SimStep: Multi-Level Abstractions for Incremental Specification and Debugging of AI-Generated Educational Simulations

Zoe Kaputa
Stanford University
USA
kaputa@stanford.edu

Anika Rajaram
The Harker School
USA
26anikar@gmail.com

Vryan Almanon Feliciano
Stanford University
USA
vgfelica@stanford.edu

Zhuoyue Lyu
Cambridge University
UK
zl536@cam.ac.uk

Maneesh Agrawala
Stanford University
USA
maneesh@cs.stanford.edu

Hari Subramonyam
Stanford University
USA
harihars@stanford.edu

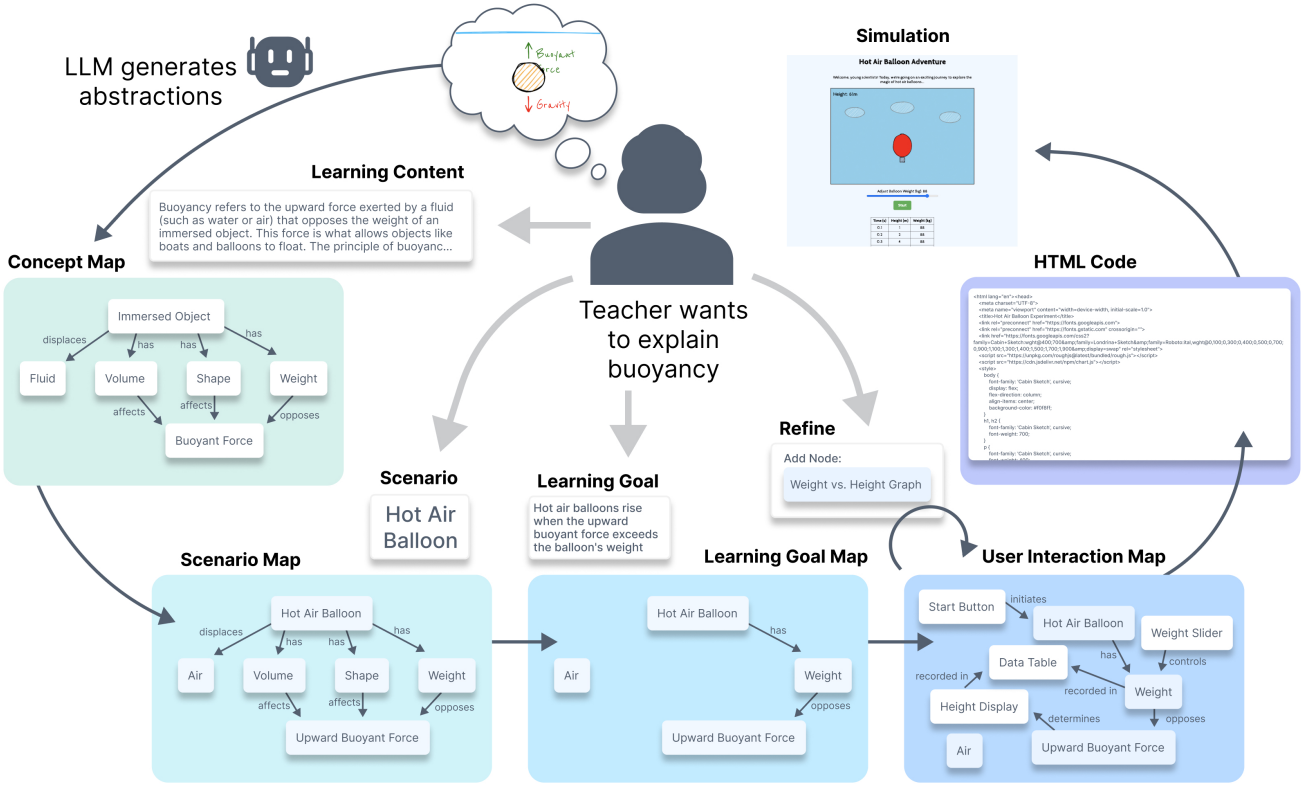

Figure 1: A teacher can use SimStep to generate interactive simulations that are accurate, integrate students' context, and reflect straightforward learning goals.

Abstract

Programming-by-prompting with generative AI offers a new paradigm for end-user programming, shifting the focus from syntactic fluency to semantic intent. This shift holds particular promise for non-programmers such as educators, who can describe instructional goals in natural language to generate interactive learning content. Yet in bypassing direct code authoring, many of programming's core affordances—such as traceability, stepwise refinement, and behavioral testing—are lost. We propose the *Chain-of-Abstractions*

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

(CoA) framework as a way to recover these affordances while preserving the expressive flexibility of natural language. CoA decomposes the synthesis process into a sequence of cognitively meaningful, task-aligned representations that function as checkpoints for specification, inspection, and refinement. We instantiate this approach in SimStep, an authoring environment for teachers that scaffolds simulation creation through four intermediate abstractions: Concept Graph, Scenario Graph, Learning Goal Graph, and UI Interaction Graph. To address ambiguities and misalignments, SimStep includes an inverse correction process that surfaces in-filled model assumptions and enables targeted revision without requiring users to manipulate code. Evaluations with educators show that CoA enables greater authoring control and interpretability in programming-by-prompting workflows.

ACM Reference Format:
Zoe Kaputa, Anika Rajaram, Vryan Almanon Feliciano, Zhuoyue Lyu, Maneesh Agrawala, and Hari Subramonyam. 2018. SimStep: Multi-Level Abstractions for Incremental Specification and Debugging of AI-Generated Educational Simulations. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 27 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Programming-by-prompting with generative AI promises to democratize code creation [29]. It shifts the focus from writing *syntactically* correct instructions to expressing *semantically* meaningful intent [50]. For non-technical experts such as educators, this approach is potentially transformative. Rather than learning formal programming constructs, teachers can describe goals, concepts, and exemplar scenarios in natural language and create rich interactive learning experiences for their students [8]. For instance, a teacher might prompt *"give me a simulation to teach about buoyancy by showing hot-air balloons rise and fall,"* and receive a functional interactive simulation aligned with their lesson (see Figure 1). Here the authoring interface is natural language itself allowing teachers to express program specifications using the same language and vocabulary they use to plan lessons, explain ideas, and promote active student engagement.

However, in abstracting away the syntactic interface for natural language, something essential is lost. Traditional syntactic programming interfaces have certain affordances. Every behavior must be explicity specified, requiring *users* to resolve ambiguities, consider conditionals and alternatives, and translate high level goals into concrete operational logic. It reveals gaps in understanding and makes assumptions visible. Prompting, by contrast, allows ideas to remain *diffuse* and critical details are often left unstated [51]. The generative AI model must infer what is missing and resolve underspecifications, and those inferences may not always be aligned with goals and intents. In the previous example, the prompt about buoyancy simulation might yield a visually appealing animation, yet omit essential causal relationships, misrepresent scientific principles, or fail to support student interactions that deepen understanding.

Further, regardless of the interface, certain core programming tasks—such as testing behavior, debugging errors, and refining algorithms—remain. Syntactic interfaces afford traceability and control by exposing the program's structure through elements such as function definitions, control flow, and variable states, enabling the programmer to trace and debug the behavior step-by-step. In the buoyancy simulation, the programmer can inspect the exact buoyancy equation governing the motion of the hot-air balloon or modify conditional logic to assess edge cases. In contrast, natural language *collapses* these structures into textual abstractions, making it difficult to isolate errors, reason about behavior, or understand how changes to the prompt affect the generated code.

Existing approaches to alleviate these challenges largely cluster around three main strategies, each targeting different aspects of the prompt-to-code process. First, *prompt engineering* techniques focus on improving the prompt quality through carefully worded instructions, structured templates, or few-shot examples that guide and constrain the model's interpretation [62]. While often helpful, prompt engineering places a cognitive burden to envision how the model "thinks," which can be especially challenging for non-technical users such as teachers [51, 57]. Second, *post-generation debugging* approaches allow users to refine code either by mapping back the code to the prompt [37], providing dynamic interfaces for editing [4], or directly inspecting and editing the generated code to correct errors [56]. A third strategy, reasoning trace externalization, includes techniques such as *chain-of-thought* [60] and *chain-of-verification* [17], which aim to improve model performance and interpretability by generating intermediate steps in natural language. While often described as revealing the model's "reasoning," these techniques do not expose internal symbolic computations or structured deliberation, but rather simulate step-by-step patterns that improve alignment with desired outputs [38]. As such, they operate within linguistic scaffolding, offering no direct support for users to specify or manipulate the underlying system behavior.

As an alternative, we propose that programming-by-prompting should not solely rely on linguistic interactions and abstractions. But instead, it should be supported by **task-level representational abstractions** that externalize and structure user intent. This approach is grounded in the theory of distributed cognition, which holds that complex tasks are accomplished not by internal reasoning alone, but through interactions with external representations that guide and transform cognitive work [28]. In our context, these task abstractions (1) align with natural task decomposition, and (2) act as cognitive checkpoints—distinct stages where users can interpret, manipulate, and refine system behavior. The checkpoints allow users to verify and refine behavior, test assumptions, and refine outcomes, all without requiring formal programming.

In this paper, we introduce SimStep, a system that operationalizes this approach through a chain-of-abstractions (CoA) framework. Grounded in the task of authoring educational simulations, SimStep guides educators through a structured sequence of *domain-aligned* representations—including a Concept Graph, Scenario Graph, Learning Goal Graph, and UI Interaction Graph—each designed to surface and refine different dimension of the intent (Figure 4). These representations allow users to incrementally clarify and operationalize their ideas, while also providing structure for the model to reason from. An *underspecification resolution engine* further supports this process by identifying ambiguous or missing elements, enabling users to *inspect* model assumptions, *test* behavior in a *guided* manner, and iteratively *steer* the generation toward their instructional goals. At every step, SimStep keeps the human in the loop, as an active participant in shaping interpretations, validating assumptions,

and steering the generation. Our approach recovers key affordances of traditional programming such as traceability, testability, and control through accessible, task-specific abstractions.

Our key contributions are: (1) a conceptual framework that repositions programming-by-prompting as authoring through task-level abstractions, emphasizing the role of human-in-the-loop reasoning and correction; (2) the design and implementation of SimStep, an instantiation of this framework that enables educators to incrementally specify, test, and revise AI-generated simulations without writing code; and (3) technical and empirical evaluation demonstrating how SimStep supports pedagogical alignment, reduces authoring complexity, and provides control in content creation.

2 Conceptual Framework for CoA

To ground prompt-to-code authoring as a human-in-the-loop process, we formalize our approach using a distributed cognition lens [28]. Distributed cognition theory views cognitive processes as extending beyond individual minds, operating across people, artifacts, and representational media. In Hutchins' study of ship navigation, for example, navigational charts, instruments, and logs serve not merely as information displays, but as cognitive artifacts that structure and distribute reasoning over time and across individuals [28].

Our Chain-of-Abstractions (CoA) framework adopts this perspective by treating intermediate representations not simply as steps in a pipeline but as structured cognitive checkpoints where reasoning is transformed and shared between humans and AI. Each abstraction in the CoA supports distinct types of cognitive work—such as articulating domain knowledge, defining causal structure, or specifying interface logic—and affords both interpretability for the human and tractability for the model. What qualifies this decomposition as a form of distributed cognition is not merely that the process is staged, but that each stage enables coordination between agents (human and AI), externalizes internal thought processes, and provides a representational substrate for validation, revision, and semantic control. In this sense, CoA reflects a system of representations that scaffolds joint reasoning across agents, aligning with core principles of distributed cognitive systems.

2.1 Human Guided Forward Transformations

Let P denote a user's initial natural language prompt and C the final executable code. Between them lies a sequence of representational abstractions $\mathcal{A} = \{A_1, A_2, ..., A_n\}$, each representing a task checkpoint where intent is transformed, clarified, and refined. In prior work, several key types of abstractions have been proposed including concept graphs, scene graphs, task decomposition structures, etc. [6, 43, 61, 67]. These abstractions are derived through transformations $T = \{T_1, ..., T_{n+1}\}$, and we propose they involve a collaboration between human agents (H) and machine agents (M) powered by large language models. We formalize the collaborative transformation process as:

$$P \xrightarrow{T_1^{(M)}} A_1 \xrightarrow{\text{Refine}^{(H)}} A_1' \xrightarrow{T_2^{(M)}} A_2 \xrightarrow{\text{Direct}^{(H)}} A_2' \xrightarrow{\cdots} C$$

Each abstraction A_i serves as a *task checkpoint* that exposes the model's current understanding of the task. Functionally, each abstractions is characterized by its *visibility*—how observable and legible its structure is to the user, its *manipulability*—how easily the user can adjust or modify its components, and its *fidelity*—how accurately it encodes the user's goals or task-relevant system logic. These properties collectively determine the abstraction's effectiveness in supporting meaningful human oversight and intervention within the generative workflow.

Further, we define four core *forward operations* that the human agent H performs at each checkpoint A_i:

- **Inspect**: Examine the abstraction A_i to understand what the model has inferred, identify mismatches with intent, and diagnose possible errors or oversights.
- **Refine**: Modify A_i to produce A_i', correcting assumptions, adding missing components, adjusting relationships, or removing irrelevant elements.
- **Validate**: Assess whether A_i' meets instructional or behavioral goals, ensuring semantic coherence, completeness, and alignment with domain expectations.
- **Direct**: Provide guidance for the next transformation $T_{i+1}^{(M)}$ by specifying priorities, constraints, or features that must be preserved in downstream abstractions.

2.2 Inverse Correction

The forward transformation pipeline produces a sequence of intermediate abstractions from a natural language prompt. Each abstraction A_i represents a progressively refined interpretation of user intent, narrowing ambiguity and structuring information for downstream synthesis. Ideally, underspecified details would be surfaced and resolved at appropriate points along this chain. However, in practice, generative models often defer filling in those details until the final stages, particularly at the code level where the model must commit to specific values, logic, or visual behavior. This can lead to implicit assumptions being introduced without the user's awareness, as those decisions were not made explicit in earlier abstractions.

While introducing intermediate abstractions helps reduce ambiguity and support human-in-the-loop correction, it also introduces a design tradeoff. Increasing the number of abstractions, or lowering their level of specificity, can overwhelm users with too many representational layers or expose them to implementation-level detail. For instance, block-based environments like Scratch [39] provide visual structure but often require users to reason about program flow, variable state, and low-level operations. This reintroduces syntactic and procedural complexity, defeating the purpose of semantic-level prompting. In contrast, our approach favors task-level abstractions that align with how users naturally organize their thinking, aiming for a balance between expressive control and cognitive accessibility.

However, to recover and revise these hidden assumptions, we introduce a complementary **inverse process**, which includes a new set of *targeted* abstractions $\mathcal{B} = \{B_1, B_2, ..., B_m\}$. The same abstraction X may exist in both $\mathcal{A}$ and $\mathcal{B}$, but these sets can also include unique abstractions. The inverse correction process begins by realizing the B_i that is most suitable for correcting the system's assumption. Then, the user can **Inspect**, **Refine**, and **Validate** B_i as described for abstractions in $\mathcal{A}$. After producing B_i', a direct

translation $T^{(M)}$ exists to transform B_i' into C'.

$$C \xrightarrow{\text{Realize}} B_i \xrightarrow{\text{Refine}^{(H)}} B_i' \xrightarrow{T^{(M)}} C'$$

This revision occurs at the abstraction level rather than the code level, allowing users to correct high-level misunderstandings without needing to inspect low-level syntax. Let $\Omega(X)$ denote the set of all valid implementations (e.g., simulation code) that are compatible with a representation X, and let $U(X)$ represent the degree of underspecification in X. If a representation is vague or abstract, $\Omega(X)$ will be large, indicating that many different implementations are possible, and $U(X)$ will be high, reflecting the ambiguity of the representation.

As the synthesis pipeline progresses from the natural language prompt P to the final code C, each transformation incrementally reduces both the ambiguity and the number of compatible implementations:

$$\Omega(P) \supset \Omega(B_1) \supset \Omega(B_1') \supset \cdots \supset \Omega(C)$$

$$U(P) > U(B_1) > U(B_1') > \cdots > U(C)$$

This formalism reflects how intent becomes increasingly concrete. Note that this process is not intended to surface all hidden details but to expose and resolve only those underspecifications that result in incorrect behavior. It turns abstractions into bi-directional interfaces that can be used for forward synthesis as well as targeted recovery.

3 User Experience

To realize the CoA framework in Section 2, we developed SimStep, an tool that allows educators to author interactive simulations through a human-guided, step-by-step approach. When designing a simulation, a human expert typically approaches this task in stages: (1) identifying the core scientific concepts to teach, (2) searching for and mapping how they relate with potential experimental scenarios, (3) concretely defining the situated learning goals, and (4) considering the space of hypothesis to realize those goals and how learners will interact with the system. Based on this understanding, SimStep's interface follows a wizard-style design pattern using the four main stages in the task, providing appropriate representational abstractions in each stage. To explore the specific authoring and testing features, let us follow Mr. Carlos, a high school science teacher, as he creates a simulation using SimStep.

3.1 CoA Code Generation

Mr. Carlos is working on his lesson plan to teach about *buoyancy* and wishes to use an interactive simulation to promote active student engagement. Mr. Carlos opens SimStep on his web browser, which shows him the authoring interface with the *text input step* open (Figure 2a). This step has a text input box on the left and a collapsible panel to display the concept graph on the right. Using the prescribed science textbook for the course, Mr. Carlos copies the text about core principles of buoyancy and pastes it in the text box. Based on this text, SimStep automatically generates a **Concept Graph**—a visual model made up of nodes representing key concepts like Object's Weight, Buoyant Force B, Fluid Density ρ, and Displaced Volume V, connected by edges that encode relationships and equations (e.g., $m = \rho_{\text{object}} \times V$, $W = m \times g$). As Carlos **Inspects** the graph, he notices an issue: Buoyant Force is linked directly to Object's Mass, skipping the required relationship with Fluid Density and Displaced Volume. Using the graph editor, he deletes the incorrect link and adds two new nodes and connecting links to represent the correct equation (i.e., **Refine**): $B = \rho \times V \times g$. Once the concept graph accurately reflects the scientific model, Carlos proceeds to the next step.

In the second step, Mr. Carlos is prompted to specify a desired experimental scenario (just like a human expert would do), which serves as the contextual foundation for generating the simulation code. To facilitate scenario grounding, SimStep uses the conceptual model to generates and displays a set of potential scenarios for teaching about buoyancy (Figure 2c). Mr. Carlos notices that one of the scenarios is using *hot air balloon*, and realizes that since his students recently saw hot air balloons rise at the annual city festival, it would be a great way to connect the abstract concept of buoyancy to something they had all experienced. Alternatively, Mr. Carlos can define his own scenario using the text input box, tailored to his understanding of his students. Once Mr. Carlos provides a scenario, system generates a **Scenario Graph**, a situation model representation (Figure 2d) in which all the nodes and links in the conceptual model are instantiated with the context of hot air balloon. As before, Carlos **Inspects** the instantiated graph and proceeds to the learning goal selection by clicking the 'Next' button.

The learning goal selection panel offers Mr. Carlos a range of objectives tailored to the chosen scenario. For instance, SimStep might presents several objectives around (1) conceptual understanding focusing on key concepts and describing phenomenon (e.g., *understand the relationship between air temperature inside the balloon and its buoyancy, and be able to describe how equilibrium is achieved*) , causal or explanatory understanding (e.g., *explain how the decrease in air density inside the balloon leads to an increase in buoyancy, allowing the balloon to rise*), and procedural knowledge (e.g., *effects of different heating methods on the rate of the balloon's ascent and procedural relationship between gradual heating and smooth altitude control*). On this panel (Figure 2e), Mr. Carlos can select a learning goal to *direct* next steps in the code generation, and as before explore the generated **Learning Goal Graph** which is derived from the situation model. From here, Mr. Carlos simply clicks the 'Next' Button to see the final interactive simulation (Figure 2f). To generate the simulation, SimStep uses the learning goals sub-graph and provides an **Interactivity Graph** including all of the controls and how controls link to behavior of elements in the situation model to meet the intended learning goals.

3.2 Interactive Debugging and Refinement

While the forward authoring path provides several affordances for aligning steering code generation, potential errors can still emerge in realizing the final interaction graph as simulation code, i.e., syntactic inference errors. The errors might include misaligned layout elements or mislabeled controls, errors where the model fills in unspecified details like slider ranges or animation timings incorrectly; or errors where UI elements fail to trigger the expected behaviors due to missing or flawed logic. To support testing, debugging and correcting these issues, SimStep offers several features including guided testing, and widget based error correction.

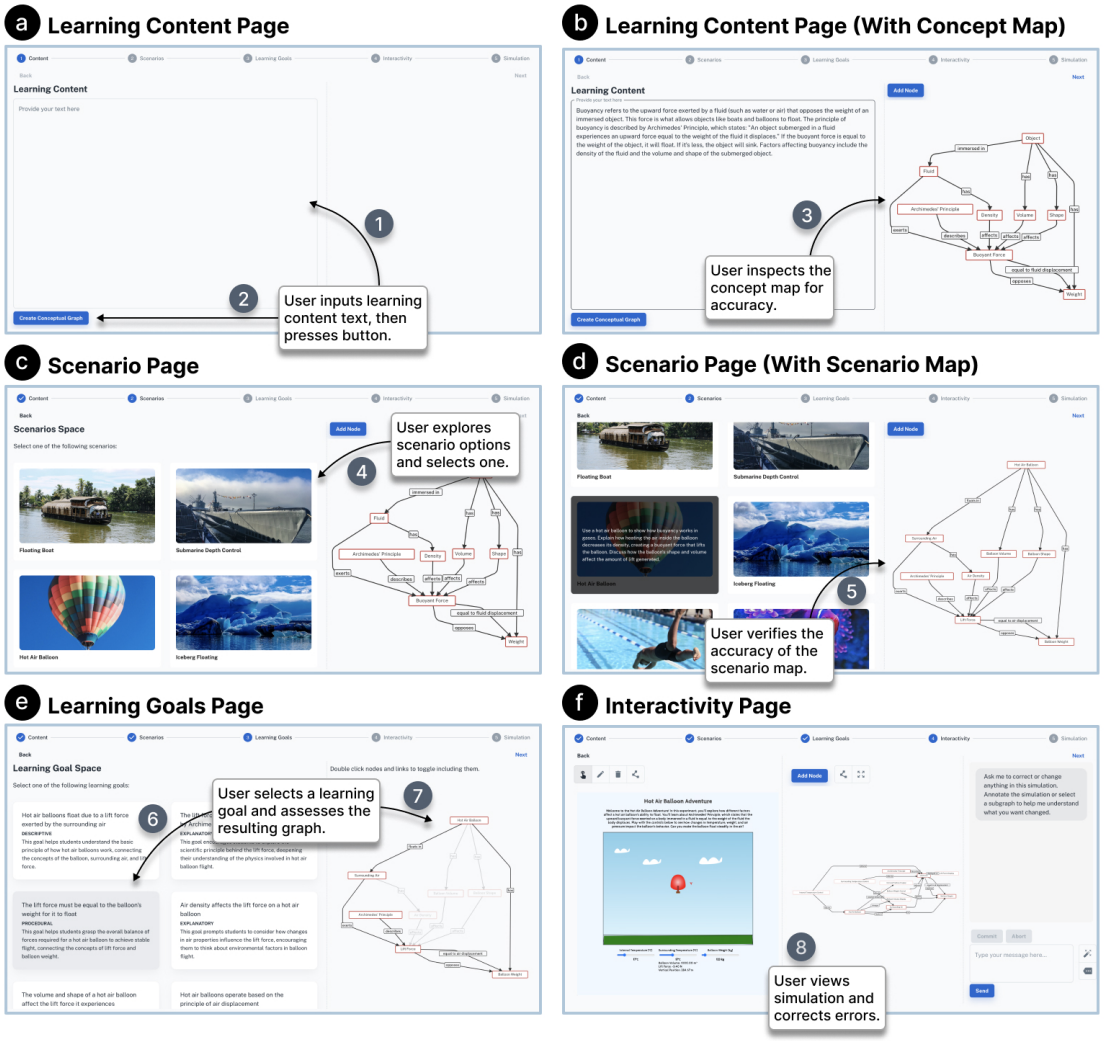

Figure 2: The main screens of the SimStep application include a Learning Content Page (a and b), a Scenario Page (c and d), a Learning Goals Page (e), and an Interactivity Page (f).

3.2.1 Guided Testing and Automated Repair: SimStep support a form of guided UI testing in which it automatically generates test cases from the abstraction pipeline and simulates learner (end-user) interactions within the final simulation UI. This feature was informed by our user study (Section 5). For instance, as seen in Figure 3b, in the buoyancy simulation, the system executes scripted interactions—such as adjusting the temperature slider or releasing the balloon—and prompts for Mr. Carlos's feedback. Specifically, Mr. Carlos is asked to evaluate whether the observed behavior aligns with his instructional intent, such as whether increasing temperature causes the balloon to rise faster. His response indicates whether the test has passed or failed, enabling it to refactor the code to fix any issues through human feedback.

3.2.2 Manual Debugging and Repair. Beyond guided testing and automated repair, Mr. Carlos can directly inspect the simulation on his own and identify points of misalignment between intents and simulation behavior. Even with state-of-the-art models, underspecification in earlier stages can result in missing or incorrect logic. SimStep provided affordances for interactive debugging and refinement through a rich chat based interface along with direct annotations on the generated simulation and interaction graph (Figure 2f). For instance, Mr. Carlos can circle around a region of interest on the simulation and inspect the relevant nodes in the interaction graph which is filtered automatically based on the annotation, and then use natural language to describe the problem. For example, he might observe that adjusting the weight slider does not affect the balloon's altitude, and describe the correction in terms of missing connections between relevant concepts.

To support such targeted correction, SimStep implements an underspecification resolution engine that interprets the context of his chat message and generates prefilled widgets that represent candidate fixes at the appropriate level of abstraction. Mr. Carlos can

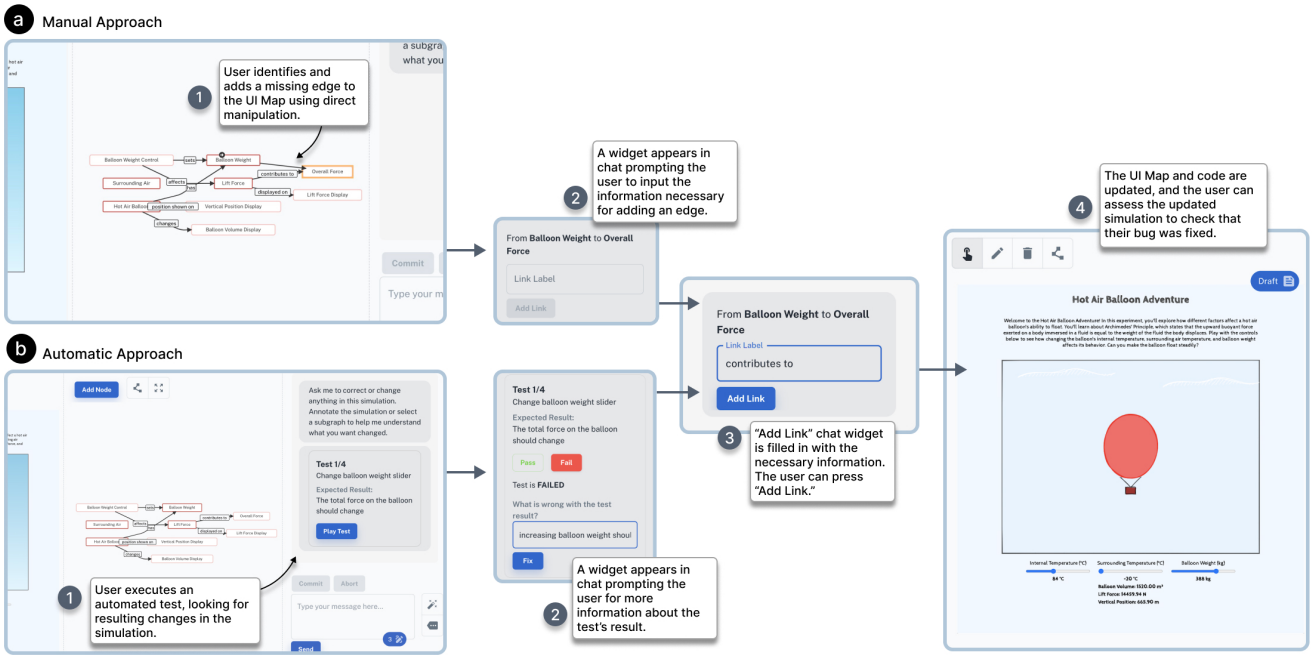

Figure 3: The debugging process can be split into two main approaches: the manual approach and the automatic approach, which provide the user varying levels of control and automation.

confirm, edit, or reject these suggestions, allowing him to refine the simulation without needing to modify code directly. Or, Mr. Carlos can manually select an abstraction and modify it, as seen in Figure 3a. All changes are shown as drafts until he chooses to commit or discard them, enabling iterative refinement grounded in his instructional intent.

4 System Architecture

Here we describe the technical details for (1) the CoA pipeline, (2) the undespecification resolution approach, (3) automated code testing, and (4) affordances for referential conversational interactions with the LLM.

4.1 Chain-of-Abstractions Pipeline

Rather than assuming educators have the necessary design experience to generate effective simulations, SimStep employs a chain-of-abstractions (CoA) technique that allows teachers' design decisions to be easily integrated into the previous steps of the simulation design process. Figure 4 shows the abstractions used in this chain. SimStep's CoA forward abstractions are node-link diagrams representing process and UI information. SimStep's set of forward transformation abstractions is

$$\mathcal{A} = \{\text{Concept Graph, Scenario Graph,}$$

$$\text{Learning Goal Graph, User Interaction Graph}\}$$

.

SimStep's set of transformations $\mathcal{T}$ therefore includes five transformation, one for the generation of each abstraction in $\mathcal{A}$ and one for generation of C. All abstractions in $\mathcal{A}$ can be refined through a set of six direct manipulation widgets. These forward abstraction refining widgets and their implementations are outlined in Table 1.

4.1.1 Concept Graph. The teacher's initial learning content prompt is translated into a knowledge graph, or node-link diagram, via LLM prompting. We call this knowledge graph abstraction the "Concept Graph,". In the Concept Graph, objects in the input text are become nodes, and the relationships between objects become directed links between nodes. This graph is visually displayed to the user upon generation. Knowledge graphs like this are commonly used for the representation of concepts in an educational setting, so teachers will be familiar with this form of abstraction [2, 14].

When prompting for this graphical abstraction, SimStep uses natural language request (including user inputted learning content) along with a list of requirements for the form of the generated graph.

4.1.2 Scenario Graph. Once an initial Concept Graph is generated, the teacher then selects a scenario. SimStep generates the Scenario Graph by promting an LLM to update the nodes in the Concept Graph to be specific to the chosen scenario. Links are remain the same. Narrowing a simulation down to a specific scenario or example does not change the conceptual relationships presented in the simulation, but may change the objects that the simulation is engaging with.

For example, a Concept Graph representing the states of matter may state that the nodes Solid and Liquid are connected via the link $\longrightarrow$ *melting* $\longrightarrow$. If the user selects the scenario "The Water Phase Transition", SimStep would generate a scenario graph where Ice is connected to Water via $\longrightarrow$ *melting* $\longrightarrow$.

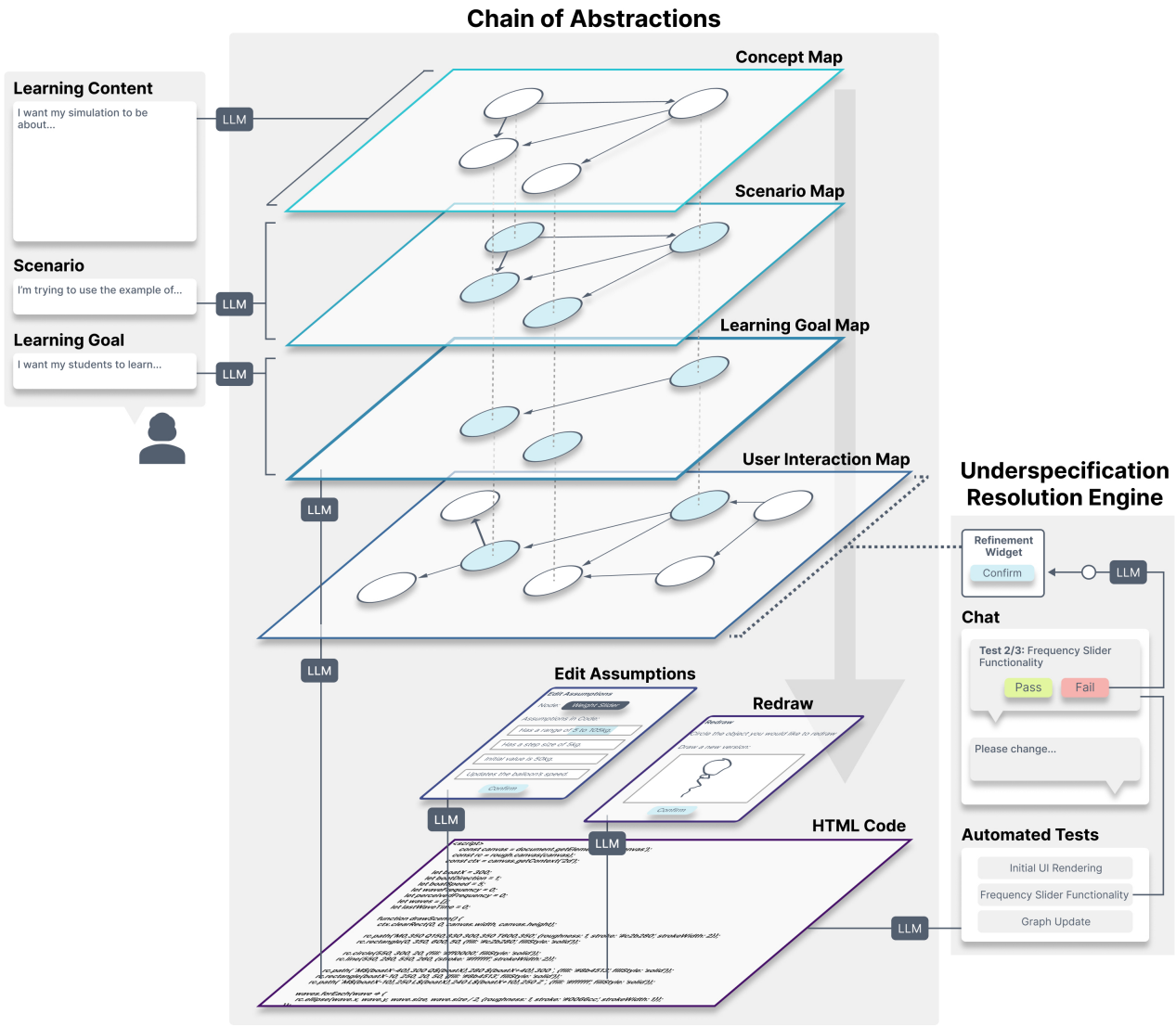

Figure 4: The system architecture of SimStep, consisting of a chain-of-abstractions pipeline for content generation and an underspecification resolution engine for inverse error correction. Abstractions in the chain-of-abstractions pipeline include node-link diagrams and chat-based widgets.

4.1.3 Learning Goal Graph. The Scenario Graph can be complex, with many nodes and links that are not relevant to the student's hypotheses-verification process. In the end simulation, students should have the ability to prove or disprove hypotheses to achieve scenario-specific learning goals without grappling with extraneous information. To address this, SimStep lets the user select a scenario-specific learning objective, then translates the Scenario Graph to a Learning Goal Graph by prompting an LLM to remove any nodes and links that are unnecessary to the chosen learning goal.

4.1.4 User Interaction Graph. The User Interaction Graph (UI Graph) represents the full simulation, including conceptual information about the content, UI elements, and visuals. It is generated once the teacher has made all three necessary simulation design decisions (Learning Content, Scenario, and Goals).

The UI Graph generation process involves identifying key objects and relationships in the learning goals and generating an experimental procedure based on this information.

Simulations addressing **descriptive knowledge** generate their UI Graph by first identifying (1) the independent variable of the learning goal, (2) the dependent variable, and (3) the relationship between these two variables. The third characteristic is identified as the learning goal itself, but SimStep's process also prompts for the

independent and dependent variables. Then all these are characteristics are used to generate an experimental procedure through LLM prompting. Finally, SimStep prompts to translate this procedure into an UI Graph with all necessary interactions to complete this procedure. For instance, if the learning goal is to understand how sunlight affects the height of plants, the procedure will involve the independent variable — amount of sunlight, the dependent variable — the height of the plant, and the relationship: more sunlight leads to taller plants. The specific procedure might be as follows:

```
(1) Select the plant for observation.
(2) Expose the plant to full sunlight
    for a week and measure its
    height daily.
(3) Change the condition to partial
    sunlight for the next week and
    continue measuring its height
    daily.
(4) Reduce sunlight to no sunlight
    for the final week and again
    measure its height daily.
(5) Record and compare the data to
    determine how the different
    sunlight conditions affected the
    plant's growth.
```

For **explanatory knowledge**, we find four main characteristics that must be identified. First, SimStep prompts for the main experimental object that the learning goal is exploring. Then SimStep again identifies the independent and dependent variables of the learning goal, along the explanatory object, or the object that explains the relationship between the independent and dependent variables. All four of these characteristics are then used to prompt for an experimental procedure, which is then translated to the UI Graph.

Learning goals that address **procedural knowledge** use two main characteristics. Namely, SimStep prompts for both the main experimental object that the learning goal is investigating, along with the underlying process that is explained through the learning goal. Using both of these features, SimStep then generates an experimental procedure to uncover the process presented in the learning goal. And again, this is translated into a UI Graph including all necessary experimental objects and processes.

4.1.5 Simulation Code. The UI Graph is then directly translated into simulation code. This simulation code includes HTML, CSS, and JavaScript for all functionality and visual elements included in the UI Graph.

4.2 Underspecification Resolution Approach

In SimStep, inverse correction is implemented using the Underspecification Resolution Engine. This process allows the user to correct any assumptions made by the LLM revealed at the final code level by refining abstractions in $\mathcal{B}$:

$$\mathcal{B} = \{\text{UI Graph, Code Assumptions Abstraction, Redraw Abstraction}\}$$

SimStep Forward Abstraction Refinement Widgets	
Widget	Description
Add Node	Requires the user to input the name of their new node and adds that node to the current abstraction.
Add Link	Requires the user to draw a new link between two nodes and input the label of their new link. Adds a new link between the provided nodes with the provided label to the current abstraction.
Remove Node	Removes the selected node from the current abstraction.
Remove Link	Removes the selected link from the current abstraction.
Edit Node Label	Changes the label of the selected node to a new, user inputted label.
Edit Link Label	This widget changes the label of the selected link to a new, user-inputted label.

Table 1: The widgets used to refine forward abstractions in SimStep.

The **UI Graph** maintains the same forward operation implementations as in the forward pass. In the **Code Assumptions Abstraction**, the user uses a chat widget to select a node in the UI Graph and inspect a list of assumptions that the code is making about that node. The user can then refine these assumptions through text editing. Once they've made edits, SimStep prompts an LLM to correct the code's implementation to reflect these updated assumptions. And the **Redraw Abstraction** requires the user to circle an object in the end simulation and create a rough sketch of what they want that object to look like visually using a chat widget. The system then prompts an LLM to update the circled visual to more closely match the user's rough sketch.

Using these abstractions, SimStep implements two approaches to underspecification resolution.

4.2.1 Guided Testing and Automated Repair. This approach combines automated code testing and abstraction modification suggestions. As described in Section 4.3.3, SimStep automatically tests and resolves code issues. However, tests involving user interface components are displayed to the user as chat widgets instead of automatically being resolved. Using these widgets, the user can automatically "play" UI actions and assess whether they produce expected results. When the user indicates that an unexpected result has occurred, SimStep invokes inverse correction. SimStep uses an LLM to select the previous abstraction in $\mathcal{B}$ that is most closely aligned with the assumption of interest, then automatically inspects and refines that abstraction. All the user must do is validate the updated abstraction B_i'. This updated abstraction is shown to the user using a chat widget..

In this approach, a pre-filled assumption modification widget is returned from the LLM as a JSON object with all necessary information. See Figure 5 for an example of this response for the "Edit Assumptions" abstraction.

LLM Response


```
{
  "type": 3,
  "message": "I suggest modifying the
Object node to ensure the hot air
balloon maintains a constant size
regardless of its vertical position.
This change will address the issue of
the balloon's size varying with
altitude.",
  "node": "Object",
  "assumptions": [
    "The hot air balloon maintains a
constant size regardless of vertical
position",
    "The balloon's visual representation
does not scale with altitude",
    "Changes in altitude only affect the
balloon's position, not its dimensions"
  ]
}
```


Visual Chat Display

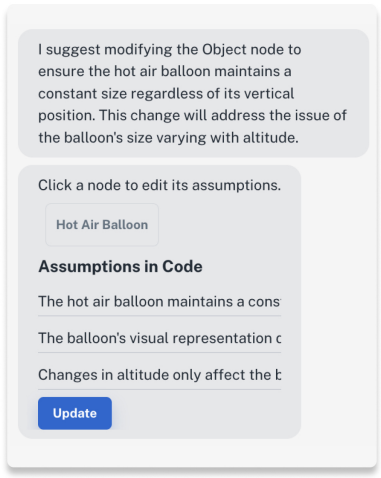

Figure 5: The JSON LLM response representing the "Edit Assumptions" abstraction, along with its visual display in the chat. "type" in the JSON object indicates the type of chat widget returned. Note that the "node" attribute corresponds to the ID of the relevant node, while the value displayed in the widget displays the label on that node.

4.2.2 Manual Debugging and Repair. The user can also manually select an abstraction from $\mathcal{B}$ to inspect, refine, and validate. For example, if the user notices that the concept graph is missing a node, they can directly select and fill out the "Add Node" widget to add it in. Or they can prompt the chat explaining the error, in which case an LLM will refine the affected abstraction for the user. The user can verify and accept this refinement.

4.3 Automated Code Testing

We noticed that some of the generated simulations do not work due to JavaScript, logical or User Interface issues. In order to fix this, we use a headless browser approach, combined with detailed logging. The detailed logging is achieved by capturing the UI and logic state of the simulation in the form of log messages, as well as taking screenshots after UI actions. We automate this using Puppeteer, a Node library used to control headless Chrome. Figure 6 shows the sequence diagram for the automated testing workflow, which happens seamless to the user.

4.3.1 JavaScript error resolution. We first ensure that the simulation has no JavaScript errors. This is crucial since JavaScript errors can prevent the UI from working properly. Since button clicks could also result in JavaScript errors, we go through all buttons and perform a click action for every button. All JavaScript errors are then captured using Puppeteer and sent to the LLM to fix.

4.3.2 Test case generation. Once the JavaScript errors are resolved, we ask the LLM to generate test cases. Each test case is defined as a JSON object with the following structure:

```
// Identifier of the UI element
"ID": "slider-weight",
// Action to perform
"action": "set_value",
// Value to set
"value": 80,
// Description of what's being tested
"description": "Adjust weight to observe balloon
    response",
// Expected outcome
"expectedOutcome": "Balloon altitude decreases",
// Whether this is UI-specific
"isUIVerification": true
```

4.3.3 Automated test case execution and verification. For tests that do not relate to UI components, SimStep automatically executes, verifies, and updates the code based on the test. In order to provide a comprehensive context to the LLM, we enable debug logging (e.g., capturing SVG coordinates and action timestamps) before executing the test cases with Puppeteer. We run each test case by executing the specified action and capturing a screenshot at the end of each test step. We also capture an initial and final screenshots of the simulation.

The LLM is then invoked with the test case execution results from Puppeteer, the simulation code and the learning goal. We ask the LLM to verify each test case and update the simulation code if the tests fail or if the learning goal is not met. If the LLM finds that a test case is not successful, or a learning goal is not satisfied, it will attempt to update the HTML with an updated version that addresses the underlying problem. We found that it does a very reliable job of ensuring logical errors related to the simulation are fixed. The detected UI errors are fixed based on the limited vision capabilities of the LLM.

Tests involving UI components are displayed in the chat for the user to verify.

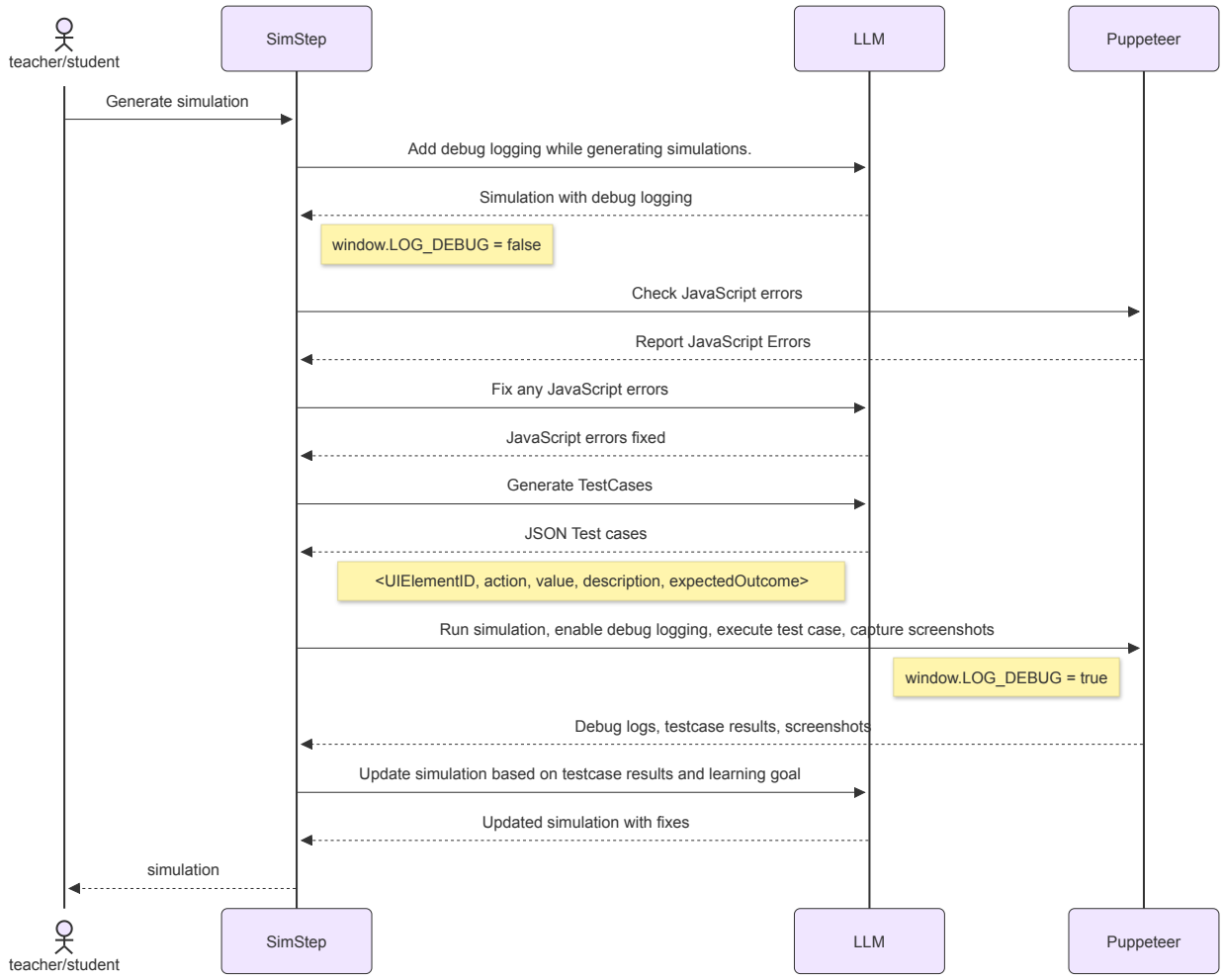

Figure 6: Automated Testing and Error Resolution Workflow for Simulation Verification

4.4 Improving Collaborative Interactions Between LLM and User

A side effect of the CoA is that the user must keep maintain understanding of multiple, often complex, abstractions of their authored simulation. This can make communicating with the LLM during the Underspecification Resolution Process cognatively expensive. As a means of improving the user experience of SimStep during this process, we have also implemented features that allow users to more clearly communicate their ideas to the LLM, along with allowing users to understand the assumptions of the LLM.

4.4.1 Chat Add-Ons. When prompting via the chat, the user often would like to reference simulation abstractions to explain their desired changes. In order the aid the user in doing this, the user can draw annotations on top on the current simulation. Every time the user annotates, the annotation is labeled. In chat, the user can type "@" to get a list of all nodes in the UI Graph, along with all annotations on the simulation. The user can select from this list to reference a specific annotation or node. When prompting for the desired change to the code, the LLM is provided with these annotations and nodes as context.

4.4.2 Connecting Code and Abstraction. The translation from UI Graph to HTML Code can be cognitively intense for users. In order to help users understand the connection between the two abstractions, we've implemented a "Subgraph Selector" tool, in which the user can circle one or multiple sections of the end simulation and the tool will display the sub-graph of the UI Graph that this region corresponds to. This is implemented by prompting the LLM with an image of the annotated simulation along with the context of the simulation code and UI Graph.

4.5 Implementation Details

4.5.1 Frontend. The frontend of the SimStep application is built using `React` [40]. For many of the UI elements and visual components, we use Material UI [41], a UI component library. In order to

visually and directly manipulate the abstractions that have a graphical structure, we use `joint.js` [19]. While for parsing these graphs, we use a `mermaid.js` [55] format. We also employ `tldraw` [45] for annotating in the Interactivity Page.

4.5.2 Backend. The backend is built using Node.js [21] and Express.js [26]. We use Anthropic's Claude [5] claude-3.5-sonnet model for all large language model prompting, which we do throughout the process of generating and modifying abstractions in our system. In the design of prompts in this system, we employed several prompting techniques in a trial-and-error approach, resulting in outputs that have consistent form and quality. We also store user-generated simulation code in a Firebase [22] database. This allows users to access the simulations they've created by link.

4.5.3 Prompt Engineering. By aligning the interface with the strengths and limitations of the LLM, we created a system that simplifies the authoring process while maintaining the necessary balance between AI capabilities and user control. As Adar puts is *"Your UI shouldn't write checks your AI can't cash, and your AI shouldn't write checks your UI can't cash*[1]*."*.

5 User Evaluation

To understand the user experience of SimStep's Chain-of-Abstractions (CoA) approach, we conducted a user study with $N = 11$ educators. Participants had an average of 9.18 years of teaching experience ($\mu = 9.18$, $\sigma = 6.98$) and were recruited through social media and the researchers' professional networks. All participants had prior experience teaching STEM subjects, spanning grade levels from middle school to undergraduate college. Each study session lasted approximately 90 minutes and was conducted one-on-one over Zoom, during which participants interacted with a deployed version of SimStep. Participants received a $40 honorarium for their time. The study protocol was approved by the institution's Institutional Review Board (IRB).

5.1 Method

In each session, one of the researchers gave a participant a 15-minute demo of SimStep, going through the process of generating a simulation about *States of Matter and Phase Change*. The participant could ask any question they had about the tool. Then, the researcher allowed the participant to use SimStep to generate their own simulation about either a subject matter of the participant's choice or one of two pre-prepared learning content (Gravitational Force and Buoyancy). When time permitted, a second simulation was generated using a second learning content. After the participant generated and corrected at least one simulation, they were asked to reflect on their experience using SimStep through an unstructured interview. Finally, participants filled out a questionnaire on the usability of the system, the task load, and the Cognitive Dimensions of Notations [23]. The usability of the system was evaluated using the Post-Study System Usability Questionnaire (PSSUQ) [36], and the task load was assessed using the NASA Task Load Index (NASA-TLX) [24].

5.2 Results

[1]https://www.youtube.com/watch?v=11UKXaELg8M

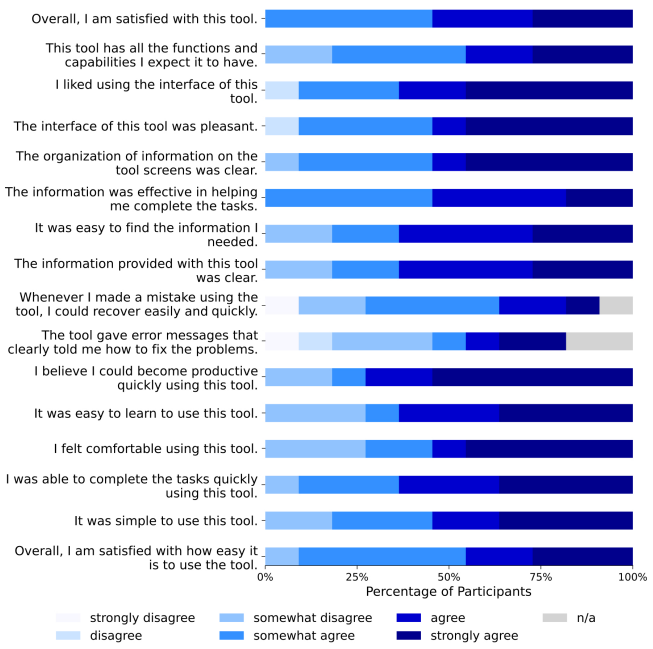

Figure 7: Teacher Participant Responses to Post-Study System Usability Questionnaire (PSSUQ)

5.2.1 System Usability. Overall, teachers found SimStep intuitive to use and reported that they would feel comfortable using this system in a classroom setting. In the PSSUQ questionnaire, we anchored 1 as "Strongly Disagree" and 6 as "Strongly Agree." Our overall usability score was 4.66 ($\sigma = 0.36$), and in qualitative feedback multiple participants commented that the system felt "natural to use." The System Usefulness (SYSUSE) sub-scale was 4.8 ($\sigma = 0.15$), indicating that the teachers viewed this tool as useful to their teaching process. Teachers also remarked that they could see themselves using this tool in a variety of ways, including to create introductory material for students, exploratory expansions of their class subjects, and replacement materials for in-class activities such as experiments. Further, the Interface Quality (INTERQUAL) sub-scale had a score of 4.78 ($\sigma = 0.15$), indicating the teachers felt that they had the ability to navigate and alter the CoA and the underspecification resolution process with ease. However the Information Quality (INFOQUAL) sub-scale had a score of 4.44 ($\sigma = 0.48$), indicating that we can improve the quality of information displayed to the user. As seen in Figure 7, several users found that error messages were not present or useful during their process. There is room for integration of more intuitive error messages when the user encounters an unexpected error in the system.

5.2.2 Task Load. Although SimStep does not eliminate the task load of simulation generation, our system provides an interface that imparts a load onto users that is not high. We evaluated task load using NASA-TLX with a 1-6 scale. We anchored 1 with "Very Low" and 6 with "Very High." We did not directly assess the physical

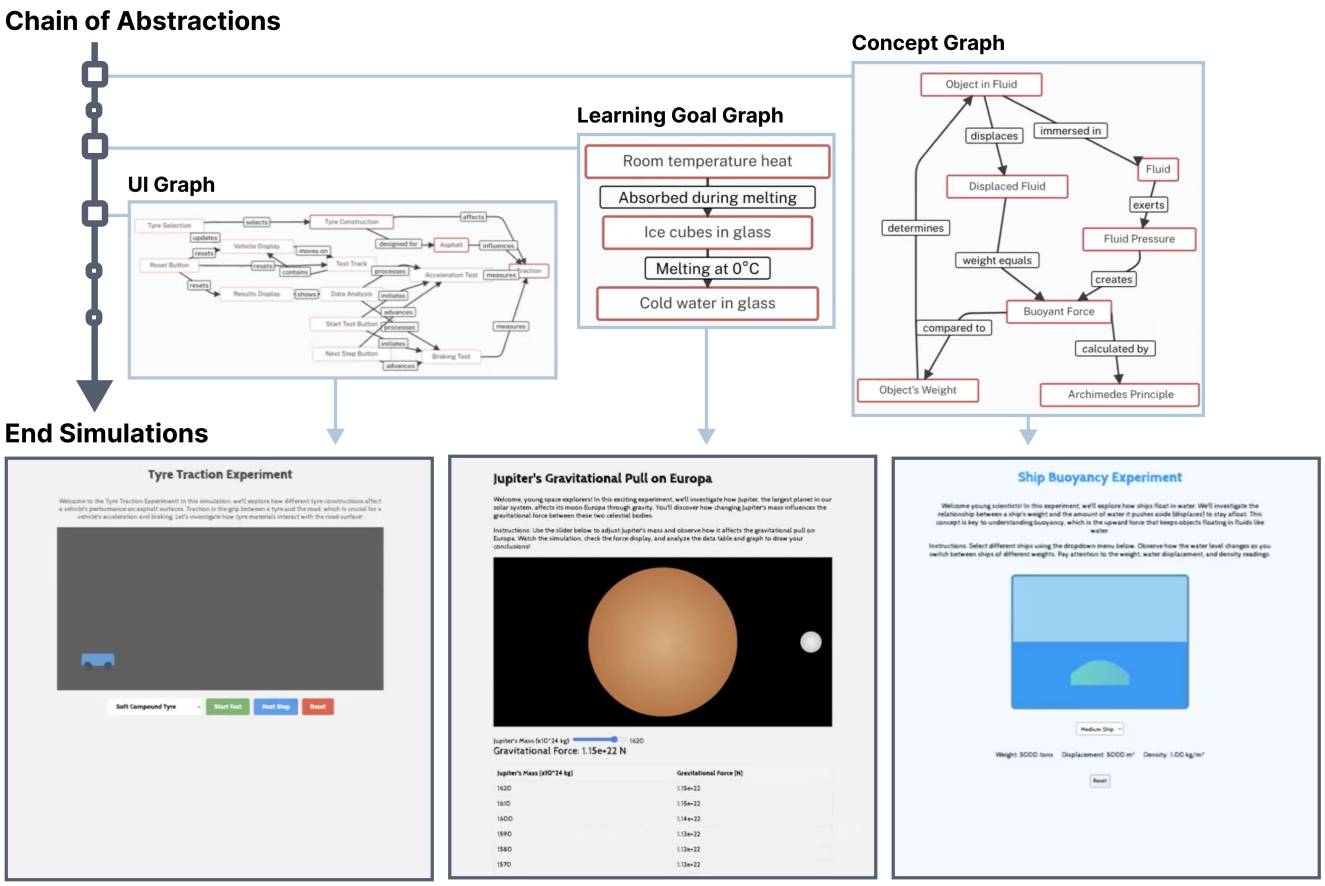

Figure 8: A selection of simulations and intermediate representations generated by teachers during our expert evaluation study. This set of simulations includes an exploration of friction on different tire materials, mass versus gravitational force, and buoyancy on different sized ships.

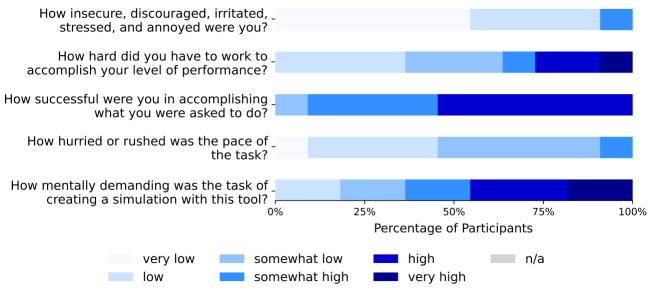

Figure 9: Teacher Participant Responses to NASA's Task Load Index (NASA-TLX) Questions

demand of our tool considering digital nature of our tool. The unweighted TLX score for SimStep was 2.64, indicating that although teachers felt that the task of generating a simulation required work, this load was not high. In Figure 9, we can see that teachers felt that there was considerable mental demand in the simulation generation process. But regardless on this mental demand, most participants felt that they were ultimately successful in the process. See Figure 8 for some of the simulations that teachers generated during this study. In qualitative feedback, participants remarked that there was low mental demand in the initial steps of the simulation generation process. Mental demand increased at the step of correcting behavior and errors at the simulation code abstraction level (i.e., the Interactivity Page). Participants most often engaged in the underspecification resolution process by prompting the chat. In association with the task load of this activity, one participant remarked "I wanted to convey my thoughts specifically enough that they would be understood, so I struggled at first with how to describe things." In response to participants' feelings of being lost in the under-specification resolution process, the researchers augmented the process with automated code testing and guided testing functionality, as discussed in section 4.3.3.

5.2.3 Cognitive Dimensions of Notations. Teachers also found that the the notations used in the SimStep's CoA were intuitive and mirrored their own internal process when planning to teach about

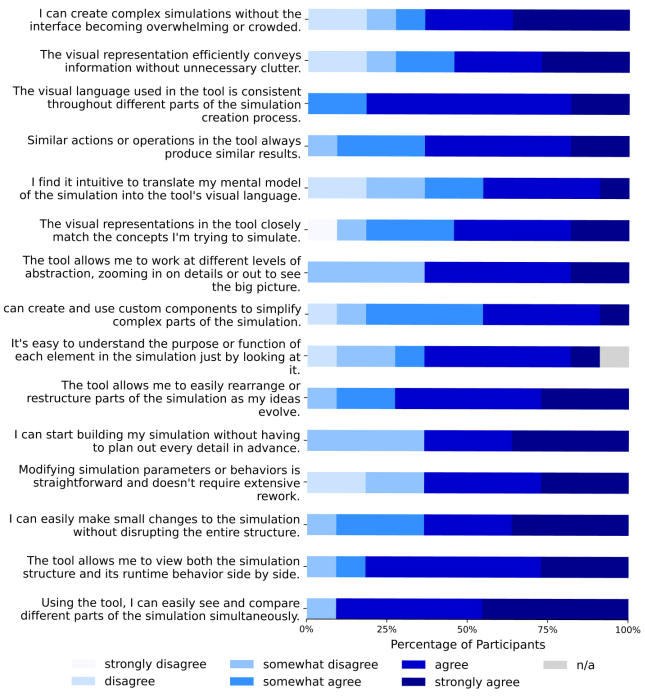

Figure 10: Teacher Participant Responses to a Questionnaire on the Cognitive Dimensions of Notations

a subject. More specifically, we assessed these notations (abstractions) using the Cognitive Dimensions of Notations [23]. These dimensions allow us to evaluate whether or not these abstractions are useful to teachers for the task of simulation generation, and are often used to evaluate representational tools [9, 34]. We found that SimStep meets all the most relevant dimensions. In this study, we evaluated users' experiences of the system specifically for the dimensions of Visibility, Viscosity, Premature Commitment, Role-Expressiveness, Abstraction, Closeness of Mapping, Consistency, and Diffuseness using Likert scale questions with a range of 1-6. We anchored 1 with "Strongly Disagree" and 6 with "Strongly Agree." Figure 10 shows teachers' response to these questions. The average score across all included questions was 4.61 (μ = 4.61, σ = 0.34). Questions related with visibility had a notable average score of 5.14 (μ = 5.14, σ = 0.27), which indicated success in our goal of generating a tool that easily displays different abstractions of the same end simulation. In qualitative feedback, teachers did however indicate that they focused less on the UI Graph on the Interactivity Page, which is expected considering this page's focus on the behavior of the simulation itself.

5.2.4 Qualitative Feedback. In a discussion period after using the tool, participants remarked that they enjoyed the graphical abstractions of the inputted learning content. They said they found this form of abstraction to be intuitive and useful for even their own knowledge formation process. One participant remarked that *"The graphical abstractions help to show the underlying concepts that are running the simulation. It is also nice to be able to adjust those which would adjust the simulation's behavior."* Some teachers even commented that they would consider giving these maps, or the tool in general, to students to help them learning the concepts at hand. Likewise, teachers found value in the Scenario Page specifically. They appreciated that they could choose a topic specific to their students. One participant also commented that they often struggle to express learning content through different examples when students do not understand their original example. This participant said they appreciated that the tool provides example scenarios to choose, which may present the content in ways that they may not have thought of.

Despite these successes, participants did have a few suggestions regarding the visual display of the graph abstractions. The current interface does not have a zoom feature, and the generated maps can become quite complex. One participant commented *"Just for me personally, I have eye disabilities, I was not able to easily zoom in on the end screen with the simulation."* In order to address this, we plan on adding a zoom and pan feature to this display.

6 Technical Evaluation

The effectiveness of a CoA approach in promoting distributed cognition is dependent on the **fidelity** of the included abstractions. We evaluate the fidelity of forward transformation abstractions in SimStep with 13 curated simulation specifications. The lead researcher collects a range of STEM learning content, culturally relevant scenarios, and learning goals using online educational resources. A learning design student then scores the generated abstractions from 1 to 10 using the question "How closely does this abstraction adhere to the previous abstraction and user inputs?", where 1 is "very far" and 10 is "very close." This student is recruited through professional connections.

6.1 Results

In Table 2, we compare the fidelity scores of all forward transformation abstractions in SimStep. In general, abstractions are shown to have high fidelity, with an overall mean fidelity score of 7.6 (μ = 7.6, σ = 2.01). However, SimStep has room to improve on both the Scenario Graph and the Learning Goal Graph abstractions, which have mean fidelity scores of 6.65 and 7.08, respectively. Future work involves performing a more in-depth study to understand the pitfalls of these abstractions and learn how to improve their fidelity by working with a range of domain experts. In Figure 11, we also compare a selection of the simulations generated in this evaluation to simulations generated using a direct prompt-to-code approach (where the prompt includes the learning content, scenario, and learning goal) with various LLM models.

7 Discussion

7.1 Desiderata for Abstractions in CoA

In software engineering, abstractions are essential for managing complexity, modularizing functionality, and enabling reasoning at multiple levels of the system [48]. Our CoA framework applies this perspective to programming-by-prompting by introducing intermediate representations that formalize partial intent and guide the step-by-step transformation from natural language to code. Ideally,

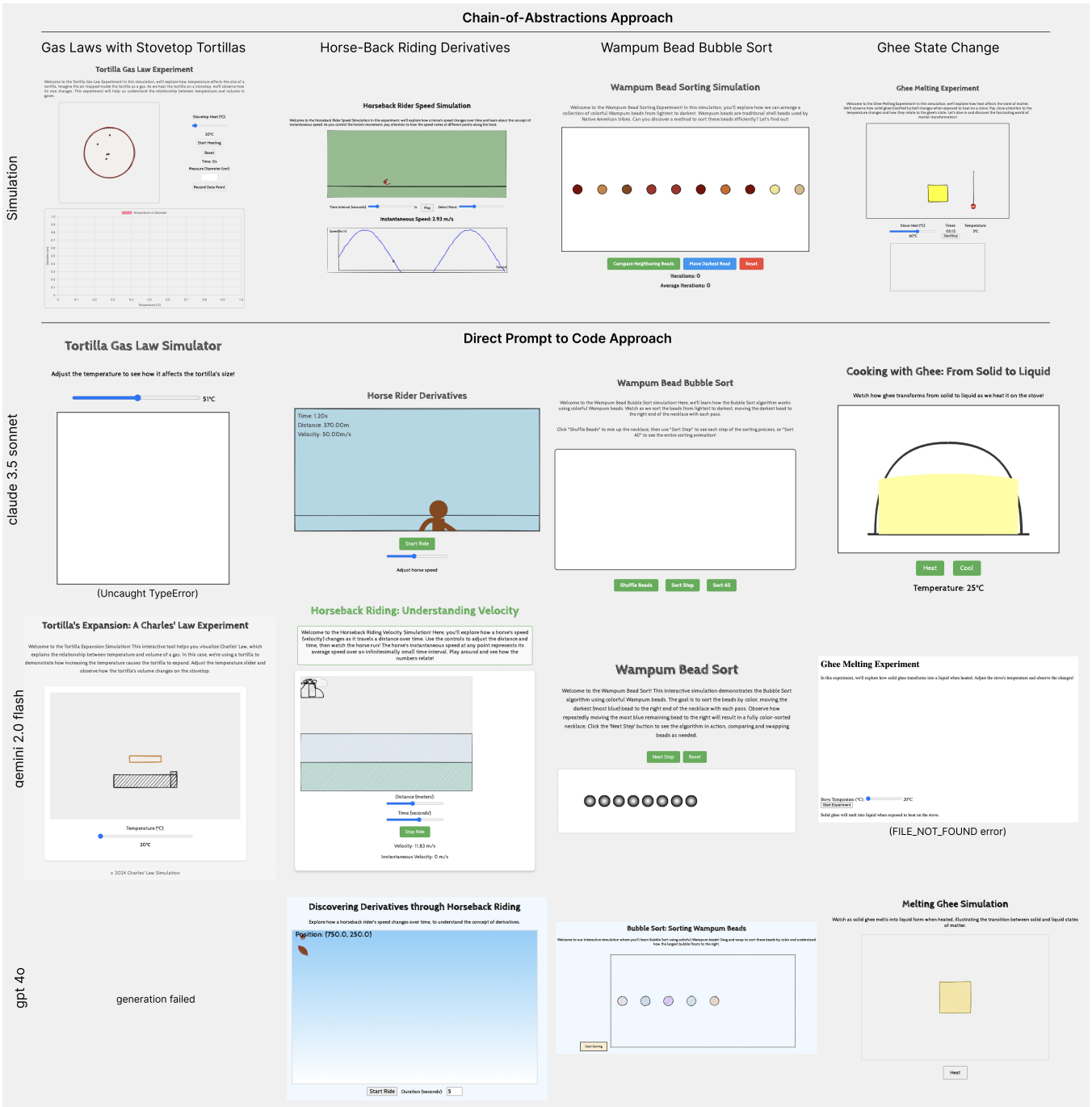

Figure 11: A selection of simulations generated through the CoA approach in comparison to simulations generated using a direct prompt-to-code approach (where the prompt includes the learning content, scenario, and learning goal)

the abstractions selected in CoA should **reflect the representational structures of the domain and the decomposition of the task**. In SimStep, for example, the Concept Graph captures the core domain specific knowledge and relationships, the Scenario Graph expresses contextualized learning situations, the Learning Goal Graph defines desired educational outcomes, and the UI Interaction Graph specifies how learners will interact with the system. Each of these representations corresponds to a meaningful task boundary in the teacher's workflow and provides a different lens for inspecting and refining intent.

SimStep Forward Abstraction Fidelity		
Abstraction	**Fidelity Scores**(1-10)	
	μ	σ
Concept Graph	8.50	1.13
Scenario Graph	6.65	2.12
Learning Goal Graph	7.08	2.46
UI Graph	8.23	1.38

Table 2: All forward abstractions are scored for fidelity over 13 curated simulation specifications, and mean (μ) and standard deviation (σ) fidelity scores are reported. The scorer notes that they could have potentially been biased towards assigning harsher scores to abstractions on topics that they are more proficient in.

Across these contexts, designing effective abstractions requires attention to several key properties. In educational simulation authoring, abstractions should exhibit **visibility** by making key relationships and behaviors perceptible. For instance, a Concept Graph should clearly show that *temperature affects air density*, helping a teacher reason about the cause-and-effect dynamics of buoyancy. Abstractions should also maintain **interpretive continuity** across levels; for example, a concept like "heat" introduced in the Concept Graph should persist through the Scenario Graph and be reflected in UI elements such as a "heating switch" or "temperature slider." They must be **actionable**, allowing users to modify content directly such as enabling a teacher to revise the range of a slider or change the linked behavior of a button without touching the generated code. Abstractions should also support **propagation**, so that edits to a node in the Learning Goal Graph (e.g., changing "students should understand buoyant force" to "students should compare hot vs. cold air") trigger meaningful updates in downstream abstractions and ultimately in the simulation logic.

Further, the sequence of abstractions should follow a logic of **progressive formalization**. Early abstractions like Concept and Scenario Graphs align with the teacher's domain expertise and planning practices, allowing them to express ideas in familiar pedagogical terms. Later abstractions like the Interaction Graph introduce more structure such as conditional logic or state transitions—providing a bridge between educational intent and executable behavior. At each level, users should be able to validate whether the representation reflects their goals (e.g., "does this scenario match my intended classroom example?"), detect where the system has introduced incorrect or missing assumptions, and revise the abstraction accordingly before continuing to code generation.

7.2 Broader Utility

While SimStep exemplifies the CoA framework by helping educators author interactive simulations, the framework (Section 2) itself has the potential to support a broad range of scenarios. The central construct of CoA is its structuring of the code generation process into a series of task-aligned abstractions. This approach connects to longstanding goals in end-user programming [32] which aims to empower non-programmers to create, adapt, and control computational artifacts. Compared to current approaches which fall along the spectrum of tradeoffs between expressive power and ease of use, the CoA framework offers a middle path as evidenced by our user study: by representing programs as structured editable abstractions that reflect end-user's task logic, it preserves expressive power while maintaining accessibility and semantic clarity. Furthermore, our framework generalizes across domains. For instance, a data scientist might move from analysis goals to transformation logic to visual outputs [64]; a game designer might articulate core mechanics, progression rules, and interaction feedback [1]. In each case, abstractions are chosen to reflect natural task decompositions and representational practices in the domain, ensuring that the pipeline aligns with how users already think.

7.3 Limitations and Future Work

While the CoA framework and our implementation of SimStep offers a principled structure for programming-by-prompting, several limitations point towards future extensions to the framework. The current implementation, though effective for educational simulation authoring, may limit flexibility in domains where task workflows are less standardized or abstractions are harder to formalize. Designing methods for customizing or synthesizing domain-appropriate abstractions remains an open challenge. Second, while CoA reduces the need for syntactic programming, it introduces new forms of cognitive load: users must interpret and manipulate layered representations. As evidenced in the user study, the quality of outputs depends not only on the model's capabilities, but also on the user's ability to structure intent within the abstraction pipeline. This calls for adaptive interfaces that scaffold abstraction complexity based on user expertise. Third, current LLMs are not inherently abstraction-aware and often fail to maintain consistency across stages, suggesting opportunities for model refinement and fine-tuning. Finally, although our inverse process supports underspecification resolution, errors stemming from hallucination, misalignment, or representational gaps remain difficult to detect and correct. Ultimately, these directions point toward a broader call for understanding human-aligned representations that support not just code generation, but meaningful engagement, control, and adaptation across diverse user groups and domains.

8 Related Work

8.1 Prompt Based Programming Systems

Programming-by-prompting enables users to generate code using natural language, providing promising opportunities for end-user programming [29]. These systems change the role of the programmer from authoring code to verifying and debugging it [47], and have shown promise in helping users solve a range of programming tasks [16]. However, users frequently struggle with prompt ambiguity, underspecified behavior, and limited control over generated outputs [16]. To address these challenges, recent tools embed prompting within structured workflows that offer modular prompting, multi-step interfaces, or visual scaffolds [10, 15, 30]. For example, Devy [11] infers user intent from natural language to apply codebase modifications, while ProgramAlly [25] and Spellburst [4] blend LLMs with domain-specific interactive UIs. These systems highlight a growing consensus that effective programming-by-prompting workflows require external representations and scaffolds to support user steerability.

Our work builds on this research trajectory by introducing the CoA framework that decomposes prompt to code generation into a sequence of structured, interpretable representations. Unlike prior systems that operate over one or two abstraction levels that are primarily program-driven, CoA defines a multi-stage pipeline explicitly designed to mirror users' task workflows and enable semantic control at each step.

8.2 Error Correction via Human-AI Interaction

Even with well-formed prompts, LLM-generated code is susceptible to errors stemming from hallucinations, incomplete specifications, or semantic mismatches [7, 50, 65]. Research has sought to classify these errors and develop strategies for correction, including test-based evaluation [27], execution-trace validation [35], and iterative refinement through self-critique [18]. Systems like Rectifier [66] automate code validation using curated test suites, while Fan et al. [20] demonstrate that program repair techniques can fix common LLM errors. Automated testing is useful, but does not reveal all hidden bugs. These testing techniques rely heavily on runtime feedback, which is not always helpful for LLM debugging [58].

Further, these methods often treat the user as a passive recipient of model output. In contrast, hybrid systems incorporate human-in-the-loop workflows by externalizing the model's assumptions, allowing users to inspect, correct, and guide synthesis [37, 63]. Other systems such as Whyline [33] and Hypothesizer [3] reimagine debugging as a process of explanation and hypothesis-testing, rather than syntactic error fixing. These tools allow users to query program behavior through natural language or runtime traces, reinforcing the idea that debugging can be a meaning-making activity. In this work, we combine automated testing techniques with human-in-the-loop prompting techniques, such as directly revealing assumptions [63]. Further, we extend this approach through a structured *inverse correction process*, which explicitly surfaces underspecifications and assumptions in code at each abstraction level in the CoA pipeline.

8.3 Abstractions for Programming

End-user programming systems have long used representational abstractions to help non-programmers express complex behavior [42]. Programming abstractions come in all shapes and sizes, but abstractions including concept graphs aimed at representing high-level relationships [61], block-based programming structures [67], and scene graphs representing semantic information [6, 43] tend to be hierarchical and domain-specific. Early tools like KidSim [49] and more recent platforms such as Spellburst [4] employ block-based or visual metaphors to enable rule creation without traditional syntax.

Beyond syntactic simplification, abstractions play a deeper cognitive role: they help users approach complex tasks by engaging with structured external representations. Scene graphs [6, 43], concept maps [61], and scenario models [12] are examples of domain-specific abstractions that encode relationships, behaviors, and user goals in an interpretable form. These representations are not just visualization aids—they serve as manipulable scaffolds for cognitive work, compliant with the theory of distributed cognition [28], which emphasizes how reasoning is offloaded onto and supported by external artifacts. Recent research also expands on how users engage with these abstractions in the context of authoring and debugging interactive systems. Tools like CodeToon [53], RealitySketch [54], and Kitty [31] demonstrate how structured workflows combined with sketching, annotation, or direct manipulation can support creative tasks.

Building on these threads, our work introduces the Chain-of-Abstractions (CoA) framework as a generalization and formalization of abstraction-based authoring. Further, CoA emphasizes domain alignment and task decomposition, drawing from interactive content authoring systems that integrate goals, scenarios, and interaction patterns into the programming workflow [13, 44, 46, 52, 59]. As such, SimStep leverages CoA not only to scaffold code generation but also to support ambiguity resolution, increase control, and make the authoring process accessible to non-programmers across diverse domains.

9 Conclusion

This work introduces the Chain-of-Abstractions (CoA) framework as a principled approach to programming-by-prompting, one that treats code generation not as a single-shot translation, but as a structured process of task-level semantic articulation. By decomposing the synthesis process into cognitively meaningful, domain-aligned representations, CoA enables users to externalize, inspect, and iteratively refine their intent. We instantiate this approach in SimStep, a tool that supports educators in authoring interactive simulations through a scaffolded, human-in-the-loop workflow. SimStep's inverse correction process addresses underspecification by surfacing assumptions and guiding revision at abstraction checkpoints, recovering key affordances of traditional programming such as traceability, testability, and control. CoA provides a foundation for more controllable, expressive, and domain-sensitive code generation.

References

[1] Ernest Adams. 2014. *Fundamentals of game design.* Pearson Education.

[2] Garima Agrawal, Yuli Deng, Jongchan Park, Huan Liu, and Ying-Chih Chen. 2022. Building knowledge graphs from unstructured texts: Applications and impact analyses in cybersecurity education. *Information* 13, 11 (2022), 526.

[3] Abdulaziz Alaboudi and Thomas D Latoza. 2023. Hypothesizer: A Hypothesis-Based Debugger to Find and Test Debugging Hypotheses. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. 1–14.

[4] Tyler Angert, Miroslav Suzara, Jenny Han, Christopher Pondoc, and Hariharan Subramonyam. 2023. Spellburst: A node-based interface for exploratory creative coding with natural language prompts. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. 1–22.

[5] Anthropic. 2023. Claude: Language Model by Anthropic. https://www.anthropic.com/.

[6] Iro Armeni, Zhi-Yang He, JunYoung Gwak, Amir Zamir, Martin Fischer, Jitendra Malik, and Silvio Savarese. 2019. 3D Scene Graph: A Structure for Unified Semantics, 3D Space, and Camera. *2019 IEEE/CVF International Conference on Computer Vision (ICCV)* (2019), 5663–5672. https://api.semanticscholar.org/CorpusID:203837042

[7] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[8] Michael P Barnett and WM Ruhsam. 1968. A natural language programming system for text processing. *IEEE transactions on engineering writing and speech* 11, 2 (1968), 45–52.

[9] Michael Bostock and Jeffrey Heer. 2009. Protovis: A graphical toolkit for visualization. *IEEE transactions on visualization and computer graphics* 15, 6 (2009), 1121–1128.

[10] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134* (2024).

[11] Nick C Bradley, Thomas Fritz, and Reid Holmes. 2018. Context-aware conversational developer assistants. In *Proceedings of the 40th International Conference on Software Engineering*. 993–1003.

[12] Corey Brady, Brian Broll, Gordon Stein, Devin Jean, Shuchi Grover, Veronica Cateté, Tiffany Barnes, and Ákos Lédeczi. 2022. Block-based abstractions and expansive services to make advanced computing concepts accessible to novices. *Journal of Computer Languages* 73 (2022), 101156.

[13] Salman Cheema and Joseph LaViola. 2012. PhysicsBook: a sketch-based interface for animating physics diagrams. In *Proceedings of the 2012 ACM international conference on Intelligent User Interfaces*. 51–60.

[14] Penghe Chen, Yu Lu, Vincent W Zheng, Xiyang Chen, and Boda Yang. 2018. Knowedu: A system to construct knowledge graph for education. *Ieee Access* 6 (2018), 31553–31563.

[15] Bhavya Chopra, Yasharth Bajpai, Param Biyani, Gustavo Soares, Arjun Radhakrishna, Chris Parnin, and Sumit Gulwani. 2024. Exploring Interaction Patterns for Debugging: Enhancing Conversational Capabilities of AI-assistants. *arXiv preprint arXiv:2402.06229* (2024).

[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* (Toronto ON, Canada) *(SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1136–1142. https://doi.org/10.1145/3545945.3569823

[17] Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. 2023. Chain-of-verification reduces hallucination in large language models. *arXiv preprint arXiv:2309.11495* (2023).

[18] Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, et al. 2024. What's Wrong with Your Code Generated by Large Language Models? An Extensive Study. *arXiv preprint arXiv:2407.06153* (2024).

[19] David Durman. 2024. joint.js. https://www.jointjs.com/.

[20] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.

[21] OpenJS Foundation. 2024. Node.js. https://nodejs.org/en/.

[22] Google. 2024. Firebase. https://firebase.google.com/.

[23] Thomas RG Green. 1989. Cognitive dimensions of notations. *People and computers V* (1989), 443–460.

[24] SG Hart. 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. *Human mental workload/Elsevier* (1988).

[25] Jaylin Herskovitz, Andi Xu, Rahaf Alharbi, and Anhong Guo. 2024. ProgramAlly: Creating Custom Visual Access Programs via Multi-Modal End-User Programming. *ArXiv* abs/2408.10499 (2024). https://api.semanticscholar.org/CorpusID:271909496

[26] TJ Holowaychuk. 2024. Express.js. https://expressjs.com/.

[27] Zichao Hu, Francesca Lucchetti, Claire Schlesinger, Yash Saxena, Anders Freeman, Sadanand Modak, Arjun Guha, and Joydeep Biswas. 2024. Deploying and evaluating llms to program service mobile robots. *IEEE Robotics and Automation Letters* 9, 3 (2024), 2853–2860.

[28] Edwin Hutchins. 1995. *Cognition in the Wild*. MIT press.

[29] Ellen Jiang, Edwin Toh, A. Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J. Cai, and Michael Terry. 2022. Discovering the Syntax and Strategies of Natural Language Programming with Generative Language Models. *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (2022). http://dl.acm.org/citation.cfm?id=3501870

[30] Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. 2022. Discovering the syntax and strategies of natural language programming with generative language models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–19.

[31] Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, and George Fitzmaurice. 2014. Kitty: sketching dynamic and interactive illustrations. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*. 395–405.

[32] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan F. Blackwell, Margaret M. Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad A. Myers, M. Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)* 43 (2011), 1 – 44. https://api.semanticscholar.org/CorpusId:128364433

[33] Amy J Ko and Brad A Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 151–158.

[34] Benjamin Lee, Arvind Satyanarayan, Maxime Cordeil, Arnaud Prouzeau, Bernhard Jenny, and Tim Dwyer. 2023. Deimos: A grammar of dynamic embodied immersive visualisation morphs and transitions. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–18.

[35] Teemu Lehtinen, Charles Koutcheme, and Arto Hellas. 2024. Let's Ask AI About Their Programs: Exploring ChatGPT's Answers To Program Comprehension Questions. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*. 221–232.

[36] James R Lewis. 1992. Psychometric evaluation of the post-study system usability questionnaire: The PSSUQ. In *Proceedings of the human factors society annual meeting*, Vol. 36. Sage Publications Sage CA: Los Angeles, CA, 1259–1260.

[37] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin G. Zorn, J. 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. *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (2023). https://api.semanticscholar.org/CorpusID:258107840

[38] Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. 2023. Faithful chain-of-thought reasoning. In *The 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (IJCNLP-AACL 2023)*.

[39] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 1–15.

[40] Meta. 2024. React. https://react.dev/.

[41] MUI. 2024. Material UI. https://mui.com/material-ui/.

[42] Brad A Myers, Amy J Ko, and Margaret M Burnett. 2006. Invited research overview: end-user programming. In *CHI'06 extended abstracts on Human factors in computing systems*. 75–80.

[43] Nico Ritschel, Felipe Fronchetti, Reid Holmes, Ronald Garcia, and David C. Shepherd. 2022. Can guided decomposition help end-users write larger block-based programs? a mobile robot experiment. *Proceedings of the ACM on Programming Languages* 6 (2022), 233 – 258. https://api.semanticscholar.org/CorpusID:253239351

[44] Karl Toby Rosenberg, Rubaiat Habib Kazi, Li-Yi Wei, Haijun Xia, and Ken Perlin. 2024. DrawTalking: Building Interactive Worlds by Sketching and Speaking. *arXiv preprint arXiv:2401.05631* (2024).

[45] Steve Ruiz. 2021. tldraw: A tiny little drawing app. https://tldraw.com/.

[46] Nazmus Saquib, Rubaiat Habib Kazi, Li-yi Wei, Gloria Mark, and Deb Roy. 2021. Constructing embodied algebra by sketching. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–16.

[47] Advait Sarkar, Andrew D Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence? *arXiv preprint arXiv:2208.06213* (2022).

[48] Ahmed Seffah, Jan Gulliksen, and Michel C Desmarais. 2005. *Human-centered software engineering-integrating usability in the software development lifecycle*. Vol. 8. Springer Science & Business Media.

[49] David Canfield Smith, Allen Cypher, and Jim Spohrer. 1994. KidSim: Programming agents without a programming language. *Commun. ACM* 37, 7 (1994), 54–67.

[50] Da Song, Zijie Zhou, Zhijie Wang, Yuheng Huang, Shengmai Chen, Bonan Kou, Lei Ma, and Tianyi Zhang. 2023. An empirical study of code generation errors made by large language models. In *7th Annual Symposium on Machine Programming*.

[51] Hari Subramonyam, Roy Pea, Christopher Pondoc, Maneesh Agrawala, and Colleen Seifert. 2024. Bridging the Gulf of Envisioning: Cognitive Challenges in Prompt Based Interactions with LLMs. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–19.

[52] Hariharan Subramonyam, Colleen Seifert, Priti Shah, and Eytan Adar. 2020. Texsketch: Active diagramming through pen-and-ink annotations. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–13.

[53] Sangho Suh, Jian Zhao, and Edith Law. 2022. Codetoon: Story ideation, auto comic generation, and structure mapping for code-driven storytelling. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. 1–16.

[54] Ryo Suzuki, Rubaiat Habib Kazi, Li-Yi Wei, Stephen DiVerdi, Wilmot Li, and Daniel Leithinger. 2020. Realitysketch: Embedding responsive graphics and visualizations in AR through dynamic sketching. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 166–181.

[55] Knut Sveidqvist and Mermaid.js contributors. 2014. Mermaid: Generation of diagrams and flowcharts from text in a similar manner as markdown. https://mermaid.js.org/.

[56] Chee Wei Tan, Shangxin Guo, Man Fai Wong, and Ching Nam Hang. 2023. Copilot for Xcode: exploring AI-assisted programming by prompting cloud-based large language models. *arXiv preprint arXiv:2307.14349* (2023).

[57] Mei Tan and Hari Subramonyam. 2024. More than model documentation: uncovering teachers' bespoke information needs for informed classroom integration of ChatGPT. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–19.

[58] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, et al. 2024. Debugbench: Evaluating debugging capability of large language models. *arXiv preprint arXiv:2401.04621* (2024).

[59] Ektor Vrettakis, Christos Lougiakis, Akrivi Katifori, Vassilis Kourtis, Stamatis Christoforidis, Manos Karvounis, and Yannis Ioanidis. 2020. The story maker-an authoring tool for multimedia-rich interactive narratives. In *Interactive Storytelling: 13th International Conference on Interactive Digital Storytelling, ICIDS 2020, Bournemouth, UK, November 3–6, 2020, Proceedings 13*. Springer, 349–352.

[60] 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.

[61] Martin Weyssow, Houari A. Sahraoui, and Bang Liu. 2022. Better Modeling the Programming World with Code Concept Graphs-augmented Multi-modal Learning. *2022 IEEE/ACM 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)* (2022), 21–25. https://api.semanticscholar.org/CorpusID:245837887

[62] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).

[63] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. 2024. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. In *Generative ai for effective software development*. Springer, 71–108.

[64] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2015. Voyager: Exploratory analysis via faceted browsing of visualization recommendations. *IEEE transactions on visualization and computer graphics* 22, 1 (2015), 649–658.

[65] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. 2024. Hallucination is inevitable: An innate limitation of large language models. *arXiv preprint arXiv:2401.11817* (2024).

[66] Xin Yin, Chao Ni, Tien N Nguyen, Shaohua Wang, and Xiaohu Yang. 2024. Rectifier: Code translation with corrector via llms. *arXiv preprint arXiv:2407.07472* (2024).

[67] Yifeng Zhu, Jonathan Tremblay, Stan Birchfield, and Yuke Zhu. 2020. Hierarchical Planning for Long-Horizon Manipulation with Geometric and Symbolic Scene Graphs. *2021 IEEE International Conference on Robotics and Automation (ICRA)* (2020), 6541–6548. https://api.semanticscholar.org/CorpusID:229156165

A Architecture Prompting

Our chain-of-abstractions architecture utilizes LLM prompting for the generation and modification of abstractions. In this section, we document the prompts used in SimStep.

A.1 Forward Abstraction Generation

All graphical abstractions are represented as mermaid.js graphs. In prompts that include the graphical structure outputs, we specify this format:

```
Use mermaidjs. The format of the mermaidjs
    graph should have each node with a label
    surrounded by square brackets, and each
    link with a label surrounded by vertical
    bars. Each line defining a node or link
    should begin with 4 spaces. Don't add any
     addition styling nor any blank lines.
    More specific instructions: The diagram
    must start with graph LR (or graph TD).
    Each node should be defined as NodeID[
    Node Label] with a single space before it
    . Each edge should be defined as SourceID
     -->|Edge Label| TargetID without any
    extra spaces between the arrow and the |.
     Do not include any Markdown formatting (
    no triple backticks, no language tags).
    Only output the raw Mermaid code.
```

A.1.1 Concept Graph. The following prompt is used to generate the concept graph using the user's learning content.

```
Given the following learning content,
    generate a conceptual diagram of this
    text.

The graph should meet the following
    requirements:
- The nodes of the graph represent physical
    objects presented in the learning content
    .
- The links represent the relationships
    connecting them.
- The graph contains relationship that are
    not necessarily stated in the learning
    content but can be inferred.
- The graph is as abstract and simple as
    possible.
- If the graph has math or numbers involved,
    they are included.
- Each node has a description in square
    brackets.
- Each link a description that explains its
    meaning.

NO Explanation.
${mermaidDirections}
 ${learningContent}
```

A.1.2 Scenario Graph. The following prompt is used to generate the scenario graph using the concept graph and the user's chosen scenario.

```
Given the concept graph and scenario, change
    the names of all the nodes in the graph
    to represent the specific objects that
    the scenario involves. I'm trying to
    understand how the scenario can represent
     the relationship that the graph explains
    . Each node should have an updated is
    that is specific to this scenario.

A good response:
- Keeps the ids of the nodes in the graph the
     same (ids being the values in square
    brackets)
NO Explanation.
${mermaidDirections}
Concept graph:${graph}
Scenario:${scenario}
```

A.1.3 Learning Goal Graph. The following prompt is used to generate the learning goal graph using the scenario graph and the user's chosen learning goal (labeled hypothesis below).

```
Given the concept graph and learning goal,
    give me an updated graph (also using
    mermaidjs) that only contains nodes and
    links that pertain to the given learning
    goal (in other words, remove unnecessary
    nodes and links).
NO Explanation
${mermaidDirections}
Concept Graph:${graph}
Learning Goal:${hypothesis}
```

A.1.4 User Interaction Graph. The following prompt is used to generate the user interaction graph using the learning goal graph and the experimental procedure. Subsequent subsections describe how this procedure is generated.

```
Given an experimental procedure and concept
    graph, create a UI interaction graph of a
     web-based interactive simulation for
    this procedure.
Consider what the main experimental object is
     in this experiment, along with the
    dependent and independent variables.
What is the best way to visually show the
    experimental object within the procedure?
Include nodes for:
- all nodes in the given concept graph that
    you think are necessary for following the
     procedure and learning the conclusion of
     the learning goal. Keep the same ids and
     labels for these nodes as were in the
    original concept graph.
- UI controls necessary for following the
    procedure
Include edges for:
- all edges in the given concept graph that
    you think are necessary for following the
     procedure and learning the conclusion of
     the learning goal.
- any more relationships between visuals
- relationships between the UI controls and
    the visuals

Give the graph using mermaid js. NO
    explanation.
${mermaidDirections}

Graph:${graph}
Procedure:${proc}
Learning Goal:${hypothesis}
```

Descriptive Learning Goals: When the selected learning goal is descriptive, procedure is generated by first identifying the independent and dependent variables in question:

```
In an experiment testing the provided
    hypothesis using the laws explained in
    the provided concept graph, what is the
    main experimental object that we are
    interacting with in this experiment? NO
    explanation.

Concept Graph:${graph}
Hypothesis: ${hypothesis}
```

```
In an experiment testing the provided
    hypothesis using the laws explained in
    the provided concept graph, what is the
    dependent variable of this experiment? NO
     explanation and don't put a period at
    the end.

Concept Graph:${graph}
Hypothesis: ${hypothesis}
```

Using this information, we prompt for a procedure:

```
You are an expert in designing experimental
    procedures for interactive simulations.
    Given the concept graph and learning goal
     that you are trying to test, generate a
    simple procedure testing what effect ${
    indep} has on ${dep} as indicated by the
    learning goal using the concept graph.

A good procedure:
- Comes to the conclusion of the learning
    goal
- Is simple to follow
- Outlines any data collection that you want
    to perform

NO explanation

Concept Graph:${graph}
Learning Goal:${hypothesis}
```

Explanatory: When the selected learning goal is explanatory, procedure is generated by again identifying the independent and dependent variables in question, then prompting for a procedure:

```
Based on the concept graph, what underlying
    process explains why ${indep} has an
    effect on ${dep}? Give me one single
    phrase. NO explanation and don't put a
    period at the end.

Concept Graph:${graph}
```

```
You are an expert in designing experimental
    procedures for interactive simulations.
    Given the concept graph and learning goal
     that I am trying to test, give me a
    simple procedure testing how ${exp} as
    indicated by the learning goal using the
    concept graph.

A good procedure:
- Comes to the conclusion of the learning
    goal
- Is simple to follow
- Outlines any data collection that you want
    to perform

NO explanation.

Concept Graph:${graph}
Learning Goal:${hypothesis}
```

Procedural: When the selected learning goal is procedural, we first prompt for the experimental object:

```
In an experiment testing the provided
    hypothesis using the laws explained in
    the provided concept graph, what is the
    main experimental object that we are
    interacting with in this experiment? NO
    explanation.

Concept Graph:${graph}
Hypothesis: ${hypothesis}
```

Then we prompt for the process in question:

```
In an experiment testing the provided
    hypothesis using the laws explained in
    the provided concept graph, what is the
    porocess that the ${obj} goes through? NO
     explanation and don't put a period at
    the end.

Concept Graph:${graph}
Hypothesis: ${hypothesis}
```

And finally, we use this information to prompt for a procedure:

```
You are an expert in designing experimental
    procedures for interactive simulations.
    Given the concept graph and learning goal
     that I am trying to test, give me a
    simple procedure for running an
    interactive simulation showing how ${obj}
     goes through ${proc}.

A good procedure:
- Comes to the conclusion of the learning
    goal
- Is simple to follow
- Outlines any data collection that you want
    to perform

NO explanation.

Concept Graph:${graph}
Learning Goal:${hypothesis}
```

A.1.5 Code. We purposefully want SimStep simulations to look low-fidelity. In order to achieve this look, we use rough.js. Therefore, any prompts generating code include the following information about rough.js:

```
Imports and uses roughjs (https://unpkg.com/
    roughjs@latest/bundled/rough.js) for the
    entire app, including UI controls. Make
    sure to include a canvas element in the
    html to reference as the rough.js canvas
    in your script.
```

The following prompt is used to generate the simulation code:

```
Create a web-based interactive simulation
    based on the UI Interface Graph. Do your
    best to interpret which nodes represent
    visuals, which represent UI controls &
    data collection mechanisms, and give each
     UI control its desired function.
Include:
- A title and descripton of the experiment.
    The description should act as an
    introduction for the students to the
    experiment and what it teaches. In this
    description include any definitions that
    directly relate to the learning goal. Don
    't just outright state the learning goal,
     we want students to figure this out on
    their own.
- Integrate any instruction necessary. It
    should be clear how students should
    interact with the simulation.
```

```
- All nodes representing visuals and
    phenomena in a visual display of the
    experiment.
- All nodes representing UI controls & data
    collection below the visual display. You
    must label what each UI control
    represents. If a control represent an
    amount, make sure to provide concrete
    experimental units.
- ${roughDirections}

Make sure to include ALL nodes in the UI
    interface graph and give them their
    desired purpose
When inplementing the relationship between
    any two nodes, think about what
    attributes of each node define the
    functional reationship between them.
Keep track of these attributes in your script
    .

The simulation is for middle school kids, so
    make the visuals fun and engaging.
The UI controls should invoke expressive
    animations in the visual display.
Generate SVGs that are as realistic as
    possible and use gradients and additional
     shapes if needed for any necessary
    experimental objects. Rather than placing
     SVGs directly in the code, clearly
    define each as a variable.
For all text, either use the font cabin-
    sketch-regular or cabin-sketch-bold. In
    order to use these fonts, add <link rel="
    preconnect" href="https://fonts.
    googleapis.com"> <link rel="preconnect"
    href="https://fonts.gstatic.com"
    crossorigin> <link href="https://fonts.
    googleapis.com/css2?family=Cabin+Sketch:
    wght@400;700&family=Londrina+Sketch&
    family=Roboto:ital,wght@0
    ,100;0,300;0,400;0,500;0,700;0,900;1,100;1,300;1,4
    display=swap" rel="stylesheet"> to the
    HTML code.


Give the HTML, CSS and any necessary JS
    interactivity.
Make sure the code displays all visual
    elements correctly and no uncaught errors
     occur when it is run.

Add comprehensive logging for debugging
    purposes ONLY. Include the following
    debug logging system in your code:
```

```
// Debug logging system - only active
    when window.LOG_DEBUG is true
function logDebug(message) {
    if (window.LOG_DEBUG) {
        console.log(\`DEBUG [\${new Date
            ().toISOString()}]: \${
            message}\`);
    }
}

// Add this in your initialization
document.addEventListener('
    DOMContentLoaded', () => {
    if (window.LOG_DEBUG) {
        logDebug('Simulation initialized
            ');
        // Log all SVG elements
        document.querySelectorAll('svg').
            forEach((svg, index) => {
            const rect = svg.
                getBoundingClientRect();
            logDebug(\`SVG #\${index}:
                Position {x: \${rect.x},
                y: \${rect.y}}, Size {w:
                \${rect.width}, h: \${
                rect.height}}\`);
        });
        // Log UI controls
        document.querySelectorAll('input,
             button, select').forEach((
            control) => {
            logDebug(\`Control \${control
                .id || 'unnamed'} (\${
                control.tagName}):
                Initial value: \${control
                .value || 'N/A'}\`);
        });

        // Add event listeners for
            logging interactions
        document.querySelectorAll('button
            ').forEach((button) => {
            button.addEventListener('
                click', () => logDebug(\`
                Button \${button.id || '
                unnamed'} clicked\`));
        });
        document.querySelectorAll('input
            ').forEach((input) => {
            input.addEventListener('
                change', (e) => logDebug
                (\`Input \${input.id || '
                unnamed'} changed to \${e
                .target.value}\`));
        });
```

```
        }
    });

    // For animations and calculations, add
        logging in those functions
    // Example: logDebug(\`Calculated \${
        variable} = \${formula} = \${result
        }\`);
    // Example: logDebug(\`Element \${id}
        moved to {x: \${newX}, y: \${newY
        }}\`);

IMPORTANT: This logging should only be active
     when window.LOG_DEBUG is true, so it won
    't affect normal usage.
The default state MUST be window.LOG_DEBUG =
    false; in your code.
Make sure to call logDebug() in key places of
     your code to track:
- All SVG elements' positions and sizes
- UI control state changes
- User interactions with timestamps
- Animation frame-by-frame updates
- Mathematical calculations with formulas
- Any errors or warnings

NO explanation.

UI Interface Graph::${graph}
Learning Goal:${hypothesis}
```

A.2 User Direction

A.2.1 Scenario Options. Below is the prompt used to generate potential scenarios given the concept graph.

```
Give me 8 contexts that I can use to teach
    the concepts involved in the concept
    graph.
By context, I mean an instantiation of the
    concepts involved in the concept graph.
For each context, write a title enclosed in
    double curly braces {{title}} without
    numbering and a few words on how it
    presents the concept and how the teacher
    should use the contexts to teach high
    school kids about this concept.
Don't talk about classroom activities or
    student activities.
MAKE SURE to separate each context with "|".

Concept graph:${graph}
```

A.2.2 Learning Goal Options. Below is the prompt used to generate potential learning goals given the scenario graph.

```
Give me 6 concise and testable learning goals
     that I can use to teach high schoolers
    about the concept involved in the concept
     map. When forming goals, please focus on
     a few specific nodes in the concept
    graph and how the links between them
    represent relationships. Phrase these
    goals as a single statement that you want
     the student to learn by looking at the
    concept map, sort of like a hypothesis.
    For each learning goal, write the goal in
     double curly braces {{title}} without
    numbering and a brief description of what
     learning gains that goal will lead to.
    If the goal just describes a process,
    begin the desciption with "1.", if the
    goal explains why a process works in the
    way it does, begin the description with
    "2.", and if the goal explains an overall
     process, begin the description with
    "3.". MAKE SURE to separate each learning
     goal from the next with "|".

Concept Graph: ${graph}
```

A.3 Abstraction Modification

A.3.1 Suggesting a Code Change. The following prompt is used to suggest a class of code change based on a chat or error description message.

```
In a previous prompt, you generated the
    provided HTML code for an interactive
    simulation based on the provided UI Map.

However, there's an issue with this code/UI
    map: ${prompt}.

This issue may also be outlined via the red
    annotations in the provided image, which
    are labeled with labels such as "A1" or "
    A2" near the annotation.
These labels may be referenced in the above
    issue description.

Given the issue mentioned above, what type of
     change to the code would you would need
    to make in order to fix this issue?
    Choose from the following types of
    changes:
1. Add new variable/visual component to the
    code.
2. Add new function that acts on/using
    specific variables/visual components to
    the code.
```

```
3. Remove a variable/visual component from
    the code.
4. Remove a function that acts on/using
    specific variables/visual components from
     the code.
5. Completely re-implement a variable/visual
    component in the code.
6. Completely re-implement a function in the
    code.
7. Change an SVG in the code.
8. Change the implementation of some
    preexisting functionality.

Given me just the number associated with the
    type of change you would make.
NO EXPLAINATION.
```

A.3.2 Populate Remove Edge. If the chat suggests removing an edge, we populate a chat widget describing the change to the user. The following prompt is used to get the necessary information.

```
In a previous prompt, you generated HTML code
     for an interactive simulation based on
    the provided UI Map.

However, there's an issue with this code/UI
    map: ${prompt}.

This issue may also be outlined via the red
    annotations in the provided image, which
    are labeled with labels such as "A1" or "
    A2" near the annotation.
These labels may be referenced in the above
    issue description.

We think removing an edge from the UI Map
    might solve the above issue conceptually.
We want some more information on the edge
    that should be removed.

Please respond in the following format:
{ type: 8, message: <SHORT message explaining
     your suggestion. Use full, eloquent
    sentences and first person.>, source: <id
     of the source node>, target: <id of the
    target node>, label: <the label of the
    edge to delete>}

NO EXPLAINATION.
```

A.3.3 Populate Remove Node.

```
In a previous prompt, you generated HTML code
     for an interactive simulation based on
    the provided UI Map.

However, there's an issue with this code/UI
    map: ${prompt}.

This issue may also be outlined via the red
    annotations in the provided image, which
    are labeled with labels such as "A1" or "
    A2" near the annotation.
These labels may be referenced in the above
    issue description.

We think removing a node from the UI Map
    might solve the above issue conceptually.
We want some more information on the node
    that should be removed.

Please respond in the following format: {
    type: 7, message: <SHORT message
    explaining your suggestion. Use full,
    eloquent sentences and first person.>,
    node: <id of the node to delete>}

NO EXPLAINATION.
```

A.3.4 Populate Edit Edge.

```
In a previous prompt, you generated HTML code
     for an interactive simulation based on
    the provided UI Map.

However, there's an issue with this code/UI
    map: ${prompt}.

This issue may also be outlined via the red
    annotations in the provided image, which
    are labeled with labels such as "A1" or "
    A2" near the annotation.
These labels may be referenced in the above
    issue description.

We think changing the label of an edge in the
     UI Map might solve the above issue
    conceptually.
We want some more information on the edge
    that should be changed.

Please respond in the following format:
```

```
{ type: 6, message: <SHORT message explaining
    your suggestion. Use full, eloquent
    sentences and first person.>, source: <id
    of the source node>, target: <id of the
    target node>, oldLabel: <the old label of
    the edge to change>, newLabel: <the new
    label>}

NO EXPLAINATION.
```

A.3.5 *Populate Edit Node.*

```
In a previous prompt, you generated HTML code
    for an interactive simulation based on
    the provided UI Map.

However, there's an issue with this code/UI
    map: ${prompt}.

This issue may also be outlined via the red
    annotations in the provided image, which
    are labeled with labels such as "A1" or "
    A2" near the annotation.
These labels may be referenced in the above
    issue description.

We think changing the label of a node in the
    UI Map might solve the above issue
    conceptually.
We want some more information on the node
    that should be changed.

Please respond in the following format:
{ type: 5, message: <SHORT message explaining
    your suggestion. Use full, eloquent
    sentences and first person.>, node: <id
    of the node to change>, oldLabel: <the
    old label of that node>, newLabel: <the
    new label>}

NO EXPLAINATION.
```

A.3.6 *Populate Redraw.*

```
In a previous prompt, you generated the
    provided HTML code for an interactive
    simulation.

However, there's an issue with this code/UI
    map: ${prompt}.

This issue may also be outlined via the red
    annotations in the provided image, which
    are labeled with labels such as "A1" or "
    A2" near the annotation.
These labels may be referenced in the above
    issue description.

We think editing one of the SVGs in the code
    might solve the above issue.
We want some more information on how we
    should update the SVG.

Please respond in the following format:
{ type: 4, message: <SHORT message explaining
    your suggestion. Use full, eloquent
    sentences and first person.>, box: <a box
    surrounding the object that is being
    redrawn in the provided image. First
    consider which object needs to be redrawn
    . Then, find the EXACT location and size
    of this object by looking at both the
    provided image and the html code. Based
    on this location and size, generate a box
    that surrounds this object using the
    format [start_x, start_y, width, height]
    that uses pixels as its units (e.g.
    [0,0,50,50] would be a 50x50px box in the
    top left corner of the image). When
    applied on top of the image, the box
    should completely encompass the object.>,
    svg: <simple svg representing the new
    visual, approx 100px by 100px> }

NO EXPLAINATION.
```

A.3.7 *Populate Edit Assumptions.*

```
In a previous prompt, you generated HTML code
    for an interactive simulation based on
    the provided UI Map.

However, there's an issue with this code/UI
    map: ${prompt}.

This issue may also be outlined via the red
    annotations in the provided image, which
    are labeled with labels such as "A1" or "
    A2" near the annotation.
These labels may be referenced in the above
    issue description.

We think editing the details of one of the
    nodes in the UI Map might solve the above
    issue.
```

```
We want some more information on how we
    should edit these details to fix the
    issue.

Please respond in the following format:
{ type: 3, message: <SHORT message explaining
    your suggestion. Use full, eloquent
    sentences and first person.>, node: <id
    of the node whose assumptions are being
    updated>, assumptions: <list of
    assumptions updated to explicitly fix the
    issue> }

NO EXPLAINATION.
```

A.3.8 *Populate Add Edge.*

```
In a previous prompt, you generated HTML code
    for an interactive simulation based on
    the provided UI Map.

However, there's an issue with this code/UI
    map: ${prompt}.

This issue may also be outlined via the red
    annotations in the provided image, which
    are labeled with labels such as "A1" or "
    A2" near the annotation.
These labels may be referenced in the above
    issue description.

We think adding an edge to the UI Map might
    solve the above issue conceptually.
We want some more information on the edge
    that should be added.

Please respond in the following format:
{ type: 2, message: <SHORT message explaining
    your suggestion. Use full, eloquent
    sentences and first person.>, source: <id
    of the source node>, target: <id of the
    target node>, label: <new edge name>}

NO EXPLAINATION.
```

A.3.9 *Populate Add Node.*

```
In a previous prompt, you generated HTML code
    for an interactive simulation based on
    the provided UI Map.

However, there's an issue with this code/UI
    map: ${prompt}.

This issue may also be outlined via the red
    annotations in the provided image, which
    are labeled with labels such as "A1" or "
    A2" near the annotation.
These labels may be referenced in the above
    issue description.

We think adding a node to the UI Map might
    solve the above issue conceptually.
We want some more information on the node
    that should be added.

Please respond in the following format:
{ type: 1, message: <SHORT message explaining
    your suggestion. Use full, eloquent
    sentences and first person.>, label: <new
    node name> }

NO EXPLAINATION.
```

```
Your designers want to know what nodes and
    links in the given UI interactivity graph
    correspond to the region(s) circled in
    red in your prototype.
Given an image, the code used to generate the
    prototype in the image, and a graph, you
    respond with the subgraph corresponding
    to the elements and relationships these
    region(s) represents.

A good subgraph has the:
- nodes corresponding to the visuals/elements
    circled in red
- nodes corresponding to the attributes of
    the elements circled in red
- links corresponding to the relationships
    between all included nodes
- original label for EVERY node and edge

NO explanation. Only respond with the
    mermaidjs graph.
```

A.3.10 *Code Assumptions.*

The following prompt is used to indentify the current code assumptions abstraction.

```
Given the following html code and the UI
    Interactivity Map that it represents,
    please give me a json object with an
    attribute for each node in the UI Map.
    The name of the each attribute should be
    the label, or value in square brackets,
    of each node (DO NOT NAME IT THE NAME IN
    SQUARE BRACKETS). The value for each of
    these attributes should be a list of all
    of the details and assumptions that the
    code is making about this node. For
    example, the numeric range of a node
    representing a slider, attributes of a
    phypical display or graph, all formulas
    used to caluculate displayed or output
    values, and the way physical objects are
    visually represented.

A good list of assumptions:
- Doesn't reference specific variables in the
    code but instead speaks generically
- Pays close attention to the implementation
    in order to identify assumptions that are
    not intended (i.e. bugs)

NO explaination.

Html Code: ${htmlCode}
UI Map: ${UIMap}
```

The following prompt is used to update the code based on the user's changes to the code assumptions.

```
The given html code is the implementation of
    an experiment based on the following
    concept graph. Please update the html
    code so that it implements the details
    outlined in the given details list for
    the ${node} object.

A good response:
- contains all the same functionality of the
    original html code
- updates the code so that ${node} now abides
    by the provided list of details
- ${roughDirections}

NO explanation.

Graph:${graph}
Html Code:${htmlCode}
Node of interest: ${node}
Details:${JSON.stringify(newAssumptions)}
```

Redraw

The following prompt is used to redraw a visual when the LLM identifies that the user wants to update it (from a chat or error description message).

```
Given the above low-fidelity hand-drawn image
    , create a high fidelity SVG representing
    the object that the hand-drawn image is
    of.

A good SVG response:
- interprets the object that the hand-drawn
    image is of (typically this object is one
    of the objects included as nodes in the
    provided UI Map)
- has similar shape and features to the hand
    drawn image
- uses gradients and additional shapes if
    needed
- ignores the fact that the hand-drawn sketch
    is drawn in red

NO EXPLAINATION
```

The following prompt updated the simulation code to use a new svg for a specific visual.

```
Given the above image and html code, replace
    the SVG circled in red in the image with
    the provided SVG in the code. Respond
    with the ENTIRE updated HTML code and
    nothing selse.

A good response MUST:
- have a the old circled SVG replaced with
    the new provided SVG
- resize the new SVG so it is the same size
    as the original (not just cropping the
    new SVG, but actaully reducing/increasing
    its attributes)
- have all of the functionality of the
    original code
- ${roughDirections}

NO EXPLAINATION
```

A.3.11 Auto-Add Edges. Below is the prompt used to automatically add links when a new node is added by the user.

```
Please add a new node to the following
    mermaid graph with the provided label.
    Add any edge that you think reasonably
    connect this new object to the rest of
    the graph.

NO explaination.
```

```
${mermaidDirections}

Graph: ${UIMap}
Label: ${newNodeName}
```

A.4 Automated Testing

The following prompt is used to generate test cases for automated testing.

```
You had previously created the following HTML
    code based on the UI Map and learning
    goal.
More info:
- HTML Code: ${htmlCode}
- Learning Goal: ${selectedHypothesis}
- UI Map: ${UIMap}.
Please generate a JSON array of structured
    test cases for the interactive simulation
    .
Each test case should be a JSON object with
    the following keys:
- uiElementId: The ID of the UI element to
    interact with.
- actionType: The type of action (one of "
    click", "set_value", "toggle", "
    verify_content").
- actionValue (optional): The value to set (
    if applicable).
- description: A brief description of what is
    being tested for.
- expectedOutcome: A brief description of the
    expected result.
- isUIVerification: A boolean indicating
    whether this test case is related to
    changes in any of the UI components or
    visual elements in the simulation.
Return only the JSON array, with no
    additional text, and ensure that the JSON
    starts between <START> and <STOP> tags.
```

Below is the prompt used to verify test results using debug log information and Puppeteer screenshots.

```
You had previously created the following HTML
    code based on the UI Map and learning
    goal.
More info:
- HTML Code: ${htmlCode}
- UI Map: ${UIMap}
- Learning Goal: ${selectedHypothesis}
- JavaScript Errors captured: ${.join('; ')}.
${debugLogsText}
${initialScreenshotNote}

Note: An initial screenshot of the UI (before
    any interactions) is available.

Based on the above context, including the
    test case results, screenshots and
    runtime logs, do the following:
1) Verify whether the actual outcomes in each
    test case (as described below) match the
    expected outcomes.
2) If discrepancies or errors remain, update
    the HTML code so that all test cases pass
    and errors are resolved.
3) Verify that the simulation satisfies the
    learning goal.
IMPORTANT: Return only the updated HTML code,
    starting between <START> and <STOP> tags
    , or return "PASS" if no changes are
    needed.

Here is the:
1) Structured list of test case results and
    verification details,
2) Indication of whether the HTML code needs
    to be changed.
If no changes are needed, simply return "PASS
    ".
If changes are required, return the updated
    HTML code starting between <START> and <
    STOP> tags.
```

The following prompt is used to fix js errors in the simulation code when they are present.

```
You had previously created the following HTML
    code based on the UI Map and learning
    goal.
More info:
- HTML Code: ${htmlCode}
- UI Map: ${UIMap}.
However, the HTML code is generating these
    JavaScript errors: ${errorMessages.join
    ('; ')}.
Please update the HTML code so that these
    errors are fixed while preserving all the
    original functionality.
Return only the updated HTML code, starting
    between <START> and <STOP> tags.
```