# DBox: Scaffolding Algorithmic Programming Learning through Learner-LLM Co-Decomposition

Shuai Ma*
Aalto University
Espoo, Finland
shuai.ma@aalto.fi

Junling Wang
ETH Zürich
Zürich, Switzerland
junling.wang@ai.ethz.ch

Yuanhao Zhang
The Hong Kong University of Science and Technology
Hong Kong, China
yzhangiy@connect.ust.hk

Xiaojuan Ma
The Hong Kong University of Science and Technology
Hong Kong, China
mxj@cse.ust.hk

April Yi Wang
ETH Zürich
Zürich, Switzerland
april.wang@inf.ethz.ch

## Abstract

Decomposition is a fundamental skill in algorithmic programming, requiring learners to break down complex problems into smaller, manageable parts. However, current self-study methods, such as browsing reference solutions or using LLM assistants, often provide excessive or generic assistance that misaligns with learners' decomposition strategies, hindering independent problem-solving and critical thinking. To address this, we introduce Decomposition Box (DBox), an interactive LLM-based system that scaffolds and adapts to learners' personalized construction of a step tree through a *"learner-LLM co-decomposition"* approach, providing tailored support at an appropriate level. A within-subjects study (N=24) found that compared to the baseline, DBox significantly improved learning gains, cognitive engagement, and critical thinking. Learners also reported a stronger sense of achievement and found the assistance appropriate and helpful for learning. Additionally, we examined DBox's impact on cognitive load, identified usage patterns, and analyzed learners' strategies for managing system errors. We conclude with design implications for future AI-powered tools to better support algorithmic programming education.

## CCS Concepts

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

## Keywords

Programming Learning, Self-Paced Learning, Large Language Models, AI for Coding, Human-AI Collaboration

*Work done during the first author's PhD studies at The Hong Kong University of Science and Technology.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*CHI '25, Yokohama, Japan*
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1394-1/25/04
https://doi.org/10.1145/3706598.3713748

**ACM Reference Format:**
Shuai Ma, Junling Wang, Yuanhao Zhang, Xiaojuan Ma, and April Yi Wang. 2025. DBox: Scaffolding Algorithmic Programming Learning through Learner-LLM Co-Decomposition. In *CHI Conference on Human Factors in Computing Systems (CHI '25), April 26–May 01, 2025, Yokohama, Japan.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3706598.3713748

## 1 Introduction

Algorithmic programming, which involves applying algorithms to solve real-world problems, is a challenging yet essential skill for computer science learners [6, 69]. Unlike introductory programming, the primary challenge in algorithmic programming lies not in algorithmic concepts, basic logic or syntax but in decomposing a complex problem to develop a holistic solution [6, 70]. Students often struggle with formulating clear strategies, getting stuck at specific steps, or overlooking edge cases [41, 58]. Online knowledge communities (e.g., LeetCode) are commonly used for self-study, where learners can browse reference solutions provided by these platforms or other online resources (e.g., search engines or Q&A platforms) when encountering difficulties. However, these resources are not tailored to individual problem-solving approaches and can hinder independent thinking and learning.

In addition to traditional resources, the rapid advancement of large language models (LLMs) has introduced AI tools that rival or even surpass human performance in certain programming tasks [20]. Recent studies in CS education and human-computer interaction (HCI) have explored the impact of LLM-based tools like ChatGPT, GitHub Copilot, and Codex on introductory programming (CS1) [19, 31, 34, 63]. While LLMs are increasingly used for tasks such as code completion, translation, debugging, summarization, and explanation [29, 64], they are not specifically designed for educational purposes, leading to issues like over-reliance and a lack of independent problem-solving [35, 67], as highlighted in our formative study observing programming learners solving algorithmic problems using various AI support tools. Moreover, most research has focused on the impact of LLM on novice learners in introductory programming [19, 31, 34, 63], with little attention given to how these tools can be tailored to support algorithmic programming learning. Therefore, we pose the following research question: **How can LLM-supported interfaces be designed to**

**effectively facilitate pedagogically meaningful learning in algorithmic programming?**

In this paper, we present the Decomposition Box (DBox), a tool that leverages LLMs to assist learners in decomposing algorithmic programming problems. DBox adopts a *Learner-LLM Co-Decomposition* design, where learners lead the decomposition process while the LLM provides scaffolding only when needed. DBox supports two key stages of algorithmic programming: solution formation and implementation. During the solution formation stage, DBox introduces an interactive step tree that adapts to students' existing code or natural language thought processes. Students can express and update their ideas by directly writing code or iteratively constructing the step tree using interactive features, with DBox providing real-time feedback on the status of each node. In the implementation stage, DBox validates the alignment between learners' code and the step tree, identifying the status of each node in real time. A progressive hint system further supports learners by starting with thought-provoking questions for incorrect or missing steps and gradually offering more specific hints after repeated attempts. This design balances problem-solving progress with fostering independent thinking, with the hint system seamlessly integrated across both stages.

To evaluate DBox's effectiveness in supporting algorithmic programming learning, we conducted a within-subjects mixed-methods user study with 24 learners. Our analysis highlighted that DBox significantly improved students' programming performance, perceived learning gains, confidence, algorithmic thinking, and self-efficacy. Students reported greater cognitive engagement, critical thinking, and a stronger sense of achievement compared to the baseline, where many felt they were "cheating" or not solving problems independently. DBox was seen as offering more appropriate assistance and benefiting learning. Since DBox relies on LLM for assessments, we also conducted a technical evaluation to assess the quality of LLM prompts used in the pipeline. This evaluation demonstrated that LLM are sufficiently accurate to support DBox's features, particularly in identifying incomplete and incorrect code approaches, though some challenges were noted in handling natural language descriptions.

Building on the key findings from our experiment, we provided several design implications for developing tools that promote algorithmic programming learning. We also discussed DBox's design philosophy, particularly the human-AI co-decomposition approach, and explored how to effectively leverage LLM to support programming learning, even with LLM's imperfection. In summary, our contributions are as follows:

- A formative study that identified four challenges learners face in algorithmic programming learning with existing tools, leading to four design goals.
- The design of DBox, an AI-assisted algorithmic programming learning tool, features a scaffolded interactive step tree that facilitates learner-AI co-decomposition of problems, supporting both the ideation and implementation stages while fostering independent thinking and active learning.
- A technical evaluation of LLM' accuracy in assessing students' fine-grained thought processes, highlighting where LLMs excel and where they face challenges.
- A user study reported DBox's effectiveness on algorithmic programming learning, including its effects on learners' learning outcomes, perceptions, user experience, and usage patterns.

## 2 Related Work

### 2.1 Scaffolding Programming Learning

Scaffolding, rooted in constructivist theory, helps bridge the gap between learners' current abilities and desired skills by integrating new information into existing cognitive frameworks, rather than passively receiving it [10, 38, 54, 73, 81]. Learners with limited knowledge particularly benefit from scaffolding to enhance understanding and avoid floundering [26, 66]. In programming education, scaffolding helps structure the learning process, guiding learners through problem-solving via hints or correcting misconceptions [38, 72]. Traditional scaffolding often relies on human-provided support, but recent technological advancements have expanded its scope to include digital solutions [15, 38]. For example, methods such as using flowcharts to brainstorm and organize solution ideas have been shown to improve algorithm design and programming skills [70]. Similarly, Cunningham et al. describe a multi-stage programming process that includes explicit planning, offering a structured approach to learning [12]. Generally, providing immediate assistance during code writing—such as detailed feedback on errors, suggestions for corrections, or next-step hints—illustrates key scaffolding strategies that effectively enhance learner understanding [62, 68, 72]. Other supportive methods include the use of worked examples [77] and Parsons problems, which engage students in active problem-solving [28].

Adaptive scaffolding, which adjusts support based on real-time feedback from learners, is particularly effective. This approach tailors instruction to the learner's evolving understanding and knowledge level, though it poses significant challenges for tools that require cognitive models to interpret learner input [3, 11, 76]. Recent advancements in large language models (LLMs) have introduced opportunities for adaptive scaffolding by inferring learner's mental state from their inputs, enabling adaptive, learner-specific assistance. *However, most LLM applications in scaffolding have been limited to generating code explanations and next-step hints [50, 63], with little focus on adaptive scaffolding tailored to individual needs during algorithmic programming*. Our research addresses this gap by using LLMs to enhance the Zone of Proximal Development (ZPD) [9] in algorithmic programming through carefully crafted prompts. We capture students' thought processes in both code and verbal forms, forming a step-tree structure that reflects their current thinking. This model facilitates targeted assistance, assessing and intervening at various levels for each identified error or gap. By ensuring that scaffolding aligns with the learner's current knowledge state and providing only essential guidance, our approach promotes independent thinking and substantially improves learning outcomes.

### 2.2 AI Coding Assistants and Application in Educational Contexts

LLMs have demonstrated their capabilities in programming-related tasks, including delivering precise feedback on syntax errors [57] and enhancing programming workflows [40]. Sarsa et al. evaluated OpenAI Codex's potential for generating engaging programming

exercises [64], while MacNeil et al. observed that student engagement with LLM-generated code explanations varied depending on the complexity and length of the content [50]. Additionally, Prather et al. highlighted the dual-edged nature of GitHub Copilot's suggestions, cautioning against potential dependency issues and underscoring the importance of fostering meta-cognitive skills in novice programmers [60].

Recent work has also explored mapping natural language to code using LLMs, which aligns with aspects of our approach. For instance, Liu et al. [43] introduced grounded abstraction matching to help non-expert end-user programmers better understand LLM capabilities and refine their input for code generation. Their method iteratively converts natural language to code and back into grounded utterances to refine user input. While their focus is on improving language refinement, our work emphasizes scaffolding learners in decomposition and idea-building for solving algorithmic problems, placing less priority on natural language quality.

As LLMs advance in code generation, their applications in programming education have garnered increasing attention, particularly for creating educational content, enhancing student engagement, and personalizing learning experiences [33]. Recent studies have examined these opportunities, focusing on task completion [16], instructional content generation [40], and innovative methods for content creation [17]. Notably, Finnie-Ansley et al. found that OpenAI Codex outperformed most students in coding tasks during CS1 and CS2 exams [20]. Similarly, Kazemitabaar et al. studied how AI code generators like Codex support novice learners, showing increased code-authoring performance without compromising manual code-modification skills [35].

While much of the existing work focuses on introductory programming, our research addresses the challenges of advanced algorithmic programming, such as those encountered in CS2 courses. One closely related piece of work is that of Jin et al. [31], who designed a teachable agent based on LLMs, employing a learning-by-teaching approach to help students grasp algorithmic concepts. However, their focus is primarily on the mastery of algorithmic concepts, whereas our work addresses the difficulties students face in applying these concepts to solve practical problems. Another tool similar to our work is Code Tutor [1], which helps students tackle programming problems by guiding their thinking without giving direct answers. However, these conversational LLM tools differ from DBox in two key ways. First, their chat-based format can lead to students getting lost in lengthy conversations, whereas DBox uses an interactive step tree to build a structured mental model. Second, these tools rely on students inputting code or questions disconnected from their existing work, while DBox integrates the step tree with the editor, providing better context understanding and reducing extra input, thereby improving learning efficiency.

## 2.3 Computational Thinking and Problem Decomposition

Computational Thinking (CT), as originally conceptualized by Wing [80], includes essential cognitive processes such as decomposition, abstraction, generalization, algorithmic thinking, and debugging [4]. Among these, decomposition is widely recognized as a cornerstone of CT. Early studies, such as those by Roy Pea, emphasized how programming environments like LOGO foster planning and decomposition skills by encouraging learners to break problems into smaller, manageable subproblems [53, 55]. McCracken et al. [51] further highlighted its significance through a five-step learning framework for CS1 courses: (1) abstracting the problem from its description, (2) generating subproblems, (3) transforming these into sub-solutions, (4) re-composing them into a complete program, and (5) evaluating and iterating.

The importance of decomposition is also reflected in various educational strategies. For example, Keen and Mammen implemented a term-long course project that initially provided explicit guidance on program decomposition, gradually reducing support as the course progressed [37]. Similarly, Sooriamurthi designed an exercise for CS1 students that required segmenting a large programming task into smaller components, promoting abstraction and iterative development skills [71]. Pearce et al. adopted a guided inquiry-based learning method in CS1, explicitly teaching decomposition strategies and using a rubric to assess students' skills. Their findings demonstrated that students who received explicit scaffolding showed greater proficiency in breaking down problems [56].

Despite the well-documented value of decomposition in cultivating CT and programming skills, few tools leverage LLMs for problem decomposition. One related work by Kazemitabaar et al. [36] explored task decomposition in data analysis programming, helping users break tasks into substeps or subphases to better steer and validate LLM-generated assumptions and code. However, their approach depends on LLMs automatically generating decomposition solutions for users to review. In contrast, our work emphasizes scenarios where learners actively develop decomposition skills, with LLMs providing minimal scaffolding. We term this approach "learner-LLM co-decomposition", fundamentally distinct from the "LLM-generate then user-verify" paradigm.

To mitigate the aforementioned gaps, this paper introduces DBox, a tool designed to help students break down complex problems into manageable steps using a structured step tree. Through learner-AI co-decomposition, learners actively build the step tree while receiving step-level feedback from the LLM, enhancing their computational thinking and problem-solving skills.

# 3 Formative Study

In addition to traditional programming support tools like LeetCode, search engines, and Q&A sites like Stack Overflow, AI-assisted tools such as ChatGPT and GitHub Copilot have further enriched these resources. However, it remains unclear whether these tools effectively enhance algorithmic programming learning and whether challenges persist despite their availability. To explore this, we conducted a formative study using contextual inquiry and interviews to understand learner's needs and obstacles. Our study focuses on students who have completed foundational computer science courses and are working to improve their algorithmic problem-solving skills.

## 3.1 Study Procedure

We recruited 15 university students (5F, 10M, aged 18-29), including 10 undergraduates and 5 graduate students. Most (11) were computer science majors, with the rest from related fields such as

data science and electrical engineering. All participants had prior experience with LeetCode or similar platforms, 14 had used ChatGPT for programming tasks, and eight had experience with GitHub Copilot or similar tools. Each session lasted 40 minutes with $10 compensation.

During the study, participants used tools like LeetCode, search engines, ChatGPT, and GitHub Copilot to solve a randomly assigned algorithmic problem for 20 minutes, during which we observed their tool usage and conducted contextual inquiries as notable behaviors arose. Afterward, we reviewed their activities and conducted a semi-structured interview to discuss their tool choices, usage, assistance received, and perceptions of the tools, including any benefits or drawbacks they experienced.

## 3.2 Data Analysis and Results

We conducted inductive thematic analysis [30] on interviews and contextual inquiry data. Recorded sessions were transcribed and manually reviewed and corrected by the first author. Two authors then independently coded the data and discussed to finalize themes and categorizations. The analysis revealed four key challenges students face with existing assistance tools during algorithmic programming learning.

*3.2.1* ***Challenge 1: Excessive Help Hindering Active Learning.*** Students expressed concerns that platforms like LeetCode and ChatGPT provide solutions that are too easily accessible, hindering their ability to engage in independent problem-solving. While learners typically need minimal guidance to overcome specific hurdles, these tools often present complete solutions prematurely, making the learning experience "*uninteresting*" and "*unfulfilling*" (P3). As P1 noted, "*I just wanted help with the step I'm stuck on, but the solution panel [in Leetcode] showed everything, making it hard to ignore what I didn't need.*" Participants also found it difficult to use ChatGPT as a learning tool, as it often provided full solutions rather than encouraging active engagement. P5 explained, "*ChatGPT often directly showed the complete solution and code, even though I just asked GPT about the specific problem I was facing. Once I saw the solution, it was hard to ignore it. It felt like I was not really improving my programming skills.*" Additionally, participants like P12 noted that making ChatGPT useful for active learning required significant effort to "carefully craft prompt", as a result they "*just copy the problem description directly into ChatGPT*".

*3.2.2* ***Challenge 2: Misalignment Between Provided Solutions and Learners' Own Problem-Solving Approaches.*** Students often struggled when provided solutions failed to align with their intended approaches or existing code. While learners preferred developing their own strategies, the solutions offered by platforms like LeetCode, search engines, or ChatGPT were often prescriptive that mismatched their efforts. Even when algorithms matched, differences in specifics made integration challenging, especially with partial or incorrect code. Rather than starting over, learners wanted help tailored to their approach and context. As P5 explained, "*I prefer sticking to my own ideas. The provided solution takes a different approach, and I just need help tailored to my method. I don't want to change my thinking to fit someone else's solution, even if it's better.*" Similarly, P8 noted the frustration of aligning external solutions with her own work: "*I don't want to scrap my code and start anew, so I try to align the solution with my code line by line. It's tedious to figure out which parts of the solution relate to what I've written, where my errors began, and what I missed.*"

*3.2.3* ***Challenge 3: Lack of a Structured Problem-Solving Approach.*** Students highlighted the need for a structured method to break down complex algorithmic problems into manageable steps, such as initialization, handling edge cases, loops, and return values. Current tools often present solutions as large blocks of text or code without clear visual structure. As P13 noted when using LeetCode's solution panel, "*The solution outlines six steps, but the content is very disorganized and the text is unclear, making it hard to follow a structured process.*" Some participants preferred step-by-step interactions with ChatGPT, asking specific questions as they progressed. However, resolving issues often required lengthy, multiple rounds of conversations, making it difficult to stay organized. Learners frequently had to scroll back and forth to find points related to specific steps. As P15 (using ChatGPT) shared, "*I had a detailed conversation with GPT spanning several pages. Scrolling back to revisit earlier points was cumbersome, and despite getting answers, it was hard to organize them into a coherent thought process.*"

*3.2.4* ***Challenge 4: Insufficient Fine-Grained Feedback on Learners' Coding Progress.*** LeetCode's "run code" button evaluates the entire solution and only provides feedback on the overall correctness. This all-or-nothing approach overlooks partial correctness and fails to highlight specific errors, making it difficult for students to track their progress or verify their understanding step by step. For example, P8 noted, "*I need to know if a step is correct before I can decide how to proceed. Without step-by-step feedback, I can only keep doing it until I complete all the steps.*" Additionally, students wanted early validation of their thought process before committing to code. Ten participants expressed the need for a feature to confirm if their approach was on the right track. As P1 stated, "*The correctness of my initial thought process is crucial. I hesitate to implement code until I'm confident it's correct. There's no tool to validate my approach early on.*"

## 3.3 Design Goals

Based on the challenges identified in our formative study, we established the following design goals for our tool:

- **D1: Scaffolding for Active Learning and Independent Thinking**: To address Challenge 1, our system should provide scaffolding support. Instead of presenting complete solutions, it should offer tailored support that encourages active problem-solving and independent thinking.
- **D2: Personalization to Individual Problem-Solving Styles**: In response to Challenge 2, our system should adapt to each student's unique problem-solving approach by analyzing how they break down problems and offering personalized feedback that preserves and enhances learners' own strategies.
- **D3: Connection and Structured Solution Presentation**: To address Challenge 2&3, our system should visually connect the system-generated guidance to students' existing codes and solutions, promoting a structured problem-solving approach that aligns with their mental models.

- **D4: Fine-Grained Evaluation and Feedback**: Addressing Challenge 4, our system should evaluate the correctness of students' thought processes—whether conveyed through code, pseudocode, or natural language—and provide detailed, step-by-step feedback to support continuous improvement.

# 4 Decomposition Box

## 4.1 Overview

As shown in Figure 1 (top row), DBox's interface has three main parts. The Problem Description (Figure 1.A)) and Solution Code Editor (Figure 1.B) are similar to the LeetCode platform. The Interactive Step Tree Widget (Figure 1.D) enables users to refine their thought process and receive feedback via an interactive step tree. Three buttons—"From Editor to Step Tree", "Check Match", and "Copy to Comments"—connect the editor and step tree. The "Check Step Tree" button provides feedback on the step tree's status, categorized into five types (Figure 1.D). Clicking "Hint" offers progressive guidance based on learners' existing attempts (Figure 1.E). Hovering over steps shows buttons for editing the step tree (Figure 1.F). Figure 2 shows two key stages in DBox's workflow:

- **Solution Formation**: The step tree starts as an empty box where students can freely add steps in either coding mode (directly writing code) or description mode (building a step tree using natural language). They can evaluate their progress with the "From Editor to Step Tree" or "Check Step Tree" buttons. The tree contains steps and substeps labeled as *Correct*, *Incorrect*, *System Generated*, *Missing*, or *Can be Divided*. Layouts adjust dynamically based on the hierarchy. Steps that can be further divided are marked with dashed outlines, serving as a reminder, though students can decide whether further division is necessary.
- **Solution Implementation**: Students can convert the step tree into comments with "Copy to Comments" or verify alignment by clicking "Check Match". Nodes in the step tree are labeled as Implemented, Incorrectly Implemented, or To Be Coded. In this stage, DBox also offers progressive hints. When all nodes are implemented, students can test their solution by clicking "Run" button against the provided test cases.

## 4.2 Target Users and A System Walkthrough

DBox is designed for learners who understand basic algorithm concepts but struggle to apply them to solve practical problems. Using a scaffolding approach, DBox emphasizes independent thinking by offering only essential support. It assumes students are motivated, self-regulated, and actively engaging with the tool to improve their decomposition skills. If a student is less motivated or prefers a quicker solution, they may bypass DBox to search for answers online. Next, we present an example walkthrough (Figure 1) of such a self-regulated student Alice:

Alice, a learner tackling the "Search in Rotated Sorted Array" problem, begins by organizing her thoughts in the solution formation stage. DBox offers two options: she can either start coding or build a step tree using natural language. She opts for the latter and adds two initial steps (Figure 1.1). To check her progress, Alice clicks "Check Step Tree" button. DBox flags Step 1 as correct, Step 2 as incorrect, and highlights a missing Step 3 (Figure 1.2). She clicks the hint button on Step 2, receiving general and detailed guidance, but after another failed attempt, DBox offers another option for revealing a substep (Figure 1.3). Alice clicks "Reveal (Sub)Step", uncovering a sub-step 2-3 while leaving sub-steps 2-1 and 2-2 for her to solve (Figure 1.4). Inspired by the hints, Alice figures out how to break down and fills in these sub-steps (Figure 1.5). After checking again, Step 2 is marked correct (Figure 1.6). Alice adds the missing Step 3 (Figure 1.7), and finally, after checking, all steps turn to correct (Figure 1.8).

Next, Alice moves to the solution implementation stage. She clicks "Copy to Comments", and DBox converts her step tree into code comments (Figure 1.9). As Alice writes her code, she uses the "Check Match" button to identify incorrectly implemented and unimplemented steps. Step 2 is identified as unimplemented and Step 3 is identified as incorrectly implemented (Figure 1.10). Following DBox's guidance, she revises the code, and after another check, all steps turn to be correctly implemented (Figure 1.11). Satisfied with her progress, Alice clicks "Run" and successfully passes all test cases, solving the problem.

Note that we have presented only a simple walkthrough here, whereas the steps in a student's actual problem-solving process are more complex and dynamic (as shown later in Sec. 7.3). Next, we introduce the specific features aligned with the four design goals as described in Sec. 3.3.

## 4.3 Stage 1: Solution Formation

*4.3.1 Two Input Modes (D1, D2, D3).* DBox offers users the flexibility to develop their solutions through two distinct input modes: by writing code directly or by constructing a step tree using natural language descriptions, without needing to start with code. In the latter mode, users begin with a blank step tree and can click "Add" to insert nodes or "Split" to create sub-steps for more granular detail. Each node contains a text input field where users can articulate their thought process. Steps and sub-steps can be rearranged or deleted, allowing learners to iteratively and interactively refine and structure their mental model.

*4.3.2 Inferring Users' Thought Process from Existing Code (D1, D3).* The "From Editor to Step Tree" function in DBox infers a learner's intended solution and thought process based on their incomplete code. When activated, the system analyzes the code and problem, presenting the inferred steps as a tree on the right-hand side of the interface. Hovering over each node highlights the corresponding lines in the code editor, linking the inferred steps directly to the code. This feature assists users in diagnosing errors and identifying potential issues, especially when they are unsure how to proceed.

*4.3.3 Step Tree Node Status Evaluation with Preservation of Original Structure (D2, D4).* When the learner clicks "From Editor to Step Tree" or "Check Step Tree", DBox evaluates each node, assigning one of five statuses: (1) **Correct**: The step aligns with the learner's intended approach. (2) **Incorrect**: Errors are identified in the step. (3) **Missing**: A required step is absent. (4) **Can Be Divided**: The step is complex and can be broken into sub-steps, indicated by dashed borders. Users decide whether to subdivide. This status can coexist with other statuses. (5) **System Generated**: Step content is created by the system. This status is triggered only when the

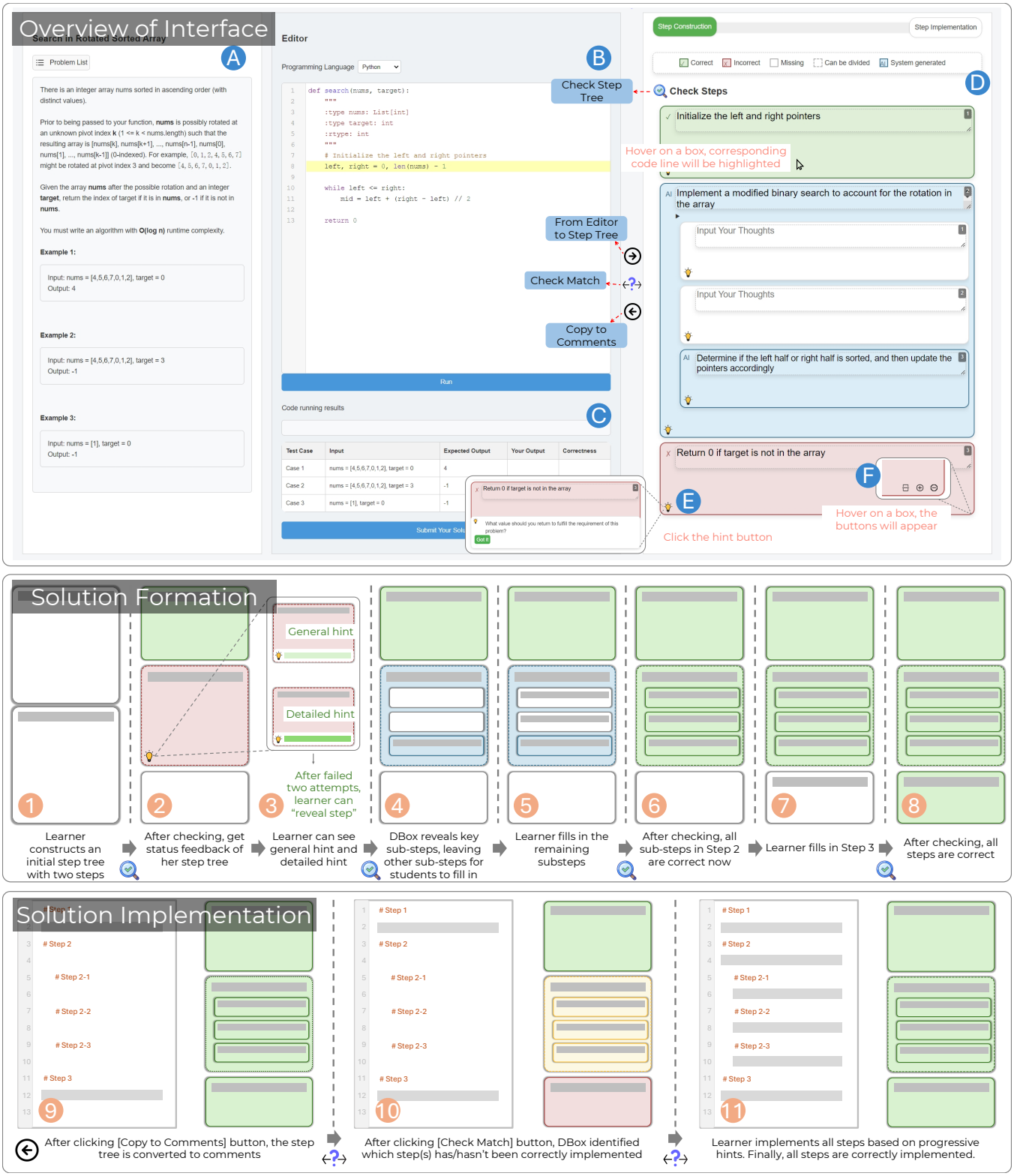


**Figure 1: The interface of Decomposition Box. The top row displays the full interface in the solution formation stage (the solution implementation stage is similar, but with different status indicators). The middle row demonstrates a learner's solution formation stage showing basic DBox features. The bottom row illustrates a learner's solution implementation stage. An overview of the DBox interface and workflow is provided in Sec. 4.1, and an illustrative example is described in Sec. 4.2. To save space, the second row omits the problem description and editor area, and the third row excludes the problem description area.**

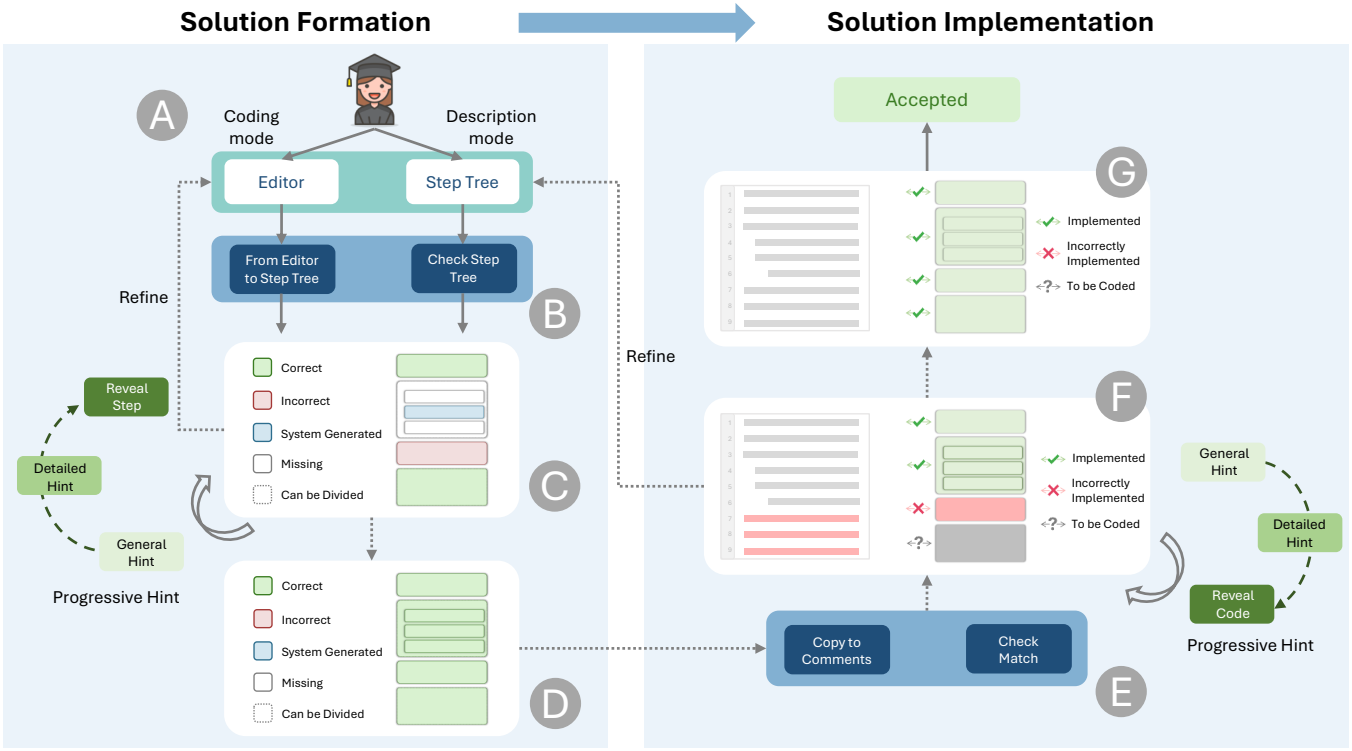


**Figure 2: The DBox workflow supports learners through solution formation and implementation stages. During solution formation, (A) students can input ideas by either coding or using natural language to build a step tree. (B) By clicking "From Editor to Step Tree" or "Check Step Tree", (C) DBox renders the step tree and identifies node statuses (e.g., correct, incorrect, missing). Students can iteratively refine their code or step tree, receiving progressive hints, (D) until the step tree is fully correct. In the solution implementation stage, (E) students can convert the step tree into code comments or (F) check the alignment between their code and the step tree. Each node displays one of three statuses, and students can refine their work with ongoing hints until (G) all nodes are marked as "implemented". Finally, students can test if their code passes all test cases.**

learner requests to reveal a (sub)step after repeated failures. During the "Check Step Tree" process, DBox preserves the original step tree (both structure and contents), only adding blank nodes for missing steps, ensuring scaffolding while respecting the learner's thought process.

*4.3.4 Progressive Hints for Solution Formation (D1).* DBox provides progressive hints to scaffold learners' problem-solving in three levels: (1) **General Hint** (Question-Based): Prompts learners' critical thinking without revealing solutions, e.g., "Before converting the string to an array, what should you do first?" (2) **Detailed Hint**: Offers more specific clues while requiring reasoning, e.g., "Think about how you can traverse each character in the string." (3) **Reveal (Sub)Step** ((Sub)Step Recommendation): For repeated errors, the AI can suggest a substep within a larger step when users click the "Reveal (Sub)Step" button. This reveals one key substep while leaving the remaining steps for the learner to complete. Notably, students can choose not to trigger this hint. These progressive hints support problem-solving development while allowing learners to maintain independence and control.

Once the step tree is complete and all nodes are correct, learners proceed to the solution implementation stage.

## 4.4 Stage 2: Solution Implementation

*4.4.1 Converting the Step Tree into Comments (D3).* This feature converts each node of the step tree into code comments. When students click "Copy to Comments", the system intelligently inserts these comments into the appropriate sections of the code editor. This guides learners to implement their solutions within the corresponding parts of their code, ensuring a smooth transition from planning to coding while reinforcing their structured approach.

*4.4.2 Validating Code Implementation against the Step Tree (D3, D4).* The "Check Match" button evaluates the alignment between the code and the step tree. Steps are categorized and color-coded as: (1) **Implemented**, (2) **Incorrectly Implemented**, and (3) **To Be Coded**. Hovering over a step highlights the corresponding lines in the code, providing a direct mapping between the step tree and the code to help users efficiently debug their implementation.

*4.4.3 Progressive Hints for Solution Implementation (D1).* For steps that are incorrectly implemented or yet to be coded, multi-level hints are available: (1) **General Hint**: Shows simple thought-provoking prompts/suggestions, e.g., "How should you correctly iterate until

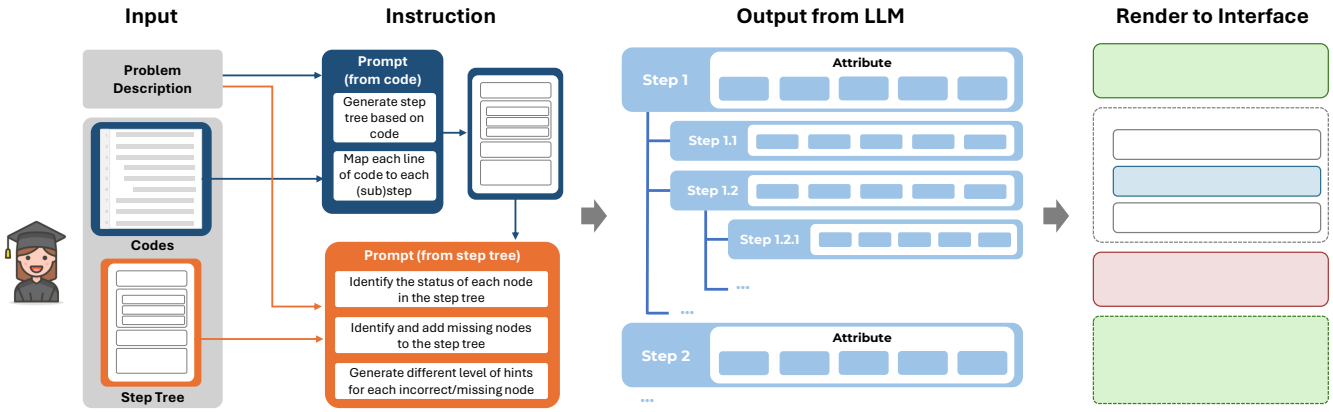


**Figure 3: An illustration of DBox's data processing workflow highlights its core function—creating a step tree with node statuses from student inputs. The LLM processes learners' incomplete code or a step tree they've constructed. It outputs a structured JSON object containing steps, sub-steps (and sub-sub-steps, etc.), each with several attributes. Then the JSON object is rendered to the interface, preserving the original structure and only adding nodes for any missing steps. Each node keeps the student's original input, without directly revealing the correct solution. DBox encodes the status of each step with colors and provides progressive hints.**

the second last character?" (2) **Detailed Hint** (Pseudocode): Provides simplified pseudocode to guide the user. (3) **Reveal Code** (Recommended Implementation): This option is activated only after two failed attempts. Clicking the "Reveal Code" button displays the recommended code implementation for the specific step.

## 4.5 Backend Design

DBox's backend is primarily powered by Large Language Models (the GPT-4o model specifically), with four distinct interactions corresponding to four buttons in the interface:

- **From Editor to Step Tree**: This button sends the problem description and the user's code to the LLM, which generates a step tree with nodes labeled as correct, incorrect, missing, or divisible.
- **Check Step Tree**: Clicking this button inputs the problem description and the user-constructed step tree into the LLM, which returns a labeled step tree with node statuses such as correct, incorrect, missing, or divisible.
- **Copy to Comments**: This button sends the problem description, current step tree, and user's code to the LLM, generating a mapping of step tree nodes to corresponding lines of code.
- **Check Match**: Pressing this button sends the problem description, step tree, and user's code to the LLM, which outputs a tree categorizing nodes as implemented, incorrectly implemented, or to be coded.

To illustrate, we present the data processing workflow for two core functions: "From Editor to Step Tree" and "Check Step Tree". Figure 3 shows how DBox processes student inputs in two modes: coding mode and language description mode (step tree input). Prompts adapt based on the input type. When only code is provided (dark blue lines in Figure 3), the system first calls [prompt from code] to generate a step tree, mapping each code line to a corresponding (sub)step based on the code's meaning. It then uses the generated step tree to invoke [prompt from step tree] (orange lines) to evaluate node statuses, add missing nodes, and generate multi-level hints for incorrect or missing nodes. If the input is a natural language step tree, the LLM directly calls [prompt from step tree] while preserving the original structure.

The LLM outputs a JSON containing steps, sub-steps, and subdivisions, each with attributes like student original input, status (e.g., correctness, completeness), LLM-validated content, and hints. The JSON is rendered conditionally, preserving the student's original structure while highlighting missing or incorrect steps. Even if a student's input differs from what the LLM considers correct, the original content is preserved and marked with a status color. Hints are provided and triggered progressively, offering "just the right" level of guidance.

## 4.6 Implementation

For the front-end, we used native HTML, JavaScript, and jQuery. On the back-end, we deployed the application with the Flask[1] framework on our university's server. The code editor utilized CodeMirror[2] integrated with pyodide.js[3] for running Python code. We employed OpenAI's GPT-4o model with a temperature of 0.8 to maintain flexibility during scaffolding. To align better with the front-end's step tree, we set the response format by setting the parameter response_format="type": "json_object", restricting the LLM's output. Prompts designed with the chain-of-thought (CoT) technique [78] are detailed in the supplementary materials.

# 5 Technical Evaluation

In DBox, the key feature is that learners construct a step tree using either code or natural language descriptions, while the LLM evaluates each step and provides necessary feedback. To assess whether

[1]https://flask.palletsprojects.com/en/stable/
[2]https://codemirror.net/
[3]https://pyodide.org/en/stable/

effective prompt engineering enables the GPT-4o model (hereafter referred to as GPT or LLM) to accurately determine node statuses (i.e., correct, incorrect, or missing), we conducted a preliminary technical evaluation. Detailed prompts used are provided in the supplementary material.

## 5.1 Dataset Creation

We created a dataset of learners' authentic thought errors to evaluate LLMs' ability to recognize the status of the thought process. Based on GPT-4's performance on coding tasks [20, 65], we selected 25 easy-level LeetCode problems covering various algorithms and data structures problems (e.g., dynamic programming, sorting, greedy algorithms).

To capture natural variations, we recruited five computer science students from a local university to manually create the step trees for five randomly selected problems (25 in total). Using correct code samples as references, annotators constructed a step tree based on their solution, including steps, substeps, and sub-substeps. They described each node in their own words and linked it to the relevant code. After collecting the annotated step trees, we manually created various types of errors to simulate common student misconceptions in coding [61]. An algorithmic programming expert created seven error types for each problem (e.g., missing steps, incorrect step order, logical errors, and syntax errors), which were reviewed by another expert, resulting in 175 error-laden step trees (25 problems x 7 error types).

## 5.2 Analysis Approach

We divide the steps into two parts based on the expert annotations: the correct part (1) or the incorrect/missing part (0). To calculate the performance of GPT, we use a very strict evaluation method: if all steps in the correct part are determined to be correct, the prediction of this part is marked as 1; otherwise, the prediction is 0. Similarly, if all steps in the incorrect/missing part are determined to be incorrect/missing, the prediction of this part is 0; otherwise, the prediction is 1. This approach enabled calculation of accuracy, F1 score, precision, recall, specificity, false positive rate, and false negative rate. We removed status fields from 175 error-containing step trees and input them into GPT for prediction. The results were compared against expert-annotated ground truth. For incorrect predictions, two authors independently coded GPT's outputs to identify error themes and causes, later consolidated through discussion.

## 5.3 Results

As shown in Table 1, **GPT more accurately identifies students' thought processes when expressed through code rather than natural language**. This difference may stem from GPT's extensive code-based training data [42] and specialized code-handling mechanisms [2], while natural language descriptions often include imprecise or non-standard terminology, leading to ambiguities [43].

**GPT sometimes identifies incorrect steps as correct (false positives) or correct steps as incorrect (false negatives)**. For *sequence change errors*, GPT's accuracy drops to 70% with an F1 score of 72% from natural language descriptions, with a False Positive Rate (FPR) of 36% and a False Negative Rate (FNR) of 24%. In contrast, code-based evaluations achieve 100% accuracy and F1 scores. For *logical errors* from natural language, GPT's accuracy is 88% (F1 score 87%, FPR 8%, FNR 16%), compared to 98% accuracy and F1 scores from code-based evaluations (FPR 4%, FNR 0%). *Missing step error* evaluations from natural language yield 86% accuracy (FPR 16%, FNR 12%), improving to 92% from code inputs (FPR/FNR 8%). *Syntax error* identification remains steady at 90% accuracy and F1 score.

Additionally, we find that **GPT occasionally alters the structure and content of the step tree**, despite instructions to only add missing steps. It sometimes modifies how steps are segmented or misinterprets the student's original input. Another key finding is that **GPT sometimes incorrectly judges non-standard approaches as wrong**. In five out of 25 tasks, GPT mistakenly flagged correct solutions as incorrect simply because they deviated from the common approaches in its training data. For example, it marked a sorting-based solution as incorrect, even though it was correct, albeit not the most optimal approach.

In summary, GPT demonstrates strong capabilities in processing code-based inputs but faces challenges with natural language, particularly in detecting sequence changes. Our analysis of participants' logs from the user study revealed that most errors encountered were logical errors or missing steps, while errors involving sequence changes were relatively uncommon. This suggests that GPT is generally effective in evaluating students' thought processes during algorithmic programming learning. We recommend that researchers considering GPT for supporting learners in natural language programming carefully evaluate its limitations and conduct technical assessments to determine its suitability for their specific scenarios. We hope this technical evaluation serves as a valuable reference for similar future research.

# 6 User Study

We conducted a user study to evaluate the effects of DBox, focusing on three questions:

- **Q1**: How does DBox support algorithmic programming learning?
- **Q2**: How does DBox affect learners' perceptions and user experience?
- **Q3**: How do learners interact with DBox and perceive the usefulness of different features?

## 6.1 Conditions

We conducted a within-subjects design to control for individual differences in programming abilities. Participants experienced two conditions in a randomly assigned order:

- **DBox**: Participants solved problems using the proposed DBox.
- **Baseline**: Participants freely used any available tools (e.g., ChatGPT, Copilot, search engines, LeetCode, QA platforms) to reflect their real-world learning habits, with no restrictions on tool usage or combinations.

## 6.2 Task and Materials

In this experiment, participants solve problems from two distinct algorithm types. Each type includes a learning problem, where participants use DBox or baseline tools, and a test problem, solved independently without assistance.

**Table 1: The technical evaluation of GPT-4o assesses its ability to identify the status of learners' steps. Precision refers to the proportion of steps correctly predicted as correct by GPT. TPR (True Positive Rate) measures the proportion of truly correct steps that GPT identifies correctly. TNR (True Negative Rate) reflects the proportion of truly incorrect/missing steps that GPT correctly predicts. FPR (False Positive Rate) indicates the proportion of incorrect/missing steps that GPT incorrectly predicts as correct. FNR (False Negative Rate) represents the proportion of correct steps that GPT incorrectly predicts as incorrect/missing.**

| Error Type | Accuracy | F1 | Precision | TPR/Recall | TNR/Specificity | FPR | FNR |
|---|---|---|---|---|---|---|---|
| **Identify step/substep status from learners' natural language-based step descriptions** | | | | | | | |
| **Sequence Changed** | 0.70 | 0.72 | 0.68 | 0.76 | 0.64 | 0.36 | 0.24 |
| **Logical Error** | 0.88 | 0.87 | 0.91 | 0.84 | 0.92 | 0.08 | 0.16 |
| **Missing** | 0.86 | 0.86 | 0.85 | 0.88 | 0.84 | 0.16 | 0.12 |
| **Identify step/substep status from learners' codes** | | | | | | | |
| **Sequence Changed** | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 0.00 | 0.00 |
| **Logical Error** | 0.98 | 0.98 | 0.96 | 1.00 | 0.96 | 0.04 | 0.00 |
| **Missing** | 0.92 | 0.92 | 0.92 | 0.92 | 0.92 | 0.08 | 0.08 |
| **Syntax Error** | 0.90 | 0.90 | 0.88 | 0.92 | 0.88 | 0.12 | 0.08 |

We selected problems from the LeetCode problem bank based on several criteria: First, all problems were of medium difficulty, with an acceptance rate between 40% and 50% to ensure sufficient challenge. Second, GPT performs well on these problems. Third, the two algorithm types are distinctly different to avoid learning effects. Finally, the learning and test problems within each algorithm type require similar programming skills to avoid unfair comparisons due to differences in additional coding skills needed for each problem. Based on these criteria, we chose two algorithm types: Greedy and Binary Search. For Greedy, we selected "Jump Game"[4] and "Jump Game II"[5]; for Binary Search, we selected "Search in Rotated Sorted Array"[6] and "Search in Rotated Sorted Array II"[7].

To help participants become familiar with or recall the algorithms used in the study, we provide them with lecture materials prior to the start of the study. The lecture materials for the two types of algorithms were sourced from GeeksforGeeks[8][9]. These materials include an introduction to each algorithm, illustrated figures, and practical examples.

## 6.3 Procedure

As shown in Figure 4, obtaining participants' consent, we begin by explaining the study's objectives and procedure. We then familiarize participants with the algorithms they will be practicing using the provided lecture materials. Note that the lecture material was designed to help participants recap the key concepts of these two algorithms. Although the two problems are labeled as Binary Search and Greedy on the LeetCode platform, participants were not restricted to using these two specific approaches to solve the problems. Afterwards, we administer a pre-test problem to assess their expertise. We also have participants rate their confidence in solving the problem on a 7-point Likert scale following [31]. After participants completed the task or indicated they could not proceed, the two authors (as experiment operators) assessed their solutions against pre-verified answers (with multiple solutions). Participants who solved the problem correctly were excluded, as it suggested higher expertise in the tested problem type. Following [31], we also excluded participants who rated their confidence at 6 or above, as high self-confidence likely indicates less need for additional support. While self-reported confidence may not always align with actual ability, this criterion helps focus the study on the intended user group for our tool. We then provided participants with an interactive tutorial to familiarize them with DBox. The tutorial guides them through each view, button, and functionality of the tool. After the tutorial, they can explore the tool by solving an exercise problem, different from the two problem types in the main study.

Next, we assigned experimental conditions and problem types to participants with a counterbalanced design. Within each problem type, one problem is randomly assigned in the learning session and the other in the testing session. In the learning session, participants use either DBox or baseline tools, during which they must successfully pass all test cases to proceed. They then complete an in-task survey about their perceptions and experience with the tool they just used. In the test session, participants solve problems without any tool assistance. After completing both problem types, they fill out a post-task survey, followed by a semi-structured interview.

## 6.4 Participants

We conducted a power analysis using G*Power [18] for our two-condition within-subjects design. Assuming an effect size of $f = 0.6$ (moderate), $\alpha = 0.05$, and power of 0.8, we estimated a required sample size of 24 participants.

After IRB approval, we recruited 24 participants via emails and social media at local universities (10 female, 14 male, average age 23.5, SD = 1.7). The group included 16 undergraduates and eight graduate students, with majors in computer science (17), data science (3), electrical engineering (3), and mathematics (1). Most participants (18) coded weekly, six coded monthly, and 23 had used platforms like LeetCode. The 90-minute study compensated participants with 20 USD, equating to 13 USD/hour.

[4]https://leetcode.com/problems/jump-game/description/
[5]https://leetcode.com/problems/jump-game-ii/description/
[6]https://leetcode.com/problems/search-in-rotated-sorted-array/description/
[7]https://leetcode.com/problems/search-in-rotated-sorted-array-ii/description/
[8]Greedy: https://www.geeksforgeeks.org/introduction-to-greedy-algorithm-data-structures-and-algorithm-tutorials/
[9]Binary Search: https://www.geeksforgeeks.org/binary-search/

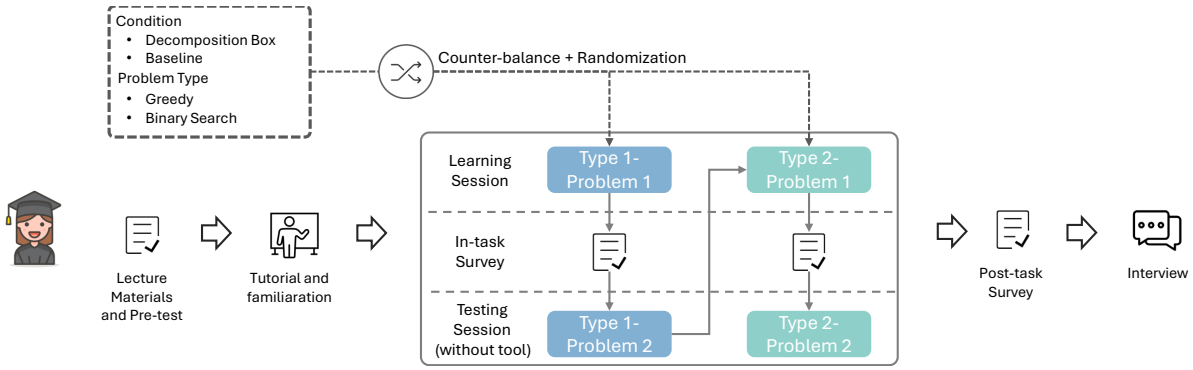


**Figure 4: The procedure of our user study. To avoid learning effects, we used a counterbalanced design with four combinations: (1) DBox-Type1 → Baseline-Type2, (2) Baseline-Type1 → DBox-Type2, (3) DBox-Type2 → Baseline-Type1, and (4) Baseline-Type2 → DBox-Type1. For each combination, six participants were randomly assigned.**

## 6.5 Measurements

Our measurements are summarized in Table 2. To address **Q1** (effects on learning outcomes), we assessed correctness in the testing session [34], perceived learning gain [84], confidence in solving similar problems [25], improvement in algorithmic thinking [82], and self-efficacy [74, 83]. For **Q2** (effects on perceptions and user experience), we measured cognitive engagement [59], critical thinking [32], sense of achievement [79], feeling of cheating [34], perceived help appropriateness, usefulness [14], mental demand, effort, frustration [24], ease of use, satisfaction [7], and future use intention [27]. For **Q3** (usage patterns and perceptions of DBox), we analyzed usage logs (e.g., clicks, edits, help-seeking), post-task feature ratings, and conducted semi-structured interviews to delve into participants' underlying reasons behind their perceptions, usage patterns, and reactions to AI errors. All questionnaires used a 7-point Likert scale.

## 6.6 Data Analysis

To eliminate the unfair comparison caused by the learning effect of participants using both tools to solve the same type of problem, we selected two distinct problem types. We implemented a randomization procedure to ensure that each participant used either DBox or the baseline tool in a random order, with a randomly assigned problem type for each tool. As a result, each participant used only one tool to solve one problem.

For the quantitative analysis, we employed a linear mixed effects model to analyze the data. The dependent variables (DVs) were our outcome measures (e.g., scores or questionnaire ratings). First, we analyzed the main effect of the two different tools (the coefficient and p-value were reported based on this analysis). Then, we examined the interaction effects between the learning tool and problem type (*Tool*Problem Type*), as well as the interaction effect between the learning tool and the order of tool usage (*Tool*Order*). The fixed effects in the models included the learning tool, problem type, and the order of tool usage, while the random effect accounted for individual differences between participants.

For the qualitative analysis of our semi-structured interview data, grounded in the designed questions, we conducted a thematic analysis [30]. Two authors independently coded the data, developed a codebook, and reached a consensus through discussion. In the results, we present key themes supported by representative participant quotes.

# 7 Results

In this section, we examine how DBox supports learners in algorithmic programming and how they interact with the tool. We compared DBox with the baseline tool and analyzed the interaction effects between tool and problem type, as well as the ordering effect (e.g., DBox first or Baseline first). Overall, we didn't find significant interaction or ordering effects for most metrics. Therefore, we report only the differences between the two tools unless notable interactions or ordering effects were observed, which are analyzed in detail.

## 7.1 How does Decomposition Box help with algorithmic programming learning?

We first compared the correctness scores of participants' test session submissions under both DBox and baseline conditions. As shown in Figure 5 (a), participants using DBox achieved significantly higher correctness scores than those in the baseline condition ($Coef.$=0.198, $p$<0.05), suggesting that practicing with DBox better prepared learners to transfer their skills to similar algorithmic challenges.

This finding aligns with participants' subjective perceptions. Figure 5 (b) shows that learners in the DBox condition reported significantly higher perceived learning gains ($Coef.$=2.250, $p$<0.001), higher confidence in solving similar problems ($Coef.$=2.333, $p$<0.001), more improvements in algorithmic thinking ($Coef.$=3.875, $p$<0.001), and greater self-efficacy ($Coef.$=3.042, $p$<0.001) compared to the baseline condition.

Interview analysis further highlighted that participants felt solving tasks independently during the learning session enhanced their perceived learning gains. Overcoming challenges on their own also boosted their confidence. In contrast, baseline participants felt their algorithmic thinking was underdeveloped due to easy access to complete solutions (e.g., via search, ChatGPT, or Copilot), leading

**Table 2: Measurements used in our user study. For the questionnaire items (within the quotation marks), a 7-point Likert scale was used, with 1 indicating "Strongly disagree/Very low" and 7 indicating "Strongly agree/Very high".**

| | Metrics | Detailed Meaning and Questions |
|---|---|---|
| Q1 | Correctness Score | The correctness of learners' test task solutions was evaluated using a consistent rubric from [34]. Two authors independently graded submissions, deducting 25% for each major issue or missing component, yielding scores of 0%, 25%, 50%, 75%, or 100%. They agreed on 87.5% of submissions, resolving disagreements through discussion for the rest. |
| | Perceived Learning Gain | "I have learned how to solve this type of problem." |
| | Confidence in Solving Similar Problems | "After solving this problem with the tool's help, I feel confident in tackling similar problems." |
| | Perceived Algorithmic Thinking Improvement | "This tool improved my ability to break down complex problems into smaller, manageable parts." |
| | Self-Efficacy | "I have mastered the problem-solving skills necessary for this type of problem." |
| Q2 | Cognitive Engagement | "I was cognitively engaged in the programming exercises." |
| | Critical Thinking | "The learning process challenged me to think critically." |
| | Sense of Achievement | "I feel a sense of accomplishment/achievement when I complete the programming task." |
| | Sense of Cheating | "Using this tool feels like cheating." |
| | Perceived Appropriateness of Help | "I felt I received the right amount of help when needed—neither too much nor too little." |
| | Perceived Usefulness | "This tool is useful for learning how to solve specific problems." |
| | Mental Demand | "How mentally demanding was the task?" |
| | Effort | "How hard did you have to work to achieve your level of performance?" |
| | Frustration | "How insecure, discouraged, irritated, stressed, and annoyed were you?" |
| | Ease of Use | "I find this tool easy to use for learning algorithms." |
| | Satisfaction | "I am satisfied with the overall learning experience using this tool." |
| | Future Use | "I would like to use this tool in my future programming learning." |
| Q3 | Button Clicking | (with timestamp) From Editor to Step Tree, Check Step Tree, Check Match, From Step Tree to Comments, Run Code |
| | Editing | (with timestamp) Code edit, step tree edit |
| | Help-Seeking | (with timestamp) see general hint, see detailed hint, and reveal step/code |
| | Usefulness Rating | Participants' ratings on the usefulness of various features in DBox using a 7-point Likert scale |
| | Interviews | Participants' detailed reasons for their perceptions, reactions to AI errors, and self-reported usage patterns, etc. |

to lower perceived learning and confidence. As P5 noted, "*Even though I couldn't write the full solution, the tool [DBox] encouraged me to break down the problem. I started with what I knew, and the tool guided me through the rest. Decomposing the problem helped me structure my approach, and as I saw the step tree fill in correctly, I felt my algorithmic thinking improve, and I gained confidence in solving the problem.*"

## 7.2 How does Decomposition Box affect learners' perceptions and user experience?

*7.2.1 Effects on Learners' Perceptions.* As shown in Figure 6, participants in the DBox condition reported significantly higher cognitive engagement ($Coef.$=2.625, $p$<0.001), greater critical thinking ($Coef.$=3.375, $p$<0.001), and a stronger sense of achievement ($Coef.$=3.375, $p$<0.001) compared to the baseline condition. Conversely, those in the baseline condition felt their problem-solving process resembled more "cheating" ($Coef.$=-4.250, $p$<0.001). DBox was also rated as providing more appropriate assistance ($Coef.$=2.625, $p$<0.001) and being more useful for programming learning ($Coef.$=2.792, $p$<0.001). Interviews supported these results, with 19 participants noting that DBox allowed them to independently develop their thought process while offering just enough feedback to guide them. This stimulated high engagement in problem-solving, as they critically analyzed both the overall solution and each step. As P1 noted, "*When I hit a block, unlike other tools (referring to the baseline), DBox didn't give me the answer outright, which forced me to think through the problem myself. Even with help, I still had to do most of the thinking.*"

In contrast, baseline participants using tools like ChatGPT, Copilot, or LeetCode often **bypassed independent thinking, focusing on comparing or copying provided answers**. Twelve out of 24 compared answers while coding, and eight simply copied solutions, leading to a lack of achievement and a sense of "cheating". As P16 (using ChatGPT) admitted, "*I tried to convert its provided code into my own, but I didn't feel like it was truly my solution; there was no sense of achievement.*" Besides, **excessive help in the baseline tools led participants to feel it was unhelpful for learning**. For example, P23 (who used LeetCode's built-in solution) shared, "*I was stuck on a small part, but the solution showed the entire answer immediately. I memorized it, but later, writing the code felt more like*

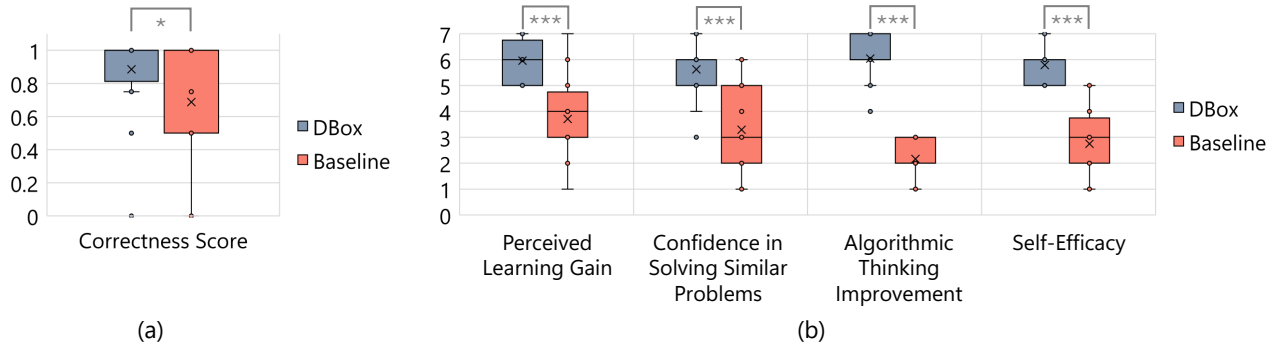


**Figure 5: Effects on participants' learning outcomes: (a) Participants' correctness scores during the testing session, where they solved the problem independently. (b) Participants' self-reported metrics on their learning outcomes.**

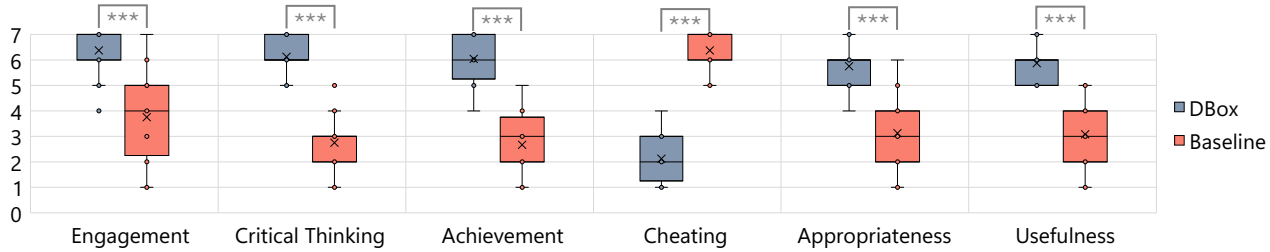


**Figure 6: Participants' perceptions of the two conditions in their learning process.**

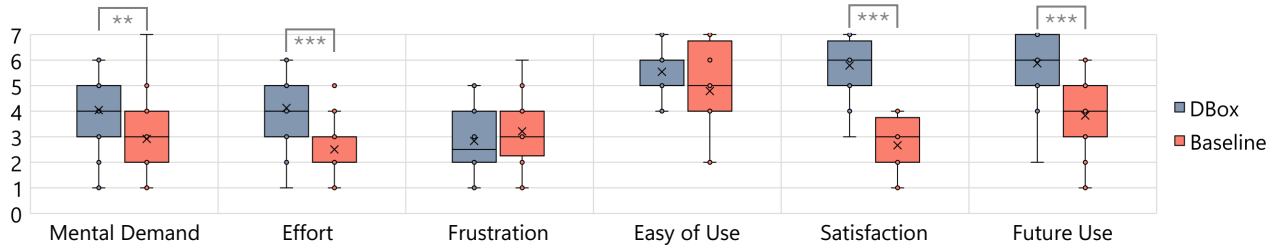


**Figure 7: Participants' cognitive load and user experience.**

*repetition than actual learning.*" P15 (using ChatGPT) added, "*ChatGPT explained the problem and gave the full code. While its solution seemed right, I realized I was just judging its correctness rather than improving my programming skills.*"

Moreover, we found an interaction effect between tool and problem type ($Coef.$=1.250, $p$<0.05) in perceived usefulness. Post-hoc analysis showed that DBox outperformed the baseline in both Binary Search ($t$ = 6.159, $p$ < 0.001) and Greedy problems ($t$ = 9.273, $p$ < 0.001). With DBox, there was no significant difference in perceived usefulness between the two problems ($t$ = 0.294, $p$ = 0.771). However, with the baseline, perceived usefulness was lower for the Greedy problem compared to Binary Search ($t$ = −2.755, $p$ < 0.05). We found no significant interaction effects on correctness scores, with participants showing similar performance across the two problems using either DBox or the baseline. This suggests that the lower perceived usefulness of the baseline for the Greedy problem was not due to the problem being inherently more difficult. A likely explanation is that the Greedy problem requires higher planning and decomposition skills (i.e., breaking a complex problem into subproblems solvable by a greedy algorithm) and the baseline did not provide scaffolding to support this, leading to lower perceived usefulness.

*7.2.2 Effects on Learners' User Experience.* As shown in Figure 7, participants in the DBox condition found the learning process more mentally demanding ($Coef.$=1.125, $p$<0.01) and reported putting in more effort ($Coef.$=1.625, $p$<0.001) than those in the baseline condition. This aligns with our expectations, as DBox requires learners to independently construct a step tree rather than merely providing solutions. Interestingly, there were no significant differences in frustration levels between the two conditions ($Coef.$=-0.375, $p$=0.305), suggesting that while DBox encouraged independent thinking and led to some failed attempts, the process of building the step tree did not cause excessive frustration. However, we found a significant

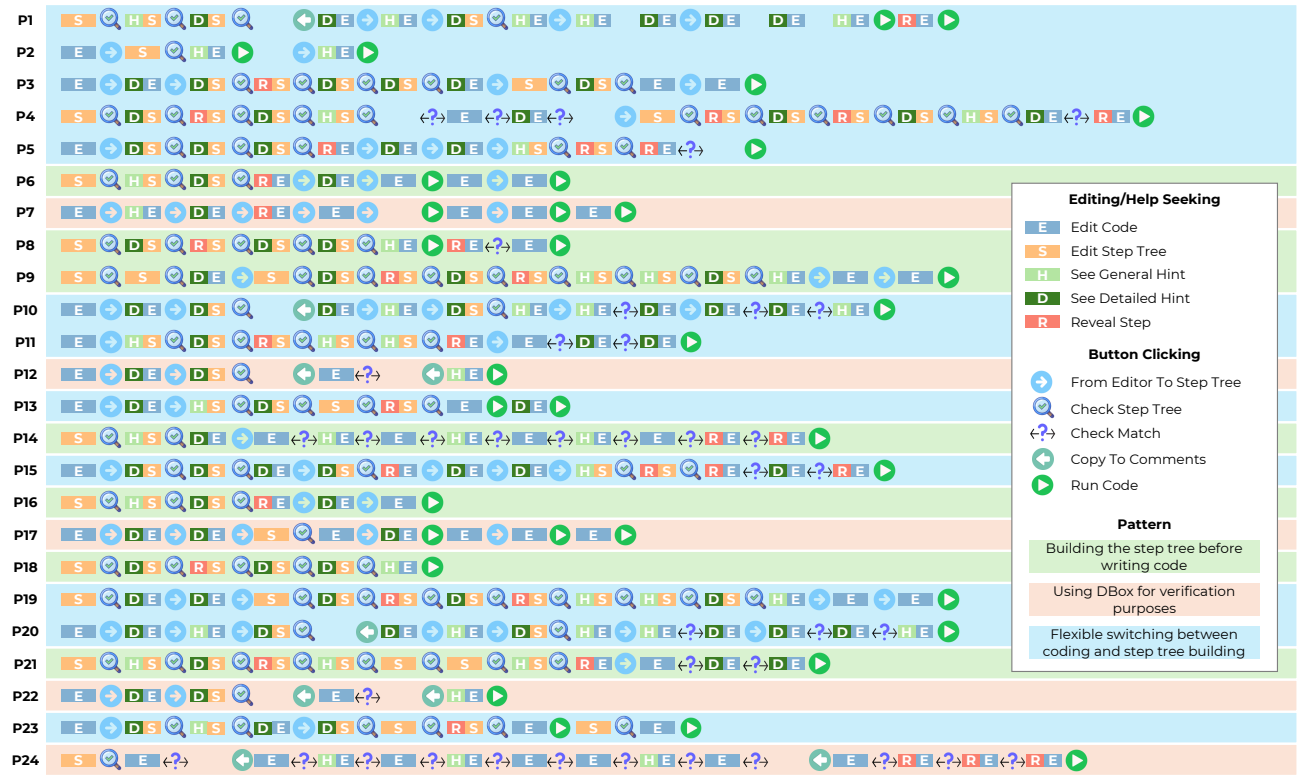


**Figure 8: Participants' three distinct types of system usage, each represented by a different color. We analyzed participants' interactions, including code editing, step tree editing, help-seeking, and five types of button clicks.**

ordering effect on frustration ($Coef.$=1.917, $p$<0.01). Post-hoc analysis showed no significant difference between the two tools when DBox was used first ($t$=1.186, $p$=0.248), but participants reported significantly higher frustration with the baseline when it was used first ($t$=2.491, $p$<0.05). One possible explanation is the "learning effect" of soliciting assistance – using DBox first better prepared participants to request targeted assistance from the baseline. This aligns how participants expressed frustration with the unsolicited assistance from the baseline. For example, P2 (using Copilot) commented, "*I typed a variable and hit [Tab], and Copilot suggested the entire code. However, the suggestion was different from what I had in mind, and I ended up spending time trying to understand it, which was frustrating.*"

We expected participants to find DBox less easy to use due to its more complex operations, but participants' perceived ease of use ($Coef.$=0.750, $p$=0.063) did not differ significantly between conditions. Interviews revealed that in the baseline condition, participants often had to switch between the editor and solution pages or spend time crafting precise prompts for ChatGPT. As P6 noted, "*I had to explain the problem and my understanding to ChatGPT, which was quite complex. I didn't just want to copy its answer, so I constantly did line-by-line comparison between ChatGPT-provided code and my code.*" Moreover, participants reported significantly higher satisfaction ($Coef.$=3.125, $p$<0.001) and a greater willingness to use DBox for future programming learning ($Coef.$=2.042, $p$<0.001).

## 7.3 How do learners interact with Decomposition Box and perceive the usefulness of different features?

*7.3.1 Learners' Overall Usage Patterns.* During the user study, we tracked participants' interactions with DBox, focusing on key actions such as code editing, step tree editing, help-seeking, and five main button clicks (e.g., From Editor to Step Tree, Check Step Tree, Check Match, Copy to Comments, and Run Code). These interactions are visualized in Figure 8, and we analyzed the patterns in combination with interview data.

Students adopted varying approaches when using DBox. Eleven began by constructing the step tree interactively, while thirteen started by writing code directly. We identified three distinct usage patterns:

**Type 1: Building the step tree before writing code.** Some participants (P6, 8, 9, 14, 16, 18, and 21) focused on constructing the step tree first, iteratively checking and refining it before moving on to code implementation. This approach enabled them to write code efficiently once the structure was finalized. As P21 noted, "*I used natural language to express my thoughts and verify them, and after a few iterations, I recognized valid ideas and wrote the code myself.*" P6 added, "*Writing code from scratch is more challenging for complex problems, so I prefer starting with the step tree.*"

**Type 2: Using DBox for verification.** Some participants (P7, 12, 17, 22, and 24) used DBox primarily to verify their code. They

wrote code first, then used the "From Editor to Step Tree" feature to check correctness and get hints. P17 explained, "*I usually solve problems by writing code first. Describing each step in natural language doesn't feel natural for me.*" Similarly, P22 stated, "*I know the general direction, so I write code first and use the tool to verify correctness or catch edge cases.*"

**Type 3: Flexible switching between the two modes.** Students like P1-5, 10, 11, 13, 15, 19, 20, and 23 alternated between coding and step tree construction, adjusting their approach based on confidence, familiarity with specific steps, and real-time coding challenges. For example, P19 stated, "*If I'm confident in certain steps, I code first and then convert it to steps for verification. If unsure, I verify my thought process before coding.*" P2 added, "*For familiar problems, I code first and refine it with the step tree. For new problems, I outline my thoughts and break down the steps to ensure accuracy before coding.*" P10 shared, "*Initially, I felt confident, but when I got stuck, I refined my understanding of a step in the step tree before continuing with the code.*"

*7.3.2 Hint Usage and Problem-Solving Approach.* We tracked hint usage frequency among all 24 participants, recording a total of 164 hints triggered. Only 32 instances (19.5%) involved the "reveal sub-step" feature, showing that students mainly relied on simpler hints and did not exploit the system by repeatedly making errors. This feature reveals only one sub-step from the user's incorrect or missing step—leaving the rest for them to solve independently—and can only be triggered after repeated struggle. This approach maximally preserves the students' independent problem-solving process. It strikes a balance between fostering independent thinking and preventing students from becoming permanently stuck, which could otherwise lead to frustration or a loss of motivation to learn.

We did a qualitative analysis which revealed varied problem-solving approaches adopted by participants in the problem-solving processes. For the "Can Jump" problem, tackled by 12 participants, we observed three approaches: Greedy (7 participants), Dynamic Programming (3), and Recursion (2). The other 12 participants addressed the "Search in Rotated Sorted Array" problem, using Binary Search (8), Two Pointers (2), Binary Tree (1), and Divide and Conquer (1). We observed that participants using the same approach still exhibited differences in reasoning and coding styles. Despite these variations, DBox effectively adapted its support to align with their individual styles and reasoning processes.

*7.3.3 Learners' Reactions to System Errors.* DBox, powered by the GPT-4o model, occasionally misjudged step statuses. During our user study, 24 participants triggered the "check" function ("Check Step Tree" and "From Editor to Step Tree") 208 times, with 16 participants encountering 18 system errors (an 8.7% error rate). Fourteen participants faced one error each, while two experienced two errors. These errors can be categorized into four types:

- **Type-1 (11 occurrences):** Misjudging correct steps as incorrect.
  - **Example:** A participant using a greedy approach stated, *"Use a greedy approach to minimize jumps."* GPT flagged this as incorrect due to insufficient detail on range expansion and jump counter updates. However, missing details does not necessarily mean the solution is incorrect.
- **Type-2 (2 occurrences):** Flagging unnecessary steps as missing.
  - **Example:** GPT incorrectly required a check for single-element arrays, though the solution worked without it.
- **Type-3 (3 occurrences):** Overlooking subtle mistakes in seemingly correct steps.
  - **Example:** A participant's wrote a step "Iterate through the array. If at any index `i`, `maxReach < i`, return false", leading to errors with unreachable indices. GPT failed to detect this error.
- **Type-4 (2 occurrences):** Missing crucial steps while DBox marking solutions as complete.
  - **Example:** A participant omitted a final return statement (`return true`), GPT still judged the solution as correct.

We then analyzed whether participants successfully identified the system errors and how they reacted to the errors. We found that all the 16 students who encountered system errors recognized these errors after one or more attempts. Since DBox evaluates rather than generates content, students remained confident in verifying their own work, which minimized over-reliance on the system. However, incorrect judgments, particularly when correct steps were flagged as wrong, could disrupt their thought process. The impact of these errors largely depended on the student's confidence. As P15 noted, "*If I know it's wrong, I ignore the message. If I'm unsure, I might follow the hint.*"

Students adopted various strategies to deal with system errors. (1) **Ignoring the error** (8 participants): Some simply moved on after recognizing a misjudgment. (2) **Focusing on the hint, not the status** (6 participants): Even if the status evaluation was incorrect, participants often found the hints useful. For example, participants wrote very brief steps, which GPT flagged as incorrect. The hints prompted them to consider important details, and despite the incorrect judgment, participants found the hints very useful. (3) **Running the code to verify** (12 participants): Many used the "Run" button to test their hypotheses. As P2 explained, "*If my code works, I stick with my approach. If it fails, I reconsider the system's judgment.*" (4) **Rechecking the step** (10 participants): Some participants revisited a step to refresh its status. As P3 mentioned, "*Even if the system is wrong, rechecking helps me identify potential issues.*"

Though system errors are unavoidable at the current stage, DBox's design allows learners to quickly recognize and manage them. Future improvements should aim to minimize disruptions caused by system inaccuracies.

*7.3.4 Learners' Ratings of Different Features of DBox.* DBox offers a range of features designed to enhance algorithmic programming learning, most of which participants found helpful, as illustrated in Figure 9. Students particularly appreciated the flexibility to either write code directly or build a step tree, and they valued DBox's ability to infer their thought process from the code. The interactive step tree and the fine-grained correctness assessment were also well-received, although two participants found it somewhat cumbersome. The progressive, multi-level hint system was praised by 22 participants. Additionally, the feature that checks the alignment between the step tree and the code implementation was seen as highly beneficial. However, some participants felt that the ability to paste the step tree as comments into the editor was unnecessary, since the step tree and code editor could already be viewed side by side. This feature could be even more useful if DBox were developed as a plugin in the future.

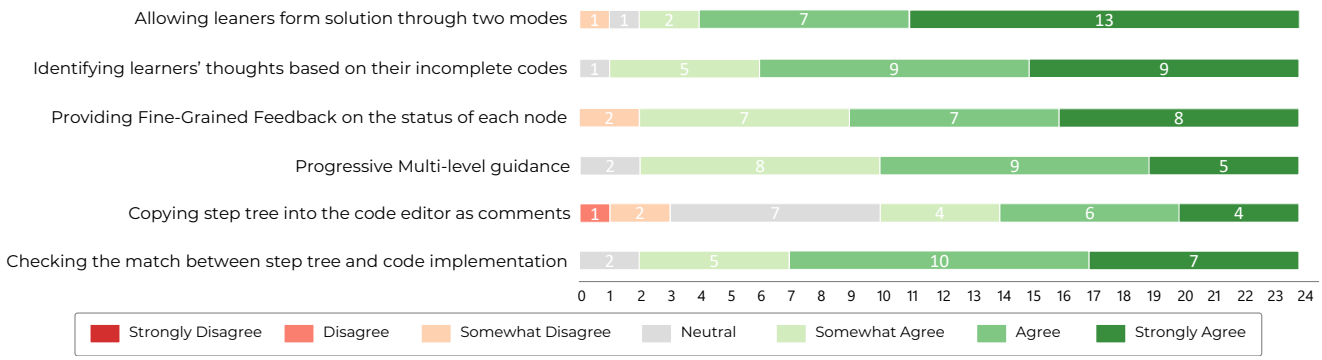


**Figure 9: Participants rated the features of DBox based on their firsthand experience during the experiment. In the questionnaire, they provided feedback from a first-person perspective (e.g., "I think [feature] is useful", rated on a 7-point Likert scale).**

## 8 Discussion

In this paper, we adopted a learner-centered design approach, beginning with a formative study to identify students' challenges with existing tools. Based on these insights, we developed DBox, a tool that scaffolds students in breaking problems into smaller parts and provides personalized, adaptive support. Our user study demonstrated that DBox improved learners' performance on similar algorithmic problems, increased perceived learning gains, and fostered greater cognitive engagement, achievement, and satisfaction. In this section, we discuss design implications and generalizability based on our key findings.

### 8.1 Chaining Learners' Thoughts with Visualized Structured UI Components

Decomposition requires students to effectively organize their thoughts. While visual elements are known to promote structured thinking and support mental model construction [44, 52], our formative and user studies revealed shortcomings in existing tools like LeetCode and ChatGPT, which rely on textual representations without adequately supporting structured mental models. In contrast, DBox uses an interactive step tree to visually organize learners' thoughts. This feature was praised by 22 of 24 participants for enhancing algorithmic thinking, serving as a progress tracker, and providing value even without AI assistance.

DBox's interactive step tree and tree-based scaffolding demonstrate the broader potential of intelligent tutoring systems (ITS) to promote active learning and self-regulated problem-solving in fields requiring problem decomposition. Similar principles could benefit STEM education, such as physics or engineering, by externalizing abstract concepts and facilitating multi-step problem-solving. Additionally, progress-tracking visual components may inspire designs for professional training tools in areas like medical diagnostics or software engineering.

### 8.2 Promoting Independent Thinking and Active Decomposition Learning

*8.2.1* ***Transforming Learners from Passive Readers to Active Thinkers.*** Many coding tools provide direct answers or solutions [35, 57], which, while efficient, often bypass opportunities to develop critical problem-solving skills. In contrast, DBox cultivates students' decomposition abilities through structured scaffolding, fostering critical thinking and self-regulated learning in line with learning by doing [5] and constructivist principles [73].

To strengthen decomposition skills, DBox first encourages students to develop their own decomposition strategies by coding or building a step tree from scratch. While DBox can generate parts of a step tree from a student's existing code, these steps are derived from the learner's own reasoning, with DBox acting solely as a modality converter. Besides, DBox provides feedback on tree node statuses, identifying potential errors or missing steps without directly showing the correct answer, challenging students to critically evaluate and refine their decomposition plans.

DBox's scaffolded hint system further supports decomposition skill development by providing adaptive guidance tailored to the student's progress without overwhelming them. All hints are based on the learner's current decomposition skeleton, with the most detailed hint—"reveal substep"—triggered only after repeated attempts and struggles. Notably, even the most detailed hints prompt only one substep, requiring students to complete the rest independently. As shown in Sec 7.3.2, only 19% of hints are this detailed, with students primarily relying on simpler, thought-provoking question hints. This scaffolded support system balances guidance and independent thinking, keeping students engaged during challenges without compromising their ability to independently decompose problems [39].

Based on these findings, we recommend fostering active problem-solving by shifting students from passive content consumption to active solution creation. Designers could adopt layered scaffolding, starting with minimal guidance and increasing support as needed, to help students progressively master decomposition skills while maintaining confidence and avoiding frustration. Additionally, adaptive learning techniques, such as real-time feedback and progress tracking, can further tailor the support to individual decomposition barriers, encouraging deeper engagement with decomposition tasks. Moreover, designers could integrate metacognitive strategies, such as encouraging students to articulate or reflect on their decomposition approaches, to further enhance critical thinking and foster habits of independent thinking.

*8.2.2* ***Choice of Scaffolding: Balancing Independent Problem-Solving and Efforts.*** Scaffolding involves providing tailored support to help learners accomplish tasks they cannot yet complete independently [38, 73]. Broadly, scaffolding strategies fall into two categories [75]: (1) gradually reducing assistance as learners gain proficiency, and (2) encouraging independent problem-solving while offering incremental support to address challenges. DBox adopts the second approach, emphasizing independent thinking and encouraging learners to actively decompose problems [85]. While our scaffolding strategies successfully enhanced critical thinking, satisfaction, and perceived usefulness, they also led to increased cognitive effort (Sec. 7.2.2). This tradeoff underscores the importance of carefully balancing cognitive effort with the promotion of independent thinking.

Future designs could incorporate adaptive scaffolding that adjusts support dynamically based on learner proficiency, reducing unnecessary effort in areas where students have demonstrated competence. Additionally, while incremental scaffolding was effective for algorithmic problem-solving, tailoring strategies to different educational contexts could enhance their applicability in diverse domains. Such adaptive, context-specific approaches could further optimize the balance between support and independence in learning environments.

## 8.3 Supporting Personalized Algorithmic Programming Learning

*8.3.1* ***Prioritizing Learners' Own Solutions Over Optimality.*** Algorithmic problems often have multiple solutions with varying time and space complexities. DBox prioritizes independent exploration by supporting learners' strategies rather than steering them toward a single "optimal" solution. Using LLM-driven prompts, it evaluates and guides each step based on the learner's reasoning, preserving their step decomposition and respecting their input—even when errors occur. While some solutions may not be the most efficient, this approach fosters autonomy by aligning feedback with learners' thought processes instead of enforcing rigid standards.

Our user study showed that this approach improves learning outcomes and is well-received by students. We recommend designing systems that respect personalized problem-solving strategies by aligning feedback with learners' reasoning while allowing for diverse approaches. Designers should balance flexibility and rigor, using prompts and interfaces that support varied strategies while gently guiding learners toward effective solutions.

*8.3.2* ***Catering to Individual Learning Styles and Contextual Needs.*** DBox accommodates diverse problem-solving approaches with two input modes: coding and natural language descriptions. Each mode offers distinct advantages tailored to different learners, stages, and situations. Learners can switch seamlessly between modes, with progress automatically synced across the interface. Features such as verifying code-step alignment ensure strong integration between modes.

Our findings reveal that this flexibility enhances user experience. Participant interaction logs and interviews revealed three usage patterns, highlighting that each mode fits different needs: code mode works well for students with a clear and detailed problem-solving plan already, while the step tree with natural language descriptions helps less experienced students with only a basic idea who are not ready to write code directly, boosting their confidence.

We argue there is no universal "best" mode for programming education—each has unique benefits depending on the learner habits, expertise, and context. Future tools should provide flexibility, like DBox, or use adaptive algorithms to recommend modes based on user needs and context. This flexibility highlights the importance of designing educational tools that accommodate varying levels of expertise and problem-solving styles, which can be generalized to other domains requiring personalized learning [8].

## 8.4 Appropriate Usage of LLMs for Supporting Algorithmic Programming Learning

*8.4.1* ***Caution About LLM Errors.*** Although LLMs have shown strong performance in coding tasks [20, 40], they remain prone to errors. Our technical evaluation and user study revealed that even with comprehensive context—such as problem statements, user code, and natural language steps—LLM sometimes misinterprets user descriptions. These errors likely arise from discrepancies between the natural language used by students and the formal, precise language the LLM was trained on, which is primarily sourced from web-based code and comments [43].

Such misinterpretations can hinder learning by causing confusion or frustration. While future improvements to training data and GPT versions may mitigate these issues, design strategies can help address them. **First**, LLMs should avoid giving direct solutions and instead focus on fostering active problem-solving through explanations and hints. **Second**, feedback could be paired with interactive features, like a "Run Code" option, allowing students to validate their reasoning. **Third**, simple tutorials could teach users how to phrase their descriptions more clearly, improving LLM's understanding. Additionally, future tools could integrate a "Language Enhancement" feature to suggest improvements or assess the clarity of descriptions, aiding LLM in accurately capturing user intent. Most importantly, we recommend designers prioritize technical feasibility, such as conducting rigorous evaluations like ours, before fully integrating LLMs into programming learning tools.

*8.4.2* ***Learner-LLM Co-Decomposition of Solutions: Learner as Leader, LLM as Aid.*** A central feature of DBox is the construction of a step tree, where students break solutions into steps and sub-steps. The LLM supports this by mapping code to step descriptions, evaluating them, and offering hints. However, students maintain full control, deciding how to decompose problems and define each step, fostering independent thinking. The LLM acts solely as an aid, using a scaffolding approach to support the development of learners' Zone of Proximal Development (ZPD) [9]. Unlike tools like ChatGPT or Copilot that dominate problem-solving, DBox fosters deeper cognitive engagement. Students reported greater accomplishment and found this approach more effective for learning.

This contrasts with existing human-AI collaboration paradigms in non-educational scenarios where AI usually suggest options, leaving final decisions to users [13, 22, 23, 48, 49], such as in human-AI decision-making [45–47]. Some educational tools, like Jin et al. [31], use LLMs to generate solutions for students to evaluate, which aids in syntax learning but such "LLM-generate then learner-evaluate" approach is less effective for algorithmic problem-solving, where

constructing solutions is key. Just evaluating LLM-generated contents can place a cognitive anchor on learners [21], limiting independent thinking and creativity. Thus, task allocation between humans and AI should align with the educational context (e.g., whether it is basic knowledge/concept learning or higher-level creative thinking). Future LLM-based educational tools should carefully define the division of roles between LLMs and learners, tailoring it to specific learning contexts and goals.

### 8.5 Limitations and Future Work

This study has several limitations. *First*, we tested DBox's effectiveness on only two problem types; future work should examine a broader range of algorithms. *Second*, participants engaged in just one learning session per condition due to time constraints, whereas mastering algorithmic problems typically requires extended practice. Longitudinal studies should explore how DBox supports skill development over time, including changes in mental models and skill retention. *Third*, we assessed learning gains based on correctness in a test session using similar learning and test problems. Future research should evaluate knowledge transfer to less similar problems. Due to time constraints, we conducted a single post-test rather than a pre-post comparison. While pre-test expertise filtering and randomization minimized prior familiarity effects, a more rigorous pre-post design would yield more accurate learning gain measurements. Looking ahead, we plan to release DBox as a Chrome plugin for integration with existing coding platforms, enabling large-scale field studies. This will allow for the collection of long-term usage data and periodic surveys to identify usage patterns and learning experiences over time.

## 9 Conclusion

In this paper, we introduced DBox, an interactive tool designed to help learners decompose algorithmic programming problems by supporting both solution formation and implementation. Featuring an intuitive tree-like box widget, DBox accepts input in both code and natural language, fostering independent problem-solving while its step tree structure helps learners develop structured mental models. It provides step-level feedback and layered guidance without compromising learner autonomy. Our user study showed that DBox significantly improved learning outcomes, cognitive engagement, and critical thinking, with students reporting a greater sense of achievement and finding the support highly effective. Additionally, we identified three key usage patterns, highlighting the importance of accommodating individual problem-solving styles. Moreover, our findings suggest that the learner-LLM co-decomposition approach fosters independent thinking while providing meaningful guidance, even with occasional imperfections. We hope the insights from our system design will inspire future research on integrating LLMs into educational tools for programming learning.

## Acknowledgments

This research was supported by the Dieter Schwarz Stiftung Foundation, ETH Foundation, and in part by the EdUHK-HKUST Joint Centre for Artificial Intelligence (JC_AI) research scheme: Grant No. FB454.

## References

[1] Khan Academy. 2024. Code Tutor. https://chatgpt.com/g/g-HxPrv1p8v-code-tutor Accessed: September 10, 2024.
[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).
[3] Vincent AWMM Aleven and Kenneth R Koedinger. 2002. An effective metacognitive strategy: Learning by doing and explaining with a computer-based cognitive tutor. *Cognitive science* 26, 2 (2002), 147–179.
[4] Charoula Angeli, Joke Voogt, Andrew Fluck, Mary Webb, Margaret Cox, Joyce Malyn-Smith, and Jason Zagami. 2016. A K-6 computational thinking curriculum framework: Implications for teacher knowledge. *Journal of Educational Technology & Society* 19, 3 (2016), 47–57.
[5] Yuichiro Anzai and Herbert A Simon. 1979. The theory of learning by doing. *Psychological review* 86, 2 (1979), 124.
[6] Roland Backhouse. 2011. *Algorithmic problem solving*. John Wiley & Sons.
[7] Aaron Bangor, Philip T Kortum, and James T Miller. 2008. An empirical evaluation of the system usability scale. *Intl. Journal of Human–Computer Interaction* 24, 6 (2008), 574–594.
[8] Matthew L Bernacki, Meghan J Greene, and Nikki G Lobczowski. 2021. A systematic review of research on personalized learning: Personalized by whom, to what, how, and for what purpose (s)? *Educational Psychology Review* 33, 4 (2021), 1675–1715.
[9] Seth Chaiklin et al. 2003. The zone of proximal development in Vygotsky's analysis of learning and instruction. *Vygotsky's educational theory in cultural context* 1, 2 (2003), 39–64.
[10] Michael Cole, Vera John-Steiner, Sylvia Scribner, and Ellen Souberman. 1978. Mind in society. *Mind in society the development of higher psychological processes. Cambridge, MA: Harvard University Press* (1978).
[11] Cristina Conati and Kurt Vanlehn. 2000. Toward computer-based support of meta-cognitive skills: A computational framework to coach self-explanation. *International Journal of Artificial Intelligence in Education* 11 (2000), 389–415.
[12] Kathryn Cunningham, Barbara J Ericson, Rahul Agrawal Bejarano, and Mark Guzdial. 2021. 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*. 1–15.
[13] Hai Dang, Sven Goller, Florian Lehmann, and Daniel Buschek. 2023. Choice over control: How users write with large language models using diegetic and non-diegetic prompting. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–17.
[14] Fred D Davis. 1989. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS quarterly* (1989), 319–340.
[15] Erhan Delen, Jeffrey Liew, and Victor Willson. 2014. Effects of interactivity and instructional scaffolding on learning: Self-regulation in online video-based environments. *Computers & Education* 78 (2014), 312–320.
[16] 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.
[17] Paul Denny, Sami Sarsa, Arto Hellas, and Juho Leinonen. 2022. Robosourcing Educational Resources–Leveraging Large Language Models for Learnersourcing. *arXiv preprint arXiv:2211.04715* (2022).
[18] Franz Faul, Edgar Erdfelder, Axel Buchner, and Albert-Georg Lang. 2009. Statistical power analyses using G* Power 3.1: Tests for correlation and regression analyses. *Behavior research methods* 41, 4 (2009), 1149–1160.
[19] James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. 2022. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference*. 10–19.
[20] James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A Becker. 2023. My ai wants to know if this will be on the exam: Testing openai's codex on cs2 programming exercises. In *Proceedings of the 25th Australasian Computing Education Conference*. 97–104.
[21] Adrian Furnham and Hua Chu Boo. 2011. A literature review of the anchoring effect. *The journal of socio-economics* 40, 1 (2011), 35–42.
[22] Jie Gao, Yuchen Guo, Gionnieve Lim, Tianqin Zhang, Zheng Zhang, Toby Jia-Jun Li, and Simon Tangi Perrault. 2024. CollabCoder: a lower-barrier, rigorous workflow for inductive collaborative qualitative analysis with large language models. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–29.
[23] Simret Araya Gebreegziabher, Zheng Zhang, Xiaohang Tang, Yihao Meng, Elena L Glassman, and Toby Jia-Jun Li. 2023. Patat: Human-ai collaborative qualitative coding with explainable interactive rule synthesis. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–19.
[24] Sandra G Hart. 2006. NASA-task load index (NASA-TLX); 20 years later. In *Proceedings of the human factors and ergonomics society annual meeting*, Vol. 50.

Sage publications Sage CA: Los Angeles, CA, 904–908.
[25] Heris Hendriana, Tri Johanto, and Utari Sumarmo. 2018. The Role of Problem-Based Learning to Improve Students' Mathematical Problem-Solving Ability and Self Confidence. *Journal on Mathematics Education* 9, 2 (2018), 291–300.
[26] Cindy E Hmelo-Silver, Ravit Golan Duncan, and Clark A Chinn. 2007. Scaffolding and achievement in problem-based and inquiry learning: a response to Kirschner, Sweller, and. *Educational psychologist* 42, 2 (2007), 99–107.
[27] Richard J Holden and Ben-Tzion Karsh. 2010. The technology acceptance model: its past and its future in health care. *Journal of biomedical informatics* 43, 1 (2010), 159–172.
[28] 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.
[29] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79.
[30] Hsiu-Fang Hsieh and Sarah E Shannon. 2005. Three approaches to qualitative content analysis. *Qualitative health research* 15, 9 (2005), 1277–1288.
[31] Hyoungwook Jin, Seonghee Lee, Hyungyu Shin, and Juho Kim. 2024. 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*. 1–28.
[32] Carol S Kamin, Patricia S O'Sullivan, Monica Younger, and Robin Deterding. 2001. Measuring critical thinking in problem-based learning discourse. *Teaching and learning in medicine* 13, 1 (2001), 27–35.
[33] Enkelejda Kasneci, Kathrin Seßler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, et al. 2023. ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and individual differences* 103 (2023), 102274.
[34] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI code generators on supporting novice learners in introductory programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–23.
[35] Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara Jane 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. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*. 1–12.
[36] Majeed Kazemitabaar, Jack Williams, Ian Drosos, Tovi Grossman, Austin Zachary Henley, Carina Negreanu, and Advait Sarkar. 2024. Improving steering and verification in AI-assisted data analysis with interactive task decomposition. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. 1–19.
[37] Aaron Keen and Kurt Mammen. 2015. Program decomposition and complexity in CS1. In *Proceedings of the 46th ACM technical symposium on computer science education*. 48–53.
[38] Minchi C Kim and Michael J Hannafin. 2011. Scaffolding problem solving in technology-enhanced learning environments (TELEs): Bridging research and theory with practice. *Computers & Education* 56, 2 (2011), 403–417.
[39] Päivi Kinnunen and Lauri Malmi. 2006. Why students drop out CS1 course?. In *Proceedings of the second international workshop on Computing education research*. 97–108.
[40] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A Becker. 2023. Using large language models to enhance programming error messages. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 563–569.
[41] Marcia C Linn and John Dalbey. 1985. Cognitive consequences of programming instruction: Instruction, access, and ability. *Educational Psychologist* 20, 4 (1985), 191–206.
[42] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).
[43] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D Gordon. 2023. "What it wants me to say": Bridging the abstraction gap between end-user programmers and code-generating large language models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–31.
[44] Zhicheng Liu and John Stasko. 2010. Mental models, visual reasoning and interaction in information visualization: A top-down perspective. *IEEE transactions on visualization and computer graphics* 16, 6 (2010), 999–1008.
[45] Shuai Ma, Qiaoyi Chen, Xinru Wang, Chengbo Zheng, Zhenhui Peng, Ming Yin, and Xiaojuan Ma. 2024. Towards human-ai deliberation: Design and evaluation of llm-empowered deliberative ai for ai-assisted decision-making. *arXiv preprint arXiv:2403.16812* (2024).
[46] Shuai Ma, Ying Lei, Xinru Wang, Chengbo Zheng, Chuhan Shi, Ming Yin, and Xiaojuan Ma. 2023. Who Should I Trust: AI or Myself? Leveraging Human and AI Correctness Likelihood to Promote Appropriate Trust in AI-Assisted Decision-Making. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–19.
[47] Shuai Ma, Xinru Wang, Ying Lei, Chuhan Shi, Ming Yin, and Xiaojuan Ma. 2024. " Are You Really Sure?" Understanding the Effects of Human Self-Confidence Calibration in AI-Assisted Decision Making. *arXiv preprint arXiv:2403.09552* (2024).
[48] Shuai Ma, Zijun Wei, Feng Tian, Xiangmin Fan, Jianming Zhang, Xiaohui Shen, Zhe Lin, Jin Huang, Radomír Měch, Dimitris Samaras, et al. 2019. SmartEye: assisting instant photo taking via integrating user preference with deep view proposal network. In *Proceedings of the 2019 CHI conference on human factors in computing systems*. 1–12.
[49] Shuai Ma, Taichang Zhou, Fei Nie, and Xiaojuan Ma. 2022. Glancee: An Adaptable System for Instructors to Grasp Student Learning Status in Synchronous Online Classes. In *CHI Conference on Human Factors in Computing Systems*. 1–25.
[50] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. 2023. Experiences from using code explanations generated by large language models in a web software development e-book. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 931–937.
[51] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. 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*. 125–180.
[52] Siné JP McDougall, Martin B Curry, and Oscar De Bruijn. 2001. 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 (2001), 59–84.
[53] Roy D Pea. 1987. Logo programming and problem solving. (1987).
[54] Roy D Pea. 2018. The social and technological dimensions of scaffolding and related theoretical concepts for learning, education, and human activity. In *Scaffolding*. Psychology Press, 423–451.
[55] Roy D Pea and D Midian Kurland. 1984. On the cognitive effects of learning computer programming. *New ideas in psychology* 2, 2 (1984), 137–168.
[56] Janice L Pearce, Mario Nakazawa, and Scott Heggen. 2015. Improving problem decomposition ability in CS1 through explicit guided inquiry-based instruction. *J. Comput. Sci. Coll* 31, 2 (2015), 135–144.
[57] 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).
[58] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. 2015. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the second (2015) acm conference on learning@ scale*. 195–204.
[59] Nicole P Pitterson, Shane Brown, Jason Pascoe, and Kathleen Quardokus Fisher. 2016. Measuring cognitive engagement through interactive, constructive, active and passive learning activities. In *2016 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–6.
[60] James Prather, Brent N Reeves, Paul Denny, Brett A Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. "It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. *ACM Transactions on Computer-Human Interaction* 31, 1 (2023), 1–31.
[61] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)* 18, 1 (2017), 1–24.
[62] Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education* 27 (2017), 37–64.
[63] Lianne Roest, Hieke Keuning, and Johan Jeuring. 2024. Next-Step Hint Generation for Introductory Programming Using Large Language Models. In *Proceedings of the 26th Australasian Computing Education Conference*. 144–153.
[64] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*. 27–43.
[65] Jaromir Savelka, Arav Agarwal, Marshall An, Chris Bogart, and Majd Sakr. 2023. Thrilled by your progress! large language models (gpt-4) no longer struggle to pass assessments in higher education programming courses. In *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1*. 78–92.
[66] Henk G Schmidt, Sofie MM Loyens, Tamara Van Gog, and Fred Paas. 2007. Problem-based learning is compatible with human cognitive architecture: Commentary on Kirschner, Sweller, and. *Educational psychologist* 42, 2 (2007), 91–97.

[67] Brad Sheese, Mark Liffiton, Jaromir Savelka, and Paul Denny. 2024. Patterns of student help-seeking when using a large language model-powered programming assistant. In *Proceedings of the 26th Australasian Computing Education Conference*. 49–57.

[68] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 15–26.

[69] Steven S Skiena. 1998. *The algorithm design manual*. Vol. 2. Springer.

[70] Renske Smetsers-Weeda and Sjaak Smetsers. 2017. Problem solving and algorithmic development with flowcharts. In *Proceedings of the 12th Workshop on Primary and Secondary Computing Education*. 25–34.

[71] Raja Sooriamurthi. 2009. Introducing abstraction and decomposition to novice programmers. *ACM SIGCSE Bulletin* 41, 3 (2009), 196–200.

[72] Edward R Sykes. 2010. Design, Development and Evaluation of the Java Intelligent Tutoring System. *Technology, Instruction, Cognition & Learning* 8, 1 (2010).

[73] Sigmund Tobias and Thomas M Duffy. 2009. Constructivist instruction. *Success or failure* (2009).

[74] Chun-Yen Tsai. 2019. Improving students' understanding of basic programming concepts through visual programming language: The role of self-efficacy. *Computers in Human Behavior* 95 (2019), 224–232.

[75] Janneke Van de Pol, Monique Volman, and Jos Beishuizen. 2010. Scaffolding in teacher–student interaction: A decade of research. *Educational psychology review* 22 (2010), 271–296.

[76] Kurt VanLehn. 2011. The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational psychologist* 46, 4 (2011), 197–221.

[77] Wengran Wang, Audrey Le Meur, Mahesh Bobbadi, Bita Akram, Tiffany Barnes, Chris Martens, and Thomas Price. 2022. Exploring design choices to support novices' example use during creative open-ended programming. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education-Volume 1*. 619–625.

[78] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[79] Susan Wiedenbeck, Deborah Labelle, and Vennila NR Kain. 2004. Factors affecting course outcomes in introductory programming.. In *PPIG*. 11.

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

[81] David Wood, Jerome S Bruner, and Gail Ross. 1976. The role of tutoring in problem solving. *Journal of child psychology and psychiatry* 17, 2 (1976), 89–100.

[82] Mustafa Yağcı. 2019. A valid and reliable tool for examining computational thinking skills. *Education and Information Technologies* 24, 1 (2019), 929–951.

[83] Hatice Yildiz Durak. 2018. Digital story design activities used for teaching programming effect on learning of programming concepts, programming self-efficacy, and participation and analysis of student experiences. *Journal of Computer Assisted Learning* 34, 6 (2018), 740–752.

[84] Xiaohua Zhou, Ching Sing Chai, Morris Siu-Yung Jong, and Xi Bei Xiong. 2021. Does relatedness matter for online self-regulated learning to promote perceived learning gains and satisfaction? *The Asia-Pacific Education Researcher* 30, 3 (2021), 205–215.

[85] Barry J Zimmerman. 2013. Theories of self-regulated learning and academic achievement: An overview and analysis. *Self-regulated learning and academic achievement* (2013), 1–36.