] Ruth C Clark and Richard E Mayer. E-learning and the science of instruction: Proven guidelines for consumers and designers of multimedia learning. john Wiley & sons, 2023.

[47] Jacob Cohen. A coefficient of agreement for nominal scales. Educational and psychological measurement, 20(1):37–46, 1960.

[48] Jacob Cohen. Statistical power analysis for the behavioral sciences. routledge, 2013.

[49] Cristina Conati and Kurt VanLehn. Teaching meta-cognitive skills: Implementation and evaluation of a tutoring system to guide self-explanation while learning from examples. In Artificial intelligence in education, pages 297–304. IOS Press, 1999.

[50] Albert T Corbett and John R Anderson. Knowledge tracing: Modeling the acquisition of procedural knowledge. User modeling and user-adapted interaction, 4(4):253–278, 1994.

[51] Kathryn Cunningham, Barbara J Ericson, Rahul Agrawal Bejarano, and Mark Guzdial. Avoiding the turing tarpit: Learning conversational programming by starting from code's purpose. In Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, pages 1–15, 2021.

[52] Jing Dai, Xiaoqing Gu, and Jiawen Zhu. Personalized recommendation in the adaptive learning system: The role of adaptive testing technology. Journal of Educational Computing Research, 61(3):523–545, 2023.

[53] Linda Darling-Hammond. The Flat World and Education: How America's Commitment to Equity Will Determine Our Future. Multicultural Education Series. Teachers College Press, New York, 2010.

[54] Elizabeth A Davis and Naomi Miyake. Explorations of scaffolding in complex classroom systems. In Scaffolding, pages 265–272. Psychology Press, 2018.

[55] Judy Davis-Dorsey, Steven M Ross, and Gary R Morrison. The role of rewording and context personalization in the solving of mathematical word problems. Journal of educational Psychology, 83, 1991.

[56] Michael De Raadt, Richard Watson, and Mark Toleman. Teaching and assessing programming strategies explicitly. In Proceedings of the 11th Australasian Computing Education Conference (ACE 2009), volume 95, pages 45–54, 2009.

[57] Adrienne Decker, Lauren E Margulieux, and Briana B Morrison. Using the SOLO taxonomy to understand subgoal labels effect in CS1. In Proceedings of the 2019 ACM Conference on International Computing Education Research, pages 209–217, 2019.

[58] Fadi P Deek, Murray Turoff, and James A McHugh. A common model for problem solving and program development. IEEE Transactions on Education, 42(4):331–336, 1999.

[59] Andre del Carpio Gutierrez, Paul Denny, and Andrew Luxton-Reilly. Automating personalized Parsons problems with customized contexts and concepts. In Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1, pages 688–694. 2024.

[60] Mehmet Arif Demirtaş, Claire Zheng, Max Fowler, and Kathryn Cunningham. Generating planning feedback for open-ended programming exercises with llms. In International Conference on Artificial Intelligence in Education, pages 33–48. Springer, 2025.

[61] Paul Denny, Viraj Kumar, and Nasser Giacaman. Conversing with copilot: Exploring prompt engineering for solving CS1 problems using natural language. In Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1, pages 1136–1142, 2023.

[62] Paul Denny, Viraj Kumar, Stephen MacNeil, James Prather, and Juho Leinonen. Probing the unknown: Exploring student interactions with probeable problems at scale in introductory programming. In Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 1, pages 618–624, 2025.

[63] Paul Denny, Juho Leinonen, James Prather, Andrew Luxton-Reilly, Thezyrie Amarouche, Brett A Becker, and Brent N Reeves. 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, pages 296–302, 2024.

[64] Paul Denny, Andrew Luxton-Reilly, and Beth Simon. Evaluating a new exam question: Parsons problems. In Proceedings of the fourth international workshop on computing education research, pages 113–124, 2008.

[65] Paul Denny, James Prather, Brett A. Becker, Catherine Mooney, John Homer, Zachary C Albrecht, and Garrett B. Powell. On designing programming error messages for novices: Readability and its constituent factors. In Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery.

[66] Indrani Dey, Dana Gnesdilow, Riley Smith, ChanMin Kim, Rebecca J Passonneau, and Sadhana Puntambekar. Factors influencing students' perceptions of automated feedback and their impact on revision. In Proceedings of the 19th International Conference of the Learning Sciences-ICLS 2025. International Society of the Learning Sciences, 2025.

[67] Samuel Dodson, Ido Roll, Matthew Fong, Dongwook Yoon, Negar M Harandi, and Sidney Fels. An active viewing framework for video-based learning. In Proceedings of the fifth annual ACM conference on Learning@ Scale, pages 1–4, 2018.

[68] Yuemeng Du, Andrew Luxton-Reilly, and Paul Denny. A review of research on Parsons problems. In Proceedings of the Twenty-Second Australasian Computing Education Conference, pages 195–202, 2020.

[69] Rodrigo Duran, Jan-Mikael Rybicki, Juha Sorva, and Arto Hellas. Exploring the value of student self-evaluation in introductory programming. In Proceedings of the 2019 ACM Conference on International Computing Education Research, pages 121–130, 2019.

[70] Rodrigo Duran, Albina Zavgorodniaia, and Juha Sorva. Cognitive load theory in computing education research: A review. ACM Transactions on Computing Education (TOCE), 22(4):1–27, 2022.

[71] Stephen H Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In Proceedings of the 35th SIGCSE technical symposium on Computer science education, pages 26–30, 2004.

[72] Barbara Ericson, Austin McCall, and Kathryn Cunningham. Investigating the affect and effect of adaptive Parsons problems. In Proceedings of the 19th Koli Calling International Conference on Computing Education Research, 2019.

[73] Barbara J Ericson, Paul Denny, James Prather, Rodrigo Duran, Arto Hellas, Juho Leinonen, Craig S Miller, Briana B Morrison, Janice L Pearce, and Susan H Rodger. Parsons problems and beyond: Systematic literature review and empirical study designs. Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education, pages 191–234, 2022.

[74] Barbara J Ericson, James D Foley, and Jochen Rick. Evaluating the efficiency and effectiveness of adaptive Parsons problems. In Proceedings of the 2018 ACM Conference on International Computing Education Research, pages 60–68, 2018.

[75] Barbara J Ericson, Lauren E Margulieux, and Jochen Rick. Solving Parsons problems versus fixing and writing code. In Proceedings of the 17th Koli Calling International Conference on Computing Education Research, pages 20–29, 2017.

[76] Barbara J Ericson and Bradley N Miller. Runestone: A platform for free, on-line, and interactive ebooks. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education, pages 1012–1018, 2020.

[77] Kürşat Erümit, Hasan Karal, Güven Şahin, Dilara Aksoy, Ayşegül Aksoy, and Ali Benzer. A model suggested for programming teaching: Programming in seven steps. Education and Science, 44(197):155–183, 2019.

[78] Geela Venise Firmalo Fabic, Antonija Mitrovic, and Kourosh Neshatian. Investigating the effectiveness of menu-based self-explanation prompts in a mobile Python tutor. In Artificial Intelligence in Education: 18th International Conference, AIED 2017, Wuhan, China, June 28–July 1, 2017, Proceedings 18, pages 498–501. Springer, 2017.

[79] Geela Venise Firmalo Fabic, Antonija Mitrovic, and Kourosh Neshatian. Evaluation of Parsons problems with menu-based self-explanation prompts in a mobile python tutor. International Journal of Artificial Intelligence in Education, 29:507–535, 2019.

[80] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. CodeBERT: A pre-trained model for programming and natural languages. In Findings of the association for computational linguistics: EMNLP 2020, pages 1536–1547, 2020.

[81] Kasra Ferdowsi, Ruanqianqian Huang, Michael B James, Nadia Polikarpova, and Sorin Lerner. Validating ai-generated code with live programming. In Proceedings of the 2024 CHI conference on human factors in computing systems, pages 1–8, 2024.

[82] Bridgid Finn. Measuring motivation in low-stakes assessments. ETS Research Report Series, 2015(2):1–17, 2015.

[83] Lawrence M Fisher. Booming enrollments. Communications of the ACM, 59(7):17–18, 2016.

[84] Kathi Fisler. The recurring rainfall problem. In Proceedings of the tenth annual conference on International computing education research, pages 35–42, 2014.

[85] Carol M Forsyth, Christine Luce, Diego Zapata-Rivera, G Tanner Jackson, Keelan Evanini, and Youngsoon So. Evaluating english language learners' conversations: Man vs. machine. Computer Assisted Language Learning, 32(4):398–417, 2019.

[86] Carol M Forsyth, Diego Zapata-Rivera, Edith Aurora Graf, and Yang Jiang. Complex conversations: Llm vs. knowledge engineering conversation-based assessment. Journal of educational data mining, 2024.

[87] Max Fowler, David H Smith IV, Mohammed Hassan, Seth Poulsen, Matthew West, and Craig Zilles. Reevaluating the relationship between explaining, tracing, and writing skills in CS1 in a replication study. Computer Science Education, 32(3):355–383, 2022.

[88] Diana Franklin, Paul Denny, David A Gonzalez-Maldonado, and Minh Tran. Generative AI in computer science education: Challenges and opportunities. Cambridge University Press, 2025.

[89] Manuel B Garcia. Teaching and learning computer programming using chatgpt: A rapid review of literature amid the rise of generative ai technologies. Education and information technologies, 30(12):16721–16745, 2025.

[90] Rita Garcia, Katrina Falkner, and Rebecca Vivian. Scaffolding the design process using Parsons problems. In Proceedings of the 18th Koli Calling International Conference on Computing Education Research, pages 1–2, 2018.

[91] Kathy Gibbs and Wendi Beamish. Differentiated instruction: A programming tool for inclusion. In Inclusive theory and practice in special education, pages 174–191. IGI Global, 2020.

[92] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. Overcode: Visualizing variation in student solutions to programming problems at scale. ACM Transactions on Computer-Human Interaction (TOCHI), 22(2):1–35, 2015.

[93] Joseph A Gliem and Rosemary R Gliem. Calculating, interpreting, and reporting cronbach's alpha reliability coefficient for likert-type scales. Midwest Research to Practice Conference in Adult, Continuing, and Community Education, 2003.

[94] Janice D Gobert, Ryan S Baker, and Michael B Wixon. Operationalizing and detecting disengagement within online science microworlds. Educational Psychologist, 50(1):43–57, 2015.

[95] Steve Goodman and Hank Bohanon. A framework for supporting all students: One-size-fits-all no longer works in schools. American School Board Journal, 2018.

[96] Jamie Gorson and Eleanor O'Rourke. Why do CS1 students think they're bad at programming? Investigating self-efficacy and self-assessments at three universities. In Proceedings of the 2020 ACM Conference on International Computing Education Research, pages 170–181, 2020.

[97] Arthur C Graesser. Conversations with AutoTutor help students learn. International Journal of Artificial Intelligence in Education, 26:124–132, 2016.

[98] Arthur C Graesser, Kurt VanLehn, Carolyn P Rosé, Pamela W Jordan, and Derek Harter. Intelligent tutoring systems with conversational dialogue. AI magazine, 22(4):39–39, 2001.

[99] Alessandro Guida, Hubert Tardieu, and Serge Nicolas. The personalisation method applied to a working memory task: Evidence of long-term working memory effects. European journal of cognitive psychology, 21(6):862–896, 2009.

[100] Gerald L Gutek. A history of the western educational experience. Waveland Press, 2022.

[101] Mark Joseph Guzdial. Emile: Software-realized scaffolding for science learners programming in mixed media. University of Michigan, 1993.

[102] Georgiana Haldeman, Peter Ohmann, and Paul Denny. Systematically thinking about the complexity of code structuring exercises at introductory level. In Proceedings of the 57th ACM Technical Symposium on Computer Science Education V. 1, pages 442–448, 2026.

[103] Jieun Han, Haneul Yoo, Yoonsu Kim, Junho Myung, Minsun Kim, Hyunseung Lim, Juho Kim, Tak Yeon Lee, Hwajung Hong, So-Yeon Ahn, and Alice Oh. Recipe: How to integrate chatgpt into efl writing education. In Proceedings of the Tenth ACM Conference on Learning @ Scale, L@S '23, page 416–420, New York, NY, USA, 2023. Association for Computing Machinery.

[104] Anton Havnes, Kari Smith, Olga Dysthe, and Kristine Ludvigsen. Formative assessment and feedback: Making learning visible. Studies in educational evaluation, 38(1):21–27, 2012.

[105] Carl C Haynes and Barbara J Ericson. Problem-solving efficiency and cognitive load for adaptive Parsons problems vs. writing the equivalent code. In Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, pages 1–15, 2021.

[106] Junda He, Christoph Treude, and David Lo. Llm-based multi-agent systems for software engineering: Literature review, vision and the road ahead. ACM Transactions on Software Engineering and Methodology, 2025.

[107] Juha Helminen, Petri Ihantola, Ville Karavirta, and Satu Alaoutinen. How do students solve Parsons programming problems?–Execution-based vs. line-based feedback. In 2013 Learning and Teaching in Computing and Engineering, pages 55–61. IEEE, 2013.

[108] Andrew D Hilton, Genevieve M Lipp, and Susan H Rodger. Translation from problem to code in seven steps. In Proceedings of the ACM Conference on Global Computing Education, pages 78–84, 2019.

[109] Bobby Hoffman and Gregory Schraw. Conceptions of efficiency: Applications in learning and problem solving. Educational Psychologist, 45(1):1–14, 2010.

[110] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In International Conference on Learning Representations, volume 2024, pages 23247–23275, 2024.

[111] Song Chang Hong and Yoon Soo Park. An analysis of the relationship between self-study, private tutoring, and self-efficacy on self-regulated learning. KEDI journal of educational policy, 9(1), 2012.

[112] Xinying Hou, Barbara Jane Ericson, and Xu Wang. Using adaptive Parsons problems to scaffold write-code problems. In Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1, pages 15–26, 2022.

[113] Xinying Hou, Barbara Jane Ericson, and Xu Wang. Parsons problems to scaffold code writing: Impact on performance and problem-solving efficiency. In Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 2, pages 665–665, 2023.

[114] Xinying Hou, Barbara Jane Ericson, and Xu Wang. Understanding the effects of using Parsons problems to scaffold code writing for students with varying cs self-efficacy levels. In Proceedings of the 23rd Koli Calling International Conference on Computing Education Research, 2023.

[115] Xinying Hou, Carol Forsyth, Jessica Andrews-Todd, James Rice, Zhiqiang Cai, Yang Jiang, Diego Zapata-Rivera, and Art Graesser. An llm-enhanced multi-agent architecture for conversation-based assessment. In International Conference on Artificial Intelligence in Education, pages 119–134. Springer, 2025.

[116] Xinying Hou, Evie Katmanivong, Xu Wang, and Barbara J. Ericson. Enhance code understanding through prompted self-explanation with mixed-up code puzzles: Novice preferences and outcomes. In Proceedings of the 20th International Conference of the Learning Sciences-ICLS 2026. International Society of the Learning Sciences, 2026.

[117] Xinying Hou, Zihan Wu, Xu Wang, and Barbara J. Ericson. CodeTailor: LLM-powered personalized Parsons puzzles for engaging support while learning programming. In Proceedings of the Eleventh ACM Conference on Learning @ Scale, L@S '24, page 51–62, New York, NY, USA, 2024. Association for Computing Machinery.

[118] Xinying Hou, Zihan Wu, Xu Wang, and Barbara J Ericson. Do students recognize personalization? Comparing personalized vs. common-solution Parsons code puzzles as problem-solving scaffolding. In Proceedings of the 20th International Conference of the Learning Sciences-ICLS 2026. International Society of the Learning Sciences, 2026.

[119] Jonathan Huang, Chris Piech, Andy Nguyen, and Leonidas Guibas. Syntactic and functional variability of a million code submissions in a machine learning mooc. In AIED 2013 Workshops Proceedings Volume, volume 25, 2013.

[120] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. ACM Transactions on Information Systems, 43(2):1–55, 2025.

[121] Mark Huxham, Fiona Campbell, and Jenny Westwood. Oral versus written assessments: A test of student performance and attitudes. Assessment & Evaluation in Higher Education, 37(1):125–136, 2012.

[122] Petri Ihantola and Ville Karavirta. Two-dimensional parson's puzzles: The concept, tools, and first observations. Journal of Information Technology Education. Innovations in Practice, 10, 2011.

[123] Alastair Irons and Sam Elkington. Enhancing learning through formative assessment and feedback. Routledge, 2022.

[124] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, et al. Fostering program comprehension in novice programmers-learning activities and learning trajectories. In Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education, pages 27–52. 2019.

[125] G Tanner Jackson, Katherine E Castellano, Debra Brockway, and Blair Lehman. Improving the measurement of cognitive skills through automated conversations. Journal of Research on Technology in Education, 50(3):226–240, 2018.

[126] G Tanner Jackson and Diego Zapata-Rivera. Conversation-based assessment. The Wiley Handbook of Cognition and Assessment: Frameworks, Methodologies, and Applications, pages 563–579, 2016.

[127] Bowen Jiang, Yangxinyu Xie, Xiaomeng Wang, Weijie J Su, Camillo Jose Taylor, and Tanwi Mallick. Multi-modal and multi-agent systems meet rationality: A survey. In ICML 2024 Workshop on LLMs and Cognition, 2024.

[128] Yuan-Hao Jiang, Tian-Yi Liu, Xinlin Zhuang, Hanglei Hu, Ruijia Li, and Rui Jia. Enhancing educational practices with multi-agent systems: A review. Enhancing educational practices: Strategies for assessing and improving learning outcomes, pages 47–65, 2024.

[129] Hyoungwook Jin, Seonghee Lee, Hyungyu Shin, and Juho Kim. Teach AI how to code: Using large language models as teachable agents for programming education. In Proceedings of the CHI Conference on Human Factors in Computing Systems, pages 1–28, 2024.

[130] Hyun Joo, Jinju Lee, and Dongsik Kim. Advancing the design of self-explanation prompts for complex problem-solving. International Journal of Learning, Teaching and Educational Research, 19(11):88–108, 2020.

[131] Firuz Kamalov, David Santandreu Calonge, and Ikhlaas Gurrib. New era of artificial intelligence in education: Towards a sustainable multifaceted revolution. Sustainability, 15(16), 2023.

[132] Shyam Sundar Kannan, Vishnunandan LN Venkatesh, and Byung-Cheol Min. Smart-llm: Smart multi-agent robot task planning using large language models. In 2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 12140–12147. IEEE, 2024.

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

[134] Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, Austin Zachary Henley, Paul Denny, Michelle Craig, and Tovi Grossman. 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, pages 1–20, 2024.

[135] Nidhi Khattri, Kevin W Riley, and Michael B Kane. Students at risk in poor, rural areas: A review of the research. Technical report, Pelavin Research Institute, 1997. ERIC Document Reproduction Service.

[136] Hassan Khosravi, Kirsty Kitto, and Joseph Jay Williams. Ripple: A crowdsourced adaptive platform for recommendation of learning activities. Journal of Learning Analytics, 6(3):91–105, Dec. 2019.

[137] Minchi C Kim and Michael J Hannafin. Scaffolding problem solving in technology-enhanced learning environments (teles): Bridging research and theory with practice. Computers & Education, 56(2):403–417, 2011.

[138] Kenneth R Koedinger and Vincent Aleven. Exploring the assistance dilemma in experiments with cognitive tutors. Educational Psychology Review, 19:239–264, 2007.

[139] Klaus Krippendorff. Reliability in content analysis: Some common misconceptions and recommendations. Human communication research, 30(3):411–433, 2004.

[140] Klaus Krippendorff. Computing krippendorff's alpha-reliability. 2011.

[141] Heng-Yu Ku and Howard J Sullivan. Personalization of mathematics word problems in taiwan. Educational Technology Research and Development, 48(3):49–60, 2000.

[142] Amruth N Kumar. 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, pages 183–188, 2013.

[143] Amruth N Kumar. An evaluation of self-explanation in a programming tutor. In Intelligent Tutoring Systems: 12th International Conference, ITS 2014, Honolulu, HI, USA, June 5-9, 2014. Proceedings 12, pages 248–253. Springer, 2014.

[144] Sandeep Kaur Kuttal, Bali Ong, Kate Kwasny, and Peter Robe. Trade-offs for substituting a human with an agent in a pair programming context: the good, the bad, and the ugly. In Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, pages 1–20, 2021.

[145] Kyungbin Kwon, Christiana D Kumalasari, and Jane L Howland. Self-explanation prompts on problem-solving performance in an interactive learning environment. Journal of Interactive Online Learning, 10(2), 2011.

[146] Thomas K Landauer, Danielle S McNamara, Simon Dennis, and Walter Kintsch. Handbook of latent semantic analysis. Psychology Press, 2007.

[147] Sam Lau, Kianoosh Boroojeni, Harry Keeling, and Jenn Marroquin. Barriers that programming instructors face while performing emergency pedagogical design to shape student-ai interactions with generative ai tools. In Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems, pages 1–15, 2026.

[148] Sam Lau and Philip Guo. 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, pages 106–121, 2023.

[149] Jeong-Kyu Lee. Educational thoughts of aristotle and confucius. The Journal of Educational Thought (JET)/Revue De La Pensée Éducative, pages 161–180, 2001.

[150] Sunbok Lee, Man Kit Lei, and Gene H Brody. Constructing confidence intervals for effect size measures of an indirect effect. Multivariate behavioral research, 50(6):600–613, 2015.

[151] Blair A Lehman and Diego Zapata-Rivera. Student emotions in conversation-based assessments. IEEE Transactions on Learning Technologies, 11(1):41–53, 2018.

[152] Teemu Lehtinen, Aleksi Lukkarinen, and Lassi Haaranen. Students struggle to explain their own program code. In Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1, pages 206–212, 2021.

[153] Colleen M Lewis, Ken Yasuhara, and Ruth E Anderson. Deciding to major in computer science: a grounded theory of students' self-assessment of ability. In Proceedings of the seventh international workshop on Computing education research, pages 3–10, 2011.

[154] Mark Liffiton, Brad E Sheese, Jaromir Savelka, and Paul Denny. Codehelp: 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, pages 1–11, 2023.

[155] Chi-Jen Lin and Husni Mubarok. Promoting young learners' computational thinking with ai-based robots and the ICAP model in blended learning contexts. Interactive Learning Environments, 32(10):7242–7256, 2024.

[156] Lijia Lin and Robert K Atkinson. Enhancing learning from different visualizations by self-explanation prompts. Journal of Educational Computing Research, 49(1):83–110, 2013.

[157] Lijia Lin, Xin Lin, Xiaofang Zhang, and Paul Ginns. The personalized learning by interest effect on interest, cognitive load, retention, and transfer: A meta-analysis. Educational Psychology Review, 36(3), 2024.

[158] Alex Lishinski and Aman Yadav. 28 motivation, attitudes, and dispositions. Cambridge University Press, 2019.

[159] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L Whalley, and Christine Prasad. Not seeing the forest for the trees: Novice programmers and the solo taxonomy. ACM SIGCSE Bulletin, 38(3):118–122, 2006.

[160] Huaiya Liu, Yuyue Zhang, and Jiyou Jia. The design of guiding and adaptive prompts for intelligent tutoring systems and its effect on students' mathematics learning. IEEE Transactions on Learning Technologies, 17:1353–1363, 2024.

[161] Lei Liu, Jonathan Steinberg, Farah Qureshi, Isaac Bejar, and Fred Yan. Conversation-based assessments: An innovative approach to measure scientific reasoning. Bulletin of the IEEE Technical Committee on Learning Technology, 18(1):10–13, 2016.

[162] Shuo Liu, Zeyu Liang, Xueguang Lyu, and Christopher Amato. Llm collaboration with multi-agent reinforcement learning. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 40, pages 32150–32158, 2026.

[163] Zhicheng Liu and John Stasko. Mental models, visual reasoning and interaction in information visualization: A top-down perspective. IEEE transactions on visualization and computer graphics, 16(6):999–1008, 2010.

[164] Evanfiya Logacheva, Arto Hellas, James Prather, Sami Sarsa, and Juho Leinonen. Evaluating contextually personalized programming exercises created with generative ai. In Proceedings of the 2024 ACM Conference on International Computing Education Research-Volume 1, pages 95–113, 2024.

[165] Dastyni Loksa and Amy J Ko. The role of self-regulation in programming problem solving process and success. In Proceedings of the 2016 ACM conference on international computing education research, pages 83–91, 2016.

[166] Dastyni Loksa, Amy J Ko, Will Jernigan, Alannah Oleson, Christopher J Mendez, and Margaret M Burnett. Programming, problem solving, and self-awareness: Effects of explicit guidance. In Proceedings of the 2016 CHI conference on human factors in computing systems, pages 1449–1461, 2016.

[167] Doug Lombardi, Thomas F Shipley, Chemistry Team Engineering Team Geography Team Geoscience Team Astronomy Team, Biology Team, and Physics Team. The curious construct of active learning. Psychological Science in the Public Interest, 22(1):8–43, 2021.

[168] Tania Lombrozo. The structure and function of explanations. Trends in cognitive sciences, 10(10):464–470, 2006.

[169] James E Los, Sara E Witmer, and Cary J Roseth. Exploring motivational factors associated with test-taking effort among middle school students. School Psychology Review, 51(5):555–573, 2022.

[170] Xinyi Lu and Xu Wang. Generative students: Using llm-simulated student profiles to support question item evaluation. In Proceedings of the Eleventh ACM Conference on Learning@ Scale, pages 16–27, 2024.

[171] Rosemary Luckin and Benedict du Boulay. Reflections on the ecolab and the zone of proximal development. International Journal of Artificial Intelligence in Education, 26(1):416–430, 2016.

[172] Rosemary Luckin, Benedict du Boulay, et al. Ecolab: The development and evaluation of a vygotskian design framework. International journal of artificial Intelligence in Education, 10(2):198–220, 1999.

[173] Qianou Ma, Weirui Peng, Chenyang Yang, Hua Shen, Ken Koedinger, and Tongshuang Wu. What should we engineer in prompts? Training humans in requirement-driven llm use. ACM Transactions on Computer-Human Interaction, 32(4):1–27, 2025.

[174] Shuai Ma, Junling Wang, Yuanhao Zhang, Xiaojuan Ma, and April Yi Wang. Dbox: Scaffolding algorithmic programming learning through learner-llm co-decomposition. In Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems, pages 1–20, 2025.

[175] David P MacKinnon, Amanda J Fairchild, and Matthew S Fritz. Mediation analysis. Annu. Rev. Psychol., 58(1):593–614, 2007.

[176] David P MacKinnon, Chondra M Lockwood, and Jason Williams. Confidence limits for the indirect effect: Distribution of the product and resampling methods. Multivariate behavioral research, 39(1):99–128, 2004.

[177] Stephen MacNeil, James Prather, Rahad Arman Nabid, Sebastian Gutierrez, Silas Carvalho, Saimon Shrestha, Paul Denny, Brent N Reeves, Juho Leinonen, and Rachel Louise Rossetti. Fostering responsible AI use through negative expertise: a contextualized autocompletion quiz. In Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 1, pages 326–332, 2025.

[178] Joyce Mahon, Brian Mac Namee, and Brett A Becker. Guidelines for the evolving role of generative ai in introductory programming based on emerging practice. In Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1, pages 10–16. 2024.

[179] Lauren Margulieux and Richard Catrambone. Improving programming instruction with subgoal labeled instructional text. In Proceedings of the Annual Meeting of the Cognitive Science Society, volume 36, 2014.

[180] Lauren E Margulieux and Richard Catrambone. Finding the best types of guidance for constructing self-explanations of subgoals in programming. Journal of the Learning Sciences, 28(1):108–151, 2019.

[181] Samiha Marwan, Anay Dombe, and Thomas W Price. Unproductive help-seeking in programming: What it is and how to address it. In Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, pages 54–60, 2020.

[182] Samiha Marwan, Ge Gao, Susan Fisk, Thomas W Price, and Tiffany Barnes. Adaptive immediate feedback can improve novice programming engagement and intention to persist in computer science. In Proceedings of the 2020 ACM conference on international computing education research, pages 194–203, 2020.

[183] Samiha Marwan, Nicholas Lytle, Joseph Jay Williams, and Thomas Price. The impact of adding textual explanations to next-step hints in a novice programming environment. In Proceedings of the 2019 ACM conference on innovation and technology in computer science education, pages 520–526, 2019.

[184] Janet Maybin, Neil Mercer, and Barry Stierer. Scaffolding learning in the classroom. Thinking voices: The work of the national oracy project, 186, 1992.

[185] Richard E Mayer. The psychology of how novices learn computer programming. ACM Computing Surveys (CSUR), 13(1):121–141, 1981.

[186] Richard E Mayer and Celeste Pilegard. Principles for managing essential processing in multimedia learning: Segmenting, pretraining, and modality principles. The Cambridge handbook of multimedia learning, 13(316-44), 2005.

[187] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In Working group reports from ITiCSE on Innovation and technology in computer science education, pages 125–180. 2001.

[188] Siné JP McDougall, Martin B Curry, and Oscar de Bruijn. The effects of visual information on users' mental models: An evaluation of pathfinder analysis as a measure of icon usability. International journal of cognitive ergonomics, 5(1):59–84, 2001.

[189] Mary L McHugh. Interrater reliability: the kappa statistic. Biochemia medica, 22(3):276–282, 2012.

[190] Danielle S McNamara, Arthur C Graesser, Philip M McCarthy, and Zhiqiang Cai. Automated evaluation of text and discourse with Coh-Metrix. Cambridge University Press, 2014.

[191] Matthew B. Miles, A. Michael Huberman, and Johnny Saldaña. Qualitative Data Analysis: A Methods Sourcebook. SAGE Publications, 3 edition, 2014.

[192] Robert J Mislevy, Russell G Almond, and Janice F Lukas. A brief introduction to evidence-centered design. ETS Research Report Series, 2003(1):i–29, 2003.

[193] Robert J Mislevy and Michelle M Riconscente. Evidence-centered assessment design. In Handbook of test development, pages 75–104. Routledge, 2011.

[194] Uday Mittal, Siva Sai, Vinay Chamola, and Devika Sangwan. A comprehensive review on generative AI for education. IEEE Access, 12:142733–142759, 2024.

[195] Irene T Miura. The relationship of computer self-efficacy expectations to computer interest and course enrollment in college. Sex roles, 16(5):303–311, 1987.

[196] Chuan-Yu Mo, Chengliang Wang, Jian Dai, and Peiqi Jin. Video playback speed influence on learning effect from the perspective of personalized adaptive learning: A study based on cognitive load theory. Frontiers in Psychology, 13, 2022.

[197] Chiara Montuori, Filippo Gambarota, Gianmarco Altoé, and Barbara Arfé. The cognitive effects of computational thinking: A systematic review and meta-analytic study. Computers & Education, 210, 2024.

[198] Roxana Moreno and Babette Park. Cognitive Load Theory: Historical Development and Relation to Other Theories, page 9–28. Cambridge University Press, 2010.

[199] Briana B Morrison, Lauren E Margulieux, and Mark Guzdial. Subgoals, context, and worked examples in learning computing problem solving. In Proceedings of the eleventh annual international conference on international computing education research, pages 21–29, 2015.

[200] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. Reading between the lines: Modeling user behavior and costs in ai-assisted programming. In Proceedings of the 2024 CHI conference on human factors in computing systems, pages 1–16, 2024.

[201] Laurie Murphy, Renée McCauley, and Sue Fitzgerald. "Explain in plain English" questions: Implications for teaching. In Proceedings of the 43rd ACM technical symposium on Computer Science Education, pages 385–390, 2012.

[202] Tom Murray and Ivon Arroyo. Toward measuring and maintaining the zone of proximal development in adaptive instructional systems. In International conference on intelligent tutoring systems, pages 749–758. Springer, 2002.

[203] Mir Murtaza, Yamna Ahmed, Jawwad Ahmed Shamsi, Fahad Sherwani, and Mariam Usman. Ai-based personalized e-learning systems: Issues, challenges, and solutions. IEEE access, 10:81323–81342, 2022.

[204] Yair Neuman and Baruch Schwarz. Is self-explanation while solving problems helpful? The case of analogical problem-solving. British Journal of Educational Psychology, 68(1):15–24, 1998.

[205] Huy A Nguyen, Xinying Hou, Hayden Stec, Sarah Di, John Stamper, and Bruce M McLaren. Examining the learning benefits of different types of prompted self-explanation in a decimal learning game. In International Conference on Artificial Intelligence in Education, pages 681–687. Springer, 2023.

[206] Manh Hung Nguyen, Victor-Alexandru Pădurean, Alkis Gotovos, Sebastian Tschiatschek, and Adish Singla. Synthesizing high-quality programming tasks with llm-based expert and student agents. In International Conference on Artificial Intelligence in Education, pages 77–91. Springer, 2025.

[207] Laura R Novick. Representational transfer in problem solving. Psychological Science, 1(2):128–132, 1990.

[208] Esko Nuutila, Seppo Törmä, and Lauri Malmi. PBL and computer programming—the seven steps method with adaptations. Computer science education, 15(2):123–142, 2005.

[209] Benjamin D Nye, Dillon Mee, and Mark G Core. Generative large language models for dialog-based tutoring: An early consideration of opportunities and concerns. In LLM@ AIED, pages 78–88, 2023.

[210] Jo Ann Oravec. Artificial intelligence implications for academic cheating: Expanding the dimensions of responsible human-ai collaboration with chatgpt. Journal of Interactive Learning Research, 34(2):213–237, 2023.

[211] Fred GWC Paas. Training strategies for attaining transfer of problem-solving skill in statistics: a cognitive-load approach. Journal of educational psychology, 84(4), 1992.

[212] Fred GWC Paas and Jeroen JG Van Merriënboer. Instructional control of cognitive load in the training of complex cognitive tasks. Educational psychology review, 6(4):351–371, 1994.

[213] Frank Pajares and M David Miller. Role of self-efficacy and self-concept beliefs in mathematical problem solving: A path analysis. Journal of educational psychology, 86(2), 1994.

[214] Frank Pajares and Gio Valiante. Influence of self-efficacy on elementary students' writing. The Journal of Educational Research, 90(6):353–360, 1997.

[215] Ernesto Panadero. A review of self-regulated learning: Six models and four directions for research. Frontiers in psychology, 8, 2017.

[216] Linda Parschau, Lena Fleig, Milena Koring, Daniela Lange, Nina Knoll, Ralf Schwarzer, and Sonia Lippke. Positive experience, self-efficacy, and action control predict physical activity changes: A moderated mediation analysis. British journal of health psychology, 18(2):395–406, 2013.

[217] Dale Parsons and Patricia Haden. 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, pages 157–163, 2006.

[218] Elizabeth Patitsas, Michelle Craig, and Steve Easterbrook. Comparing and contrasting different algorithms leads to increased student learning. In Proceedings of the ninth annual international ACM conference on International computing education research, pages 145–152, 2013.

[219] Mrigank Pawagi and Viraj Kumar. Probeable problems for beginner-level programming-with-ai contests. In Proceedings of the 2024 ACM Conference on International Computing Education Research-Volume 1, pages 166–176, 2024.

[220] Roy D Pea. Logo programming and problem solving. Technical Report 12, Center for Children and Technology, Bank Street College of Education, 1983.

[221] Roy D Pea. The social and technological dimensions of scaffolding and related theoretical concepts for learning, education, and human activity. The journal of the learning sciences, 13(3):423–451, 2004.

[222] Roy D Pea and D Midian Kurland. Logo programming and the development of planning skills. Technical Report 16, 1984.

[223] Roy D Pea and D Midian Kurland. On the cognitive effects of learning computer programming. New ideas in psychology, 2(2):137–168, 1984.

[224] Judea Pearl. Direct and indirect effects. In Probabilistic and causal inference: the works of Judea Pearl, pages 373–392. 2022.

[225] Hongchao Peng, Shanshan Ma, and Jonathan Michael Spector. Personalized adaptive learning: an emerging pedagogical approach enabled by a smart learning environment. Smart Learning Environments, 6(1):1–14, 2019.

[226] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. Cognitive psychology, 19(3):295–341, 1987.

[227] Ivica Pesovski, Ricardo Santos, Roberto Henriques, and Vladimir Trajkovik. Generative ai for customizable learning experiences. Sustainability, 16(7), 2024.

[228] Paul R Pintrich and Barbara Schrauben. Students' motivational beliefs and their cognitive engagement in classroom academic tasks. Student perceptions in the classroom, 7(1):149–183, 1992.

[229] Vikki Pollard and Christine Armatas. Feedback is integral: Using a revised ICAP framework to achieve active learning in an asynchronous online course. Online Learning, 29(3):236–254, 2025.

[230] Alexander Pollatsek and Arnold D Well. On the use of counterbalanced designs in cognitive research: a suggestion for a better and more powerful analysis. Journal of Experimental psychology: Learning, memory, and Cognition, 21, 1995.

[231] George Polya. How to solve it. Princeton Press, 1957.

[232] Leo Porter and Daniel Zingaro. Learn AI-assisted Python programming: with github Copilot and ChatGPT. Manning, 2023.

[233] Seth Poulsen, Mahesh Viswanathan, Geoffrey L Herman, and Matthew West. Proof blocks: autogradable scaffolding activities for learning to write proofs. In Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education Vol. 1, pages 428–434, 2022.

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

[235] James Prather, John Homer, Paul Denny, Brett A Becker, John Marsden, and Garrett Powell. Scaffolding task planning using abstract Parsons problems. In IFIP World Conference on Computers in Education, pages 591–602. Springer, 2022.

[236] James Prather, Raymond Pettit, Brett A Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci. First things first: Providing metacognitive scaffolding for interpreting problem prompts. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education, pages 531–537, 2019.

[237] James Prather, Brent N Reeves, Paul Denny, Brett A Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. "It's weird that it knows what I want": Usability and interactions with copilot for novice programmers. ACM Transactions on Computer-Human Interaction, 31(1):1–31, 2023.

[238] James Prather, Brent N Reeves, Juho Leinonen, Stephen MacNeil, Arisoa S Randrianasolo, Brett A Becker, Bailey Kimmel, Jared Wright, and Ben Briggs. 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, pages 469–486, 2024.

[239] Kristopher J Preacher and Andrew F Hayes. SPSS and SAS procedures for estimating indirect effects in simple mediation models. Behavior research methods, instruments, & computers, 36:717–731, 2004.

[240] Thomas W Price, Yihuan Dong, Rui Zhi, Benjamin Paaßen, Nicholas Lytle, Veronica Cateté, and Tiffany Barnes. A comparison of the quality of data-driven programming hint generation algorithms. International Journal of Artificial Intelligence in Education, 29(3):368–395, 2019.

[241] Thomas W Price, Zhongxiu Liu, Veronica Cateté, and Tiffany Barnes. Factors influencing students' help-seeking behavior while programming with human and computer tutors. In Proceedings of the 2017 ACM Conference on international computing education research, pages 127–135, 2017.

[242] Thomas W Price, Joseph Jay Williams, Jaemarie Solyst, and Samiha Marwan. Engaging students with instructor solutions in online programming homework. In Proceedings of the 2020 CHI conference on human factors in computing systems, pages 1–7, 2020.

[243] Thomas W Price, Rui Zhi, and Tiffany Barnes. Hint generation under uncertainty: The effect of hint quality on help-seeking behavior. In International conference on artificial intelligence in education, pages 311–322. Springer, 2017.

[244] Michael Prince. Does active learning work? A review of the research. Journal of engineering education, 93(3):223–231, 2004.

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

[246] Victor Qiu, Liam Parker, Kaitlin Riegel, Nasser Giacaman, Paul Denny, Stephen MacNeil, and James Prather. One line at a time: Scaffolding reflective code evaluation through structured ai assistance. In Proceedings of the 28th Australasian Computing Education Conference, pages 102–111, 2026.

[247] Rolf Reber, Elizabeth A Canning, and Judith M Harackiewicz. Personalized education to increase interest. Current directions in psychological science, 27(6):449–454, 2018.

[248] Peter Reimann and C Neubert. The role of self-explanation in learning to use a spreadsheet through examples. Journal of Computer Assisted Learning, 16(4):316–325, 2000.

[249] Alexander Renkl. Learning from worked-out examples: A study on individual differences. Cognitive science, 21(1):1–29, 1997.

[250] Alexander Renkl. Worked-out examples: Instructional explanations support learning by self-explanations. Learning and instruction, 12(5):529–556, 2002.

[251] Alexander Renkl, Robin Stark, Hans Gruber, and Heinz Mandl. Learning from worked-out examples: The effects of example variability and elicited self-explanations. Contemporary educational psychology, 23(1):90–108, 1998.

[252] Peter J Rich, Garrett Egan, and Jordan Ellsworth. A framework for decomposition in computational thinking. In Proceedings of the 2019 ACM conference on innovation and technology in computer science education, pages 416–421, 2019.

[253] Robert S Rist. Knowledge creation and retrieval in program design: A comparison of novice and intermediate student programmers. Human-Computer Interaction, 6(1):1–46, 1991.

[254] Robert S Rist. Learning to program: Schema creation, application, and evaluation. Computer Science Education Research, pages 175–197, 2004.

[255] Elijah Rivera, Alexander Steinmaurer, Kathi Fisler, and Shriram Krishnamurthi. Iterative student program planning using transformer-driven feedback. In Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1, pages 45–51, 2024.

[256] Kelly Rivers. Automated data-driven hint generation for learning programming. PhD thesis, Carnegie Mellon University, 2017.

[257] Kelly Rivers and Kenneth R Koedinger. Automating hint generation with solution space path construction. In International Conference on Intelligent Tutoring Systems, pages 329–339. Springer, 2014.

[258] Kelly Rivers and Kenneth R Koedinger. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. International Journal of Artificial Intelligence in Education, 27(1):37–64, 2017.

[259] Steven B Robbins, Kristy Lauver, Huy Le, Daniel Davis, Ronelle Langley, and Aaron Carlstrom. Do psychosocial and study skill factors predict college outcomes? A meta-analysis. Psychological bulletin, 130(2), 2004.

[260] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming: A review and discussion. Computer science education, 13(2):137–172, 2003.

[261] Anthony V. Robins. Novice programmers and introductory programming. In Sally A. Fincher and Anthony V. Robins, editors, The Cambridge Handbook of Computing Education Research, chapter 12. Cambridge University Press, 2019.

[262] Jürgen Rudolph, Shannon Tan, and Samson Tan. War of the chatbots: Bard, bing chat, chatgpt, ernie and beyond. the new ai gold rush and its impact on higher education. Journal of Applied Learning and Teaching, 6(1), 2023.

[263] Juan Pablo Saenz and Luigi De Russis. On how novices approach programming exercises before and during coding. In Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems, CHI EA '22, New York, NY, USA, 2022. Association for Computing Machinery.

[264] Adrian Salguero, William G Griswold, Christine Alvarado, and Leo Porter. Understanding sources of student struggle in early computer science courses. In Proceedings of the 17th ACM Conference on International Computing Education Research, pages 319–333, 2021.

[265] Debra Sanders and Dorette Sugg Welk. Strategies to scaffold student learning: Applying vygotsky's zone of proximal development. Nurse educator, 30(5):203–207, 2005.

[266] Rémi Saurel, Franck Silvestre, Jean-Baptiste Raclet, and Emmanuel Lescure. Maestro: Multi-agent educational system for tutoring and recommendation orchestration. In European Conference on Technology Enhanced Learning, pages 380–385. Springer, 2025.

[267] R Keith Sawyer. The Cambridge handbook of the learning sciences. Cambridge University Press, 2005.

[268] Linda J Sax, Kathleen J Lehman, and Christina Zavala. Examining the enrollment growth: Non-cs majors in CS1 courses. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, pages 513–518, 2017.

[269] Wolfgang Schnotz and Christian Kürschner. A reconsideration of cognitive load theory. Educational psychology review, 19(4):469–508, 2007.

[270] Carsten Schulte. Block model: an educational model of program comprehension as a tool for a scholarly approach to teaching. In Proceedings of the fourth international workshop on computing education research, pages 149–160, 2008.

[271] Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James H Paterson. An introduction to program comprehension for computer science educators. Proceedings of the 2010 ITiCSE working group reports, pages 65–86, 2010.

[272] Silke Schworm and Alexander Renkl. Learning by solved example problems: Instructional explanations reduce self-explanation activity. In Proceedings of the Twenty-Fourth Annual Conference of the Cognitive Science Society, pages 816–821. Routledge, 2019.

[273] Sue Sentance and Andrew Csizmadia. Computing in the curriculum: Challenges and strategies from a teacher's perspective. Education and Information Technologies, 22(2):469–495, 2017.

[274] Karim Shabani, Mohamad Khatib, and Saman Ebadi. Vygotsky's zone of proximal development: Instructional implications and teachers' professional development. English language teaching, 3(4):237–248, 2010.

[275] Dale Shaffer, Wendy Doube, and Juhani Tuovinen. Applying cognitive load theory to computer science education. In PPIG, volume 1, pages 333–346, 2003.

[276] Kathleen M Sheehan, Michael Flor, Diane Napolitano, and Chaitanya Ramineni. Using TextEvaluator® to quantify sources of linguistic complexity in textbooks targeted at first-grade readers over the past half century. ETS Research Report Series, 2015(2):1–17, 2015.

[277] Atikah Shemshack, Kinshuk, and Jonathan Michael Spector. A comprehensive analysis of personalized learning components. Journal of Computers in Education, 8(4):485–503, 2021.

[278] Yuhong Shi, Kun Yu, Yifei Dong, and Fang Chen. Large language models in education: a systematic review of empirical applications, benefits, and challenges. Computers and Education: Artificial Intelligence, 2025.

[279] Leonardo Silva, António José Mendes, Anabela Gomes, and Gabriel Fortes Cavalcanti de Macêdo. Regulation of learning interventions in programming education: A systematic literature review and guideline proposition. In Proceedings of the 52nd ACM Technical Symposium on Computer Science Education, pages 647–653, 2021.

[280] Simon and Susan Snowdon. Explaining program code: giving students the answer helps-but only just. In Proceedings of the seventh international workshop on Computing education research, pages 93–100, 2011.

[281] Daevesh Singh and Ramkumar Rajendran. Cognitive engagement as a predictor of learning gain in python programming. Smart Learning Environments, 11(1), 2024.

[282] Alexander Skulmowski and Kate Man Xu. Understanding cognitive load in digital and online learning: A new perspective on extraneous cognitive load. Educational psychology review, 34(1):171–196, 2022.

[283] Adele Smolansky, Andrew Cram, Corina Raduescu, Sandris Zeivots, Elaine Huber, and Rene F Kizilcec. Educator and student perspectives on the impact of generative ai on assessments in higher education. In Proceedings of the Tenth ACM Conference on Learning@ Scale, pages 378–382, 2023.

[284] John Stamper, Ruiwei Xiao, and Xinying Hou. Enhancing llm-based feedback: Insights from intelligent tutoring systems and the learning sciences. In International Conference on Artificial Intelligence in Education, pages 32–43. Springer, 2024.

[285] Jerry Chih-Yuan Sun and Kelly Yi-Chuan Hsu. A smart eye-tracking feedback scaffolding approach to improving students' learning self-efficacy and performance in a C programming course. Computers in Human Behavior, 95:66–72, 2019.

[286] John Sweller. Element interactivity and intrinsic, extraneous, and germane cognitive load. Educational psychology review, 22:123–138, 2010.

[287] John Sweller. Cognitive load theory. In Psychology of learning and motivation, volume 55, pages 37–76. Elsevier, 2011.

[288] John Sweller, Paul Ayres, and Slava Kalyuga. The guidance fading effect. In Cognitive load theory, pages 171–182. Springer, 2011.

[289] John Sweller and Paul Chandler. Why some material is difficult to learn. Cognition and instruction, 12(3):185–233, 1994.

[290] John Sweller, Jeroen JG Van Merriënboer, and Fred Paas. Cognitive architecture and instructional design: 20 years later. Educational psychology review, 31(2):261–292, 2019.

[291] Keith S Taber. Scaffolding learning: Principles for effective teaching and the design of classroom resources. Effective teaching and learning: Perspectives, strategies and implementation, pages 1–43, 2018.

[292] Charlene Tan. Confucius, volume 20. Bloomsbury Academic, London, 2013.

[293] Kuok Ho Daniel Tang. Student-centered approach in teaching and learning: What does it really mean? Acta Pedagogia Asiana, 2(2):72–83, 2023.

[294] Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. Magis: Llm-based multi-agent framework for github issue resolution. Advances in Neural Information Processing Systems, 37:51963–51993, 2024.

[295] Catherine Taylor. Assessment for measurement or standards: The peril and promise of large-scale assessment reform. American Educational Research Journal, 31(2):231–262, 1994.

[296] Paul Thagard. Analogy, explanation, and education. Journal of Research in science Teaching, 29(6):537–544, 1992.

[297] Sverrir Thorgeirsson, Theo B Weidmann, and Zhendong Su. Computer science achievement and writing skills predict vibe coding proficiency. In Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems, pages 1–17, 2026.

[298] Sue Timmis, Patricia Broadfoot, Rosamund Sutherland, and Alison Oldfield. Rethinking assessment in a digital age: Opportunities, challenges and risks. British Educational Research Journal, 42(3):454–476, 2016.

[299] Cris Tovani. So what do they really know?: Assessment that informs teaching and learning. Routledge, 2023.

[300] Dwayne Towell and Brent Reeves. From walls to steps: Using online automatic homework checking tools to improve learning in introductory programming courses. ACET Journal of Computer Education and Research, 2010.

[301] Yune Tran. Computational thinking equity in elementary classrooms: What third-grade students know and can do. Journal of Educational Computing Research, 57(1):3–31, 2019.

[302] Chun-Yen Tsai. Improving students' understanding of basic programming concepts through visual programming language: The role of self-efficacy. Computers in Human Behavior, 95:224–232, 2019.

[303] Serap Uğur and Gülfem Gürses. Adult learners' experiences of learning with interactive video according to technological literacy levels in the context of andragogy and ICAP framework. Interactive Learning Environments, 34(2):956–971, 2026.

[304] Esther Ulitzsch, Christiane Penk, Matthias von Davier, and Steffi Pohl. Model meets reality: Validating a new behavioral measure for test-taking effort. Educational Assessment, 26(2):104–124, 2021.

[305] Yildiz Uzun, Wannapon Suraworachet, Qi Zhou, Andrea Gauthier, and Mutlu Cukurova. Engagement with analytics feedback and its relationship to self-regulated learning competence and course performance. International Journal of Educational Technology in Higher Education, 22(1), 2025.

[306] Annapurna Vadaparty, David H Smith IV, Samvrit Srinath, Mounika Padala, Christine Alvarado, Jamie Gorson Benario, Leo Porter, and Daniel Zingaro. Evaluating CS1-LLM: Integrating llms and examining student outcomes in an introductory computer science course. In Proceedings of the 28th Australasian Computing Education Conference, pages 32–41, 2026.

[307] Eva Van de Weijer-Bergsma and Sanne HG Van der Ven. Why and for whom does personalizing math problems enhance performance? Testing the mediation of enjoyment and cognitive load at different ability levels. Learning and Individual Differences, 87, 2021.

[308] Jeroen JG Van Merriënboer. Strategies for programming instruction in high school: Program completion vs. program generation. Journal of educational computing research, 6(3):265–285, 1990.

[309] Jeroen JG Van Merriënboer and Marcel BM De Croock. Strategies for computer-based programming instruction: Program completion vs. program generation. Journal of Educational Computing Research, 8(3):365–394, 1992.

[310] Jeroen JG Van Merriënboer, Paul A Kirschner, and Liesbeth Kester. Taking the load off a learner's mind: Instructional design for complex learning. Educational psychologist, 38(1):5–13, 2003.

[311] Jeroen JG Van Merrienboer and Hein PM Krammer. Instructional strategies and tactics for the design of introductory computer programming courses in high school. Instructional Science, 16(3):251–285, 1987.

[312] Jeroen JG Van Merrienboer and John Sweller. Cognitive load theory and complex learning: Recent developments and future directions. Educational psychology review, 17:147–177, 2005.

[313] Rani Van Schoors, Sohum M Bhatt, Jan Elen, Annelies Raes, Wim Van den Noortgate, and Fien Depaepe. Design and development of a digital personalized learning track: Bridging the gap between textual and visual programming. International Journal of Designs for Learning, 15(1):74–95, 2024.

[314] Irina Verenikina. Understanding scaffolding and the ZPD in educational research. 2003.

[315] Camilo Vieira, Alejandra J Magana, Anindya Roy, and Michael L Falk. Student explanations in the context of computational science and engineering education. Cognition and Instruction, 37(2):201–231, 2019.

[316] Arto Vihavainen, Craig S Miller, and Amber Settle. Benefits of self-explanation in introductory programming. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education, pages 284–289, 2015.

[317] Lev Semenovich Vygotsky and Michael Cole. Mind in society: Development of higher psychological processes. Harvard university press, 1978.

[318] Sierra Wang, John Mitchell, and Chris Piech. A large scale RCT on effective error messages in CS1. In Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1, pages 1395–1401, 2024.

[319] Tianfu Wang, Yi Zhan, Jianxun Lian, Zhengyu Hu, Nicholas Jing Yuan, Qi Zhang, Xing Xie, and Hui Xiong. Llm-powered multi-agent framework for goal-oriented learning in intelligent tutoring system. In Companion Proceedings of the ACM on Web Conference 2025, pages 510–519, 2025.

[320] Xu Wang, Diyi Yang, Miaomiao Wen, Kenneth Koedinger, and Carolyn P Rosé. Investigating how student's cognitive behavior in mooc discussion forums affect learning gains. In Proceedings of the 8th International Conference on Educational Data Mining, 2015.

[321] Elliot Washor and Charles Mojkowski. Student disengagement: It's deeper than you think. Phi Delta Kappan, 95(8):8–10, 2014.

[322] Christopher Watson and Frederick WB Li. Failure rates in introductory programming revisited. In Proceedings of the 2014 conference on Innovation & technology in computer science education, pages 39–44, 2014.

[323] Field M Watts and Solaire A Finkenstaedt-Quinn. The current state of methods for establishing reliability in qualitative chemistry education research articles. Chemistry Education Research and Practice, 22(3):565–578, 2021.

[324] Nathaniel Weinman, Armando Fox, and Marti A Hearst. Improving instruction of programming patterns with faded Parsons problems. In Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, pages 1–4, 2021.

[325] Christina Wekerle, Martin Daumiller, Stefan Janke, Oliver Dickhäuser, Markus Dresel, and Ingo Kollar. Putting ICAP to the test: how technology-enhanced learning activities are related to cognitive and affective-motivational learning outcomes in higher education. Scientific Reports, 14(1), 2024.

[326] Gordon Wells. Dialogic inquiry. Cambridge, MA: Cambridge Press, 1999.

[327] Jacqueline Whalley and Nadia Kasto. How difficult are novice code writing tasks? A software metrics approach. In Proceedings of the Sixteenth Australasian Computing Education Conference-Volume 148, pages 105–112, 2014.

[328] Eric Wiebe, Laurie Ann Williams, Kai Yang, and Carol S Miller. Computer science attitude survey. Technical report, North Carolina State University. Dept. of Computer Science, 2003.

[329] Susan Wiedenbeck. Factors affecting the success of non-majors in learning to program. In Proceedings of the first international workshop on Computing education research, pages 13–24, 2005.

[330] Joseph B Wiggins, Joseph F Grafsgaard, Kristy Elizabeth Boyer, Eric N Wiebe, and James C Lester. Do you think you can? The influence of student self-efficacy on the effectiveness of tutorial dialogue for computer science. International Journal of Artificial Intelligence in Education, 27(1):130–153, 2017.

[331] Dylan Wiliam. What is assessment for learning? Studies in educational evaluation, 37(1):3–14, 2011.

[332] Kimberly Wilson and Anupama Narayan. Relationships among individual task self-efficacy, self-regulated learning strategy use and academic performance in a computer-supported collaborative learning environment. Educational Psychology, 36(2):236–253, 2016.

[333] Jeannette M Wing. Computational thinking. Communications of the ACM, 49(3):33–35, 2006.

[334] Steven L Wise and Christine E DeMars. Low examinee effort in low-stakes assessment: Problems and potential solutions. Educational assessment, 10(1):1–17, 2005.

[335] Steven L Wise and Lingyun Gao. A general approach to measuring test-taking effort on computer-based tests. Applied Measurement in Education, 30(4):343–354, 2017.

[336] Regina MF Wong, Michael J Lawson, and John Keeves. The effects of self-explanation training on students' problem solving in high-school mathematics. Learning and Instruction, 12(2):233–262, 2002.

[337] David Wood, Jerome S Bruner, and Gail Ross. The role of tutoring in problem solving. Journal of child psychology and psychiatry, 17(2):89–100, 1976.

[338] Zihan Wu and Barbara J Ericson. Sql puzzles: Evaluating micro Parsons problems with different feedbacks as practice for novices. In Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems, pages 1–15, 2024.

[339] Ruth Wylie and Michelene T. H. Chi. The self-explanation principle in multimedia learning. In Richard E. Mayer, editor, The Cambridge Handbook of Multimedia Learning, chapter 17. Cambridge University Press, 2014.

[340] Ruiwei Xiao, Xinying Hou, and John Stamper. Exploring how multiple levels of gpt-generated programming hints support or disappoint novices. In Extended Abstracts of the CHI Conference on Human Factors in Computing Systems, pages 1–10, 2024.

[341] Benjamin Xie, Dastyni Loksa, Greg L Nelson, Matthew J Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Amy J Ko. A theory of instruction for introductory programming skills. Computer Science Education, 29(2-3):205–253, 2019.

[342] Yi Yang, Yitong Ma, Hao Feng, Yiming Cheng, and Zhu Han. Minimizing hallucinations and communication costs: Adversarial debate and voting mechanisms in llm-based multi-agents. Applied Sciences, 15(7), 2025.

[343] Ethan Yazzie-Mintz. Leading for engagement. Principal Leadership, 10(7):54–58, 2010.

[344] Seyma Yildirim-Erbasli and Okan Bulut. Innovating assessment with conversational agents: A technology-enhanced approach to formative assessments. In 2023 IEEE International Conference on Advanced Learning Technologies (ICALT), pages 331–335. IEEE, 2023.

[345] Seyma N Yildirim-Erbasli and Okan Bulut. Conversation-based assessment: A novel approach to boosting test-taking effort in digital formative assessment. Computers and Education: Artificial Intelligence, 4, 2023.

[346] Seyma N Yildirim-Erbasli and Okan Bulut. Designing predictive models for early prediction of studentsâ€™ test-taking engagement in computerized formative assessments. Journal of Applied Testing Technology, pages 34–47, 2023.

[347] Seyma N Yildirim-Erbasli, Okan Bulut, Carrie Demmans Epp, and Ying Cui. Conversation-based assessments in education: Design, implementation, and cognitive walkthroughs for usability testing. Journal of Educational Technology Systems, 52(1):27–51, 2023.

[348] Seyma Nur Yildirim-Erbasli and Okan Bulut. The impact of students' test-taking effort on growth estimates in low-stakes educational assessments. Educational Research and Evaluation, 26(7-8):368–386, 2020.

[349] Diego Zapata-Rivera and Carol M Forsyth. Learner modeling in conversation-based assessment. In International Conference on Human-Computer Interaction, pages 73–83. Springer, 2022.

[350] Diego Zapata-Rivera, CM Forsyth, A Graf, and Yang Jiang. Designing and evaluating evidence-centered-design-based conversations for assessment with llms. In Proceedings of EDM 2024 workshop: Leveraging large language models for next generation educational technologies, 2024.

[351] Diego Zapata-Rivera, Lei Liu, Lei Chen, Jiangang Hao, and Alina A von Davier. Assessing science inquiry skills in an immersive, conversation-based scenario. Big data and learning analytics in higher education: Current theory and practice, pages 237–252, 2017.

[352] Ling Zhang, James D Basham, and Sohyun Yang. Understanding the implementation of personalized learning: A research synthesis. Educational research review, 31, 2020.

[353] Na Zhang, Jinghan Xu, Ran Wei, and Yifang Wang. From scaffolding to transfer: The impact of ai-powered support on children's narrative skills. Learning and Instruction, 104, 2026.

[354] Jianing Zhao, Peng Gao, Jiannong Cao, Zhiyuan Wen, Chen Chen, Jianing Yin, Ruosong Yang, and Bo Yuan. Codeedu: A multi-agent collaborative platform for personalized coding education. In International Conference on Behavioural and Social Computing, pages 261–269. Springer, 2025.

[355] Rui Zhi, Min Chi, Tiffany Barnes, and Thomas W Price. Evaluating the effectiveness of Parsons problems for block-based programming. In Proceedings of the 2019 ACM Conference on International Computing Education Research, pages 51–59, 2019.

[356] Rui Zhi, Thomas W Price, Samiha Marwan, Alexandra Milliken, Tiffany Barnes, and Min Chi. Exploring the impact of worked examples in a novice programming environment. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education, pages 98–104, 2019.

[357] Yangtao Zhou, Hua Chu, Yongxiang Chen, Ziwen Wang, Jiacheng Liu, Jianan Li, Yueying Feng, Xiangming Li, Zihan Han, and Qingshan Li. Knowledge starts with practice: Knowledge-aware exercise generative recommendation with adaptive multi-agent cooperation. In The Thirty-ninth Annual Conference on Neural Information Processing Systems, 2025.

[358] Bo Zhu, Kien Tsong Chau, and Nur Azlina Mohamed Mokmin. Optimizing cognitive load and learning adaptability with adaptive microlearning for in-service personnel. Scientific Reports, 14(1), 2024.

[359] Barry J Zimmerman. Becoming a self-regulated learner: Which are the key subprocesses? Contemporary educational psychology, 11(4):307–313, 1986.

[360] Barry J Zimmerman. Self-regulated learning and academic achievement: An overview. Educational psychologist, 25(1):3–17, 1990.

[361] Barry J Zimmerman. Developing self-fulfilling cycles of academic regulation: An analysis of exemplary instructional models. In Dale H Schunk and Barry J Zimmerman, editors, Self-Regulated Learning: From Teaching to Self-Reflective Practice. Guilford Press, 1998.

[362] Barry J Zimmerman. Attaining self-regulation: A social cognitive perspective. In Handbook of self-regulation, pages 13–39. Elsevier, 2000.

[363] Barry J Zimmerman. Becoming a self-regulated learner: An overview. Theory into practice, 41(2):64–70, 2002.

[364] Barry J Zimmerman and Manuel Martinez-Pons. Student differences in self-regulated learning: Relating grade, sex, and giftedness to self-efficacy and strategy use. Journal of educational Psychology, 82, 1990.

[365] Barry J Zimmerman and Adam R Moylan. Self-regulation: Where metacognition and motivation intersect. In Handbook of metacognition in education, pages 299–315. Routledge, 2009.

[366] Barry J Zimmerman and Dale H Schunk. Self-regulated learning and performance: An introduction and an overview. In Handbook of self-regulation of learning and performance, pages 15–26. Routledge, 2011.

[367] Daniel Zingaro. Peer instruction contributes to self-efficacy in CS1. In Proceedings of the 45th ACM technical symposium on Computer Science Education, pages 373–378, 2014.