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

Xinying Hou
University of Michigan
Ann Arbor, Michigan, USA
xyhou@umich.edu

Zihan Wu
University of Michigan
Ann Arbor, Michigan, USA
ziwu@umich.edu

Xu Wang
University of Michigan
Ann Arbor, Michigan, USA
xwanghci@umich.edu

Barbara J. Ericson
University of Michigan
Ann Arbor, Michigan, USA
barbarer@umich.edu

ABSTRACT

Learning to program can be challenging, and providing high-quality and timely support at scale is hard. Generative AI and its products, like ChatGPT, can create a solution for most intro-level programming problems. However, students might use these tools to just generate code for them, resulting in reduced engagement and limited learning. In this paper, we present CodeTailor, a system that leverages a large language model (LLM) to provide personalized help to students while still encouraging cognitive engagement. CodeTailor provides a personalized Parsons puzzle to support struggling students. In a Parsons puzzle, students place mixed-up code blocks in the correct order to solve a problem. A technical evaluation with previous incorrect student code snippets demonstrated that CodeTailor could deliver high-quality (correct, personalized, and concise) Parsons puzzles based on their incorrect code. We conducted a within-subjects study with 18 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 baseline. Overall, most participants preferred using CodeTailor versus 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. We suggest future design ideas to facilitate active learning opportunities with generative AI techniques.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**; • **Human-centered computing** → **Interactive systems and tools**.

KEYWORDS

Parsons Problems, Personalization, Large Language Models, Introductory Programming, GPT, Generative AI, Active Learning

ACM Reference Format:

Xinying Hou, Zihan Wu, Xu Wang, and Barbara J. Ericson. 2024. CodeTailor: LLM-Powered Personalized Parsons Puzzles for Engaging Support While Learning Programming. In *Proceedings of the Eleventh ACM Conference on Learning @ Scale (L@S '24), July 18–20, 2024, Atlanta, GA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3657604.3662032

L@S '24, July 18–20, 2024, Atlanta, GA, USA

ACM ISBN 979-8-4007-0633-2/24/07
https://doi.org/10.1145/3657604.3662032

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 [5, 53]. This process can be frustrating and lead to doubts about their ability to learn to program [20, 36]. Meanwhile, the notable increase in the enrollment of intro CS courses makes one-on-one guidance from human experts hard to scale in large classrooms [18, 54]. Recently, large language models (LLMs) have opened up new ways for timely, adaptive, and scalable programming support [46]. For example, they can create complete programs directly from natural language input [30, 44]. However, as LLM products have increased adaptability and ease of use, there are rising concerns about their over-utilization in computing education [35]. Because LLMs can solve most existing CS1 programming problems [12], students may simply copy the problem description to an AI code generator and copy the solution back to their development environment, without thinking about the AI-generated solution [30]. Another concern is that today's AI systems can generate inaccurate solutions and potentially mislead students [35].

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-staged personalized puzzles** to support struggling students while programming (Fig. 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 [11].

CodeTailor supports students with personalized Parsons puzzles. In a Parsons puzzle, students are presented with mixed-up blocks in a source area on the left, and the student drags blocks and arranges them in order in a solution area on the right, as shown in Fig. 1 [15, 42]. Parsons puzzles can have distractor blocks that are not needed in a correct solution. In CodeTailor, when students work on programming tasks and get stuck, they can request help, and CodeTailor will then provide a two-staged 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 students' current problem-solving progress in three dimensions: *pre-placement of correct lines* – students' correctly

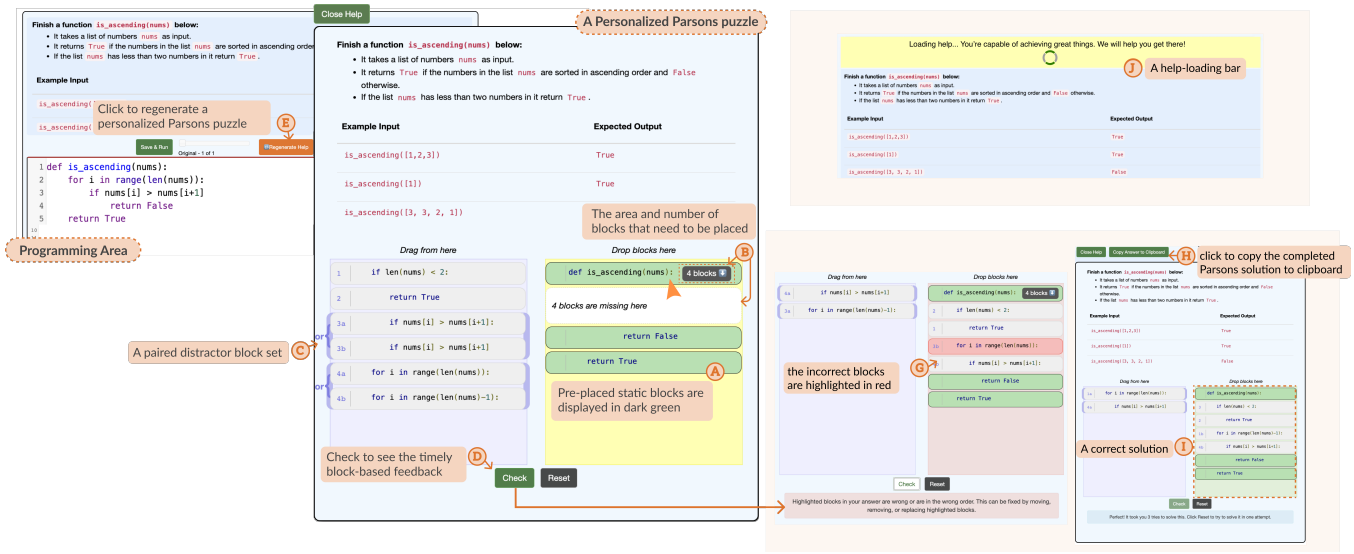

Figure 1: The CodeTailor interface contains a programming area on the left and a pop-up personalized Parsons puzzle as support on the right. It provides timely feedback after checking (G) and allows students to copy (H) the finished solution (I).

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 simulates 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 [30]. Results showed that students found CodeTailor more engaging than just receiving an AI generated solution and most students (88%) preferred using CodeTailor versus 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.

2 RELATED WORK

This section motivates the design of CodeTailor by discussing prior research on using LLMs in CS education and other methods that assist students who struggle while programming.

2.1 Large Language Models in CS education

With the development of generative AI, educational researchers are studying how it can contribute to instructional content creation and personalized learning experiences [3, 21, 51]. In CS education, researchers are exploring the potential of LLMs, including assessing the performance on completing different CS learning tasks [12] and generating instructional content [8, 33, 43]. Research has also explored how LLMs can fulfill educational roles beyond offline instructional content creation, including serving as TAs to respond to help requests [23, 63], as pair programmers [46], and as simulated students for teacher training [38].

While the above work highlights LLMs' capability to benefit CS learning, concerns over the inappropriate application of such systems in education are growing [32, 41, 56], especially when it comes to students' abusing LLMs' ability to generate code. After speaking with instructors of beginner-level programming courses, Lau and Guo reported that a common concern was that students who relied on AI code generation tools to get answers would not learn the material [35]. As Kazemitabaar et al. 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. Of these initial AI code generator usages, nearly half of the instances were requesting a complete solution [30]. Additionally, novice students who over-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 [30]. 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 [37]. However, some students found it difficult to use because they had to type their input and did not always know what to ask. Furthermore, the long and plain textual materials discouraged active learning unless students used self-explanation while reading the text [11]. In addition, the generated output could be factually incorrect due to the limitations of LLMs. Recent generative AI models perform well in understanding and generating code outputs, which forms a basis for scalable personalization in programming learning scenarios [48]. To harness the power of LLMs and address the two aforementioned concerns, CodeTailor aims to achieve two high-level goals: **generate correct code solutions using LLMs in real-time**, and **establish a programming support system that encourages engagement and active learning**.

2.2 Scaffolding student programming

Previous research on assisting students while learning programming has investigated different scaffolding methods. Scaffolding, as defined by Bruner [7], involves providing support to students to learn a skill beyond their current level. One common approach to timely assistance is detailed hints based on the students' current code state, aiming to help students progress toward a correct solution [31, 39, 50, 63]. A more comprehensive scaffolding resource could be a library of code examples for students to refer to during practice [61]. However, one concern about using examples or hints is that they could lead to disengagement and passive learning [11].

Active learning refers to instructional methods that engage students cognitively and meaningfully with the instructional materials in a learning task [11, 47]. Instead of just passively receiving information, students interact with the instructional materials and actively digest the content [6]. The ICAP framework, proposed by Chi and Wylie, 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 [11]. 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 [2, 9, 19]. *Parsons puzzles* are a type of completion problem [57] and solving these puzzles involve physical movement and attention by rearranging code blocks and choosing a block from a set of options, aligning with the concept of active engagement in the ICAP framework [11, 14]. Therefore, delivering personalized Parsons puzzles as scaffolding can encourage students to cognitively attend to the practice material. This approach has the potential to provide greater learning than just reading text. This goal drove the design of CodeTailor to **deliver Parsons puzzles as scaffolding to support students who struggle while programming**.

2.3 Parsons puzzles

In Parsons puzzles, students arrange mixed-up blocks to solve a problem [16, 42]. Typically, the mixed-up blocks are presented separately from the solution area. Students need to drag all the required blocks into the solution area and arrange them there [15]. There are various Parsons puzzle types. For example, in one-dimensional Parsons puzzles, code blocks only need to be organized in the right vertical sequence, while in two-dimensional Parsons puzzles, the blocks must additionally be appropriately indented horizontally [28]. Distractors, code blocks that are not part of the correct solution, can also be added to illustrate common misconceptions [15]. Problems with paired distractors, where learners are instructed to pick the correct block from a visually paired block set with one correct block and one distractor block, are easier to solve than those with unpaired distractors [15].

Previous research has explored using a fully movable two-dimensional Parsons puzzle to assist students who struggle while programming from scratch [24–26]. In a fully movable Parsons puzzle, students start with an empty solution area, and have to drag and arrange all the needed blocks into the solution area to form a solution. Also, Parsons puzzles were created from a representative most common prior student solution and had expert-created distractors in the previous studies[24–26]. The results demonstrated that this scaffolding could improve students' practice performance and problem-solving efficiency compared to traditional text entry without any support [24, 26]. However, challenges were also reported. For example, the provided solution in the puzzle did not make sense to students who had a different strategy in mind. Also, the expert-written distractor blocks sometimes led students astray unnecessarily [22, 24]. In addition, using a fully movable Parsons puzzle as scaffolding may be unnecessary for some students, as they may merely scan or move some blocks without actually solving the problem [26]. Students may benefit from more concise support that does not require them to work on a full puzzle. CodeTailor was designed to address these challenges, and it incorporates improvements at both the solution and Parsons block level: **solutions in the Parsons puzzles should be close to students' existing code; the provided Parsons puzzles should be concise;** and **the Parsons puzzles should pinpoint students' misconceptions.**

3 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 [59].

3.1 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 (Fig. 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, it 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 (Fig. 2A-P1) with a "Combine Blocks" feature (Fig. 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** (Fig. 1.A), which are pre-placed in the solution area to make the puzzle more concise to solve, and **paired distractor sets** (Fig. 1.C), which are unnecessary blocks that emphasize misconceptions. Static correct blocks are created from the student's correctly written code (Fig. 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 (Fig. 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 (Fig. 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" (Fig. 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 (Fig. 2B-P2) or generated using an LLM from the student's correctly written code (Fig. 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 (Fig. 1.G). After completing a puzzle (Fig. 1.I), students can copy the solution to the clipboard with the "Copy Answer to Clipboard" button instead of retyping it (Fig. 1.H). Students can also regenerate a new personalized Parsons puzzle (Fig. 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.

3.2 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 block" feature and no distractors (Fig. 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 (Fig. 2A). Rita tries to solve the puzzle but still struggles after four tries. She then combines two blocks into one (Fig. 2A.F). She then completes the puzzle, retypes the puzzle solution, and passes all unit tests.

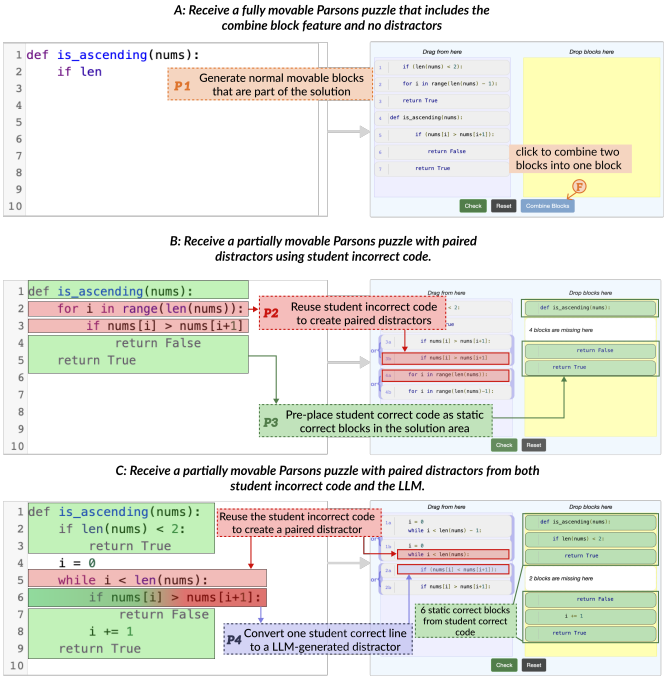

Figure 2: Three example scenarios with CodeTailor

Scenario B: Receive a partially movable Parsons puzzle with paired distractors created from student errors (Fig. 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. (Fig. 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 her code, and passes all the unit tests.

4 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.

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

4.1.1 Pipeline overview. First, CodeTailor fills in a prompt template with the student's incorrect code and corresponding problem information (Fig. 3-1). Then, it sends the finished prompt to the LLM model (OpenAI's GPT-4 model in our case) to generate a response. After receiving the response, CodeTailor pre-processes the response to only extract the LLM-generated code (Fig. 3-2) and automatically evaluates this LLM-generated code (Fig. 3-3) based on its correctness (Fig. 3-3.1) and closeness with the student's incorrect code (Fig. 3-3.2). If the LLM-generated code passes both evaluations, it continues to Stage 2 to generate Parsons puzzle blocks (Fig. 3-Stage 2). If the code does not pass the evaluations, CodeTailor sends a request back to the LLM with an updated prompt (Fig. 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 most common prior solution.

4.1.2 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[1]. The system message

[1]https://platform.openai.com/docs/api-reference/chat/create

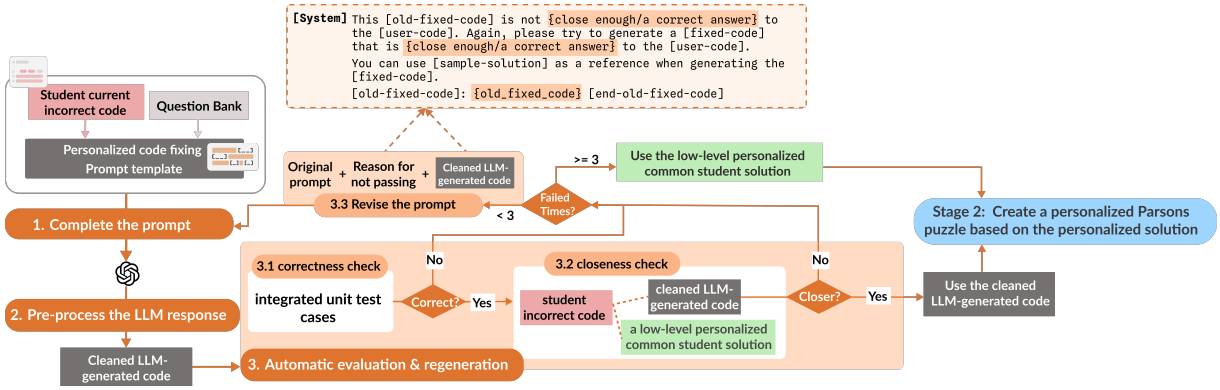

Figure 3: Pipeline for generating a personalized correct solution from an incorrect code in CodeTailor (Stage 1)

```
[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 4: The main prompt template used in CodeTailor

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 Fig. 4. If the initial LLM-generated code does not meet CodeTailor's evaluation criteria (for correctness and closeness), two more requests will be 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 Fig. 3-3.3.

4.1.3 Preprocessing and automatic evaluation of the LLM response. After getting the LLM response, CodeTailor pre-processes it by only keeping the LLM-generated code (Fig. 3-2). Next, it performs an automatic evaluation of the LLM-generated code in two ways (Fig. 3-3): (1) *correctness*: the code must pass all the unit tests integrated in CodeTailor (Fig. 3-3.1); (2) *closeness*: the code must be closer to the student's incorrect code than a most common prior solution personalized with the student's current variable names (low-level personalization) (Fig. 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 [50] 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 (Fig. 3-3.2-Green).

In determining how close two code pieces are, standard methods like Abstract Syntax Tree (AST) edit distance [27] 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 [17] or OpenAI embeddings, can be time-consuming. Therefore, CodeTailor compared 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.

4.2 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 (Fig. 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 (Fig. 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 student's incorrect line as the distractor (e.g., above 0.7 in CodeTailor) (Fig. 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 (Fig. 2C), if the students have less 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 (Fig. 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 (Fig. 1, right).

5 TECHNICAL EVALUATION: CODETAILOR'S MATERIAL QUALITY

As described in Section 2, 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 a 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 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 (Institutional Review Board) 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 Stage 1 (Section 4.1) and puzzles from Stage 2 (Section 4.2) for evaluation.

5.2 Evaluation Results

5.2.1 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 representative most common solution, as mentioned in Section 4.1.3. Therefore, the final CodeTailor-generated solutions were always correct.

5.2.2 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 representative most common student code solution as the baseline [24]. 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 4.1. After obtaining the two similarities, a Wilcoxon signed-rank test [2] 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), *Median* = 0.7, is significantly higher than the incorrect-baseline similarity, *M (SD)* = 0.5 (0.1), *Median* = 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.

5.2.3 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), *Median* = 6.0, had significantly fewer movable blocks compared to the fully movable Parsons puzzles, *M (SD)* = 8.1 (3.0), *Median* = 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.

6 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 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?

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

- 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?

6.1 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.

6.1.1 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've 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.

6.1.2 The Baseline Support. One goal was to understand if 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 4.1.

6.1.3 Procedure. Participants were randomly 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 (Fig. 5). Participants only received one type of support for the two practice questions on the same topic. This allows us to compare their practice, posttest performance, and application of supported elements in the posttest by support type.

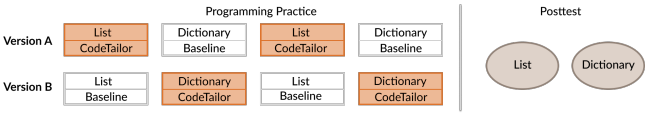

Figure 5: Two practice versions: In Version A, students do two list exercises with personalized Parsons puzzles (CodeTailor) and two dictionary exercises with just the AI-generated code solutions. Version B has the reverse setting.

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.

6.1.4 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 have equal difficulty levels. The posttest included two programming problems that were at the near to middle transfer level for the corresponding practice questions [4, 40]. Each posttest problem consisted of the same key elements that aligned with the practice questions on the same topic in the study.

6.1.5 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 with 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

Table 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

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 was not normally distributed, we used the Wilcoxon signed-ranks 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 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.

6.2 Results and Findings

6.2.1 RQ1: CodeTailor is 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 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."*

6.2.2 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. Seventy percent (10 of 14) of the 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 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 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 that, *"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.

6.2.3 RQ3: Most students prefer 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 has already reduced the solution space and pre-placed their correctly written lines, but still gives 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 Block" feature when he was unable to identify the issue with his puzzle after 15 failed attempts. He utilized the "Combine Block" to obtain the correct solution less frustratingly.

CodeTailor encouraged students to think about the construction of the solution. Because CodeTailor includes a correct answer in the mixed-up order, students are more encouraged to think when ordering the blocks. Their main focus shifted to understanding the program flow. Participants pointed out that CodeTailor helps 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 arranging block activity in CodeTailor, students can understand the overall structure 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 when 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 seems easy to solve with only four blocks in total, P11 had difficulty understanding it (Fig. 6).

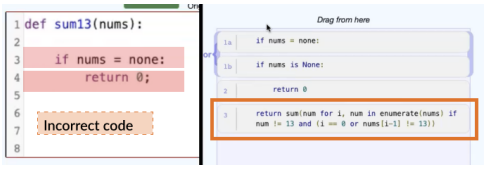

Figure 6: A complex shorthand one-line block in CodeTailor.

Getting a direct solution (the baseline) is still preferred for quick problem-solving. Two students (P3&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)."*

7 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.

7.1 Address the concerns of LLMs in education

Two growing concerns about using GenAI 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 [30, 35]. 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 and allows students to apply existing knowledge [11] and prevents them from being passively engaged or even disengaged by only reading the materials [9, 58]. In addition, deeper cognitive engagement strategies could be triggered in scenarios B and C through the selected-response activity with distractor blocks (Fig. 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 complete solution all at once.

Regarding the concern for AI models to be *incorrect*, CodeTailor has created an automated evaluation pipeline to check the raw LLM output, with guardrails in place to make sure the materials shown to students are correct [29]. 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.

7.2 Enhance the instructional effectiveness

In the student evaluation, we found some students struggled to understand the code blocks. As reported in Kuttal et al. [34] and Simon and Snowdon [55], students are unable to understand the purpose of code without a proper explanation. 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 [10].

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 attempting 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 [13, 52]. 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.

Thirdly, 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 (Fig. 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 spin 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.

7.3 Potentials 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 [35, 44]. 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 [60, 62]. 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 [49]. 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 [15]. For example, in math learning, sequencing items are commonly required for proofs [45]. 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 plagiarism with generative AI tools in different learning contexts. CodeTailor can be adapted to support these activities by simply changing the content.

8 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) Limited feedback on the quality of LLM-produced Parsons distractors due to few falling 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 LLM in real-time, hence the costs could be high for large classroom settings; future work should explore alternative methods to reduce costs.

9 CONCLUSION

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, preplace 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. Also, students found CodeTailor as more engaging and could apply significantly more supported elements from the scaffolded practice to the posttest after using CodeTailor than just getting the correct solution. Overall, most students preferred CodeTailor for learning versus just receiving an AI-generated code solution.

10 ACKNOWLEDGEMENT

The funding came from the National Science Foundation award 2143028. Any conclusions expressed in this material do not necessarily reflect the views of NSF.

REFERENCES

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

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

[3] David Baidoo-Anu and Leticia Owusu Ansah. 2023. Education in the era of generative artificial intelligence (AI): Understanding the potential benefits of ChatGPT in promoting teaching and learning. *Available at SSRN 4337484* (2023).

[4] Susan M Barnett and Stephen J Ceci. 2002. When and where do we apply what we learn?: A taxonomy for far transfer. *Psychological bulletin* 128, 4 (2002), 612.

[5] Klara Benda, Amy Bruckman, and Mark Guzdial. 2012. 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 (2012), 1–38.

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

[7] Jerome Seymour Bruner. 1966. *Toward a theory of instruction.* Harvard University Press.

[8] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022).

[9] 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. 2018. Translating the ICAP theory of cognitive engagement into practice. *Cognitive science* 42, 6 (2018), 1777–1832.

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

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

[12] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1.* 1136–1142.

[13] Paul Denny, James Prather, Brett A. Becker, Catherine Mooney, John Homer, Zachary C Albrecht, and Garrett B. Powell. 2021. 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* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 55, 15 pages. https://doi.org/10.1145/3411764.3445696

[14] Barbara Ericson and Carl Haynes-Magyar. 2022. Adaptive Parsons Problems as Active Learning Activities During Lecture. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1.* 290–296.

[15] 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. 2022. 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* (2022), 191–234.

[16] Barbara J Ericson, Janice L Pearce, Susan H Rodger, Andrew Csizmadia, Rita Garcia, Francisco J Gutierrez, Konstantinos Liaskos, Aadarsh Padiyath, Michael James Scott, David H Smith IV, et al. 2023. Multi-Institutional Multi-National Studies of Parsons Problems. In *Proceedings of the 2023 Working Group Reports on Innovation and Technology in Computer Science Education.* 57–107.

[17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[18] Lawrence M Fisher. 2016. Booming enrollments. *Commun. ACM* 59, 7 (2016), 17–18.

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

[20] Jamie Gorson and Eleanor O'Rourke. 2020. 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.* 170–181.

[21] 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. 2023. RECIPE: How to Integrate ChatGPT into EFL Writing Education. In *Proceedings of the Tenth ACM Conference on Learning @ Scale* (Copenhagen, Denmark) *(L@S '23)*. Association for Computing Machinery, New York, NY, USA, 416–420. https://doi.org/10.1145/3573051.3596200

[22] Kyle James Harms, Jason Chen, and Caitlin L Kelleher. 2016. Distractors in Parsons problems decrease learning efficiency for young novice programmers. In *Proceedings of the 2016 ACM Conference on International Computing Education Research.* 241–250.

[23] Arto Hellas, Juho Leinonen, Sami Sarsa, Charles Koutcheme, Lilja Kujanpää, and Juha Sorva. 2023. Exploring the Responses of Large Language Models to Beginner Programmers' Help Requests. *arXiv e-prints* (2023), arXiv–2306.

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

[25] Xinying Hou, Barbara Jane Ericson, and Xu Wang. 2023. 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.* 665–665.

[26] Xinying Hou, Barbara Jane Ericson, and Xu Wang. 2023. 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.* Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3631802.3631832

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

[28] Petri Ihantola and Ville Karavirta. 2011. Two-dimensional parson's puzzles: The concept, tools, and first observations. *Journal of Information Technology Education* 10, 2 (2011), 119–132.

[29] Firuz Kamalov, David Santandreu Calonge, and Ikhlaas Gurrib. 2023. New Era of Artificial Intelligence in Education: Towards a Sustainable Multifaceted Revolution. *Sustainability* 15, 16 (2023), 12451.

[30] Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara J Ericson, David Weintrop, and Tovi Grossman. 2023. How Novices Use LLM-Based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment. *arXiv preprint arXiv:2309.14049* (2023).

[31] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)* 19, 1 (2018), 1–43.

[32] Mohammad Khalil and Erkan Er. 2023. Will ChatGPT get you caught? Rethinking of plagiarism detection. *arXiv preprint arXiv:2302.04335* (2023).

[33] Natalie Kiesler, Dominic Lohr, and Hieke Keuning. 2023. Exploring the Potential of Large Language Models to Generate Formative Programming Feedback. *arXiv preprint arXiv:2309.00029* (2023).

[34] Sandeep Kaur Kuttal, Bali Ong, Kate Kwasny, and Peter Robe. 2021. 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.* 1–20.

[35] Sam Lau and Philip Guo. 2023. From" Ban it till we understand it" to" Resistance is futile": How university programming instructors plan to adapt as more students use AI code generation and explanation tools such as ChatGPT and GitHub Copilot. In *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1.* 106–121.

[36] Colleen M Lewis, Ken Yasuhara, and Ruth E Anderson. 2011. 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.* 3–10.

[37] Mark Liffiton, Brad Sheese, Jaromir Savelka, and Paul Denny. 2023. CodeHelp: Using Large Language Models with Guardrails for Scalable Support in Programming Classes. *arXiv preprint arXiv:2308.06921* (2023).

[38] Julia M Markel, Steven G Opferman, James A Landay, and Chris Piech. 2023. GPTeach: Interactive TA Training with GPT Based Students. (2023).

[39] Samiha Marwan, Nicholas Lytle, Joseph Jay Williams, and Thomas Price. 2019. 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.* 520–526.

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

[41] Jo Ann Oravec. 2023. Artificial Intelligence Implications for Academic Cheating: Expanding the Dimensions of Responsible Human-AI Collaboration with ChatGPT. *Journal of Interactive Learning Research* 34, 2 (2023), 213–237.

[42] Dale Parsons and Patricia Haden. 2006. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52.* 157–163.

[43] Tung Phung, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. 2023. Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models. *arXiv preprint arXiv:2302.04662* (2023).

[44] Leo Porter. 2024. *Learn AI-Assisted Python Programming: With Github Copilot and ChatGPT.* Simon and Schuster.

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

[46] James Prather, Brent N Reeves, Paul Denny, Brett A Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. "It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. *arXiv preprint arXiv:2304.02491* (2023).

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

[48] Tareq Rasul, Sumesh Nair, Diane Kalendra, Mulyadi Robin, Fernando de Oliveira Santini, Wagner Junior Ladeira, Mingwei Sun, Ingrid Day, Raouf Ahmad Rather, and Liz Heathcote. 2023. The role of ChatGPT in higher education: Benefits, challenges, and future research directions. *Journal of Applied Learning and Teaching* 6, 1 (2023).

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

[50] Kelly Rivers. 2017. *Automated data-driven hint generation for learning programming*. Ph. D. Dissertation. Carnegie Mellon University.

[51] Jürgen Rudolph, Shannon Tan, and Samson Tan. 2023. 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).

[52] Juan Pablo Saenz and Luigi De Russis. 2022. On How Novices Approach Programming Exercises Before and During Coding. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 361, 6 pages. https://doi.org/10.1145/3491101.3519655

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

[54] Linda J Sax, Kathleen J Lehman, and Christina Zavala. 2017. Examining the enrollment growth: Non-CS majors in CS1 courses. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 513–518.

[55] Simon and Susan Snowdon. 2011. Explaining program code: giving students the answer helps-but only just. In *Proceedings of the seventh international workshop on Computing education research*. 93–100.

[56] Adele Smolansky, Andrew Cram, Corina Raduescu, Sandris Zeivots, Elaine Huber, and Rene F Kizilcec. 2023. 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*. 378–382.

[57] John Sweller, Paul Ayres, and Slava Kalyuga. 2011. The worked example and problem completion effects. In *Cognitive load theory*. Springer, 99–109.

[58] John Sweller, Jeroen JG van Merriënboer, and Fred Paas. 2019. Cognitive architecture and instructional design: 20 years later. *Educational Psychology Review* 31 (2019), 261–292.

[59] Dwayne Towell and Brent Reeves. 2010. From Walls to Steps: Using online automatic homework checking tools to improve learning in introductory programming courses. (2010).

[60] Yune Tran. 2019. Computational thinking equity in elementary classrooms: What third-grade students know and can do. *Journal of Educational Computing Research* 57, 1 (2019), 3–31.

[61] Wengran Wang, John Bacher, Amy Isvik, Ally Limke, Sandeep Sthapit, Yang Shi, Benyamin T Tabarsi, Keith Tran, Veronica Cateté, Tiffany Barnes, et al. 2023. Investigating the Impact of On-Demand Code Examples on Novices' Open-Ended Programming Experience. In *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1*. 464–475.

[62] Jeannette M Wing. 2006. Computational thinking. *Commun. ACM* 49, 3 (2006), 33–35.

[63] Ruiwei Xiao, Xinying Hou, and John Stamper. 2024. Exploring How Multiple Levels of GPT-Generated Programming Hints Support or Disappoint Novices. *arXiv preprint arXiv:2404.02213* (2024).