From Code Generation to Computational Design: Generative AI as an Interactive Partner for Future Computer Science Education

Yichen Andy Yu[1][0009-0001-0175-3253] and Qiao Jin[1][0000-0001-5493-1343]

Department of Computer Science, North Carolina State University, Raleigh, NC, USA
{yyu55,qjin4}@ncsu.edu

Abstract. Recent advances in generative AI have made it easier for students to generate code, debug errors, and receive programming explanations through natural language prompts. However, code generation alone does not ensure that students understand the computational reasoning behind a program. Current AI coding tools may help students produce syntactically correct solutions while bypassing key learning processes such as problem decomposition, abstraction, algorithmic reasoning, debugging, and evaluation.
In this position paper, we argue that generative AI for computer science education should move beyond answer generation and become a computational design partner. Such systems should help learners express computational intent, transform that intent into inspectable intermediate representations, simulate program behavior, and support iterative refinement. We propose a framework in which students express intent, AI generates plans or scaffolds, students inspect and revise behavior, and the system explains the consequences of their changes. Our goal is to support students in learning how to design, reason about, and evaluate computational systems rather than simply obtain final code.

Keywords: Generative AI · Computer Science Education · Human-AI Interaction · Computational Thinking · Learning Interfaces

1 Introduction

Recent advances in generative AI have changed how students write code. Large language models can generate programs, explain errors, suggest test cases, and provide debugging support from natural language prompts [6,21,11,1,17]. While these tools can lower syntax-level barriers, producing code is not the same as learning computer science.

A central goal of CS education is to help students develop computational thinking, including problem decomposition, abstraction, algorithmic reasoning, debugging, and evaluation [27,7,20]. When AI directly returns complete code, students may skip the reasoning process needed to understand how a program

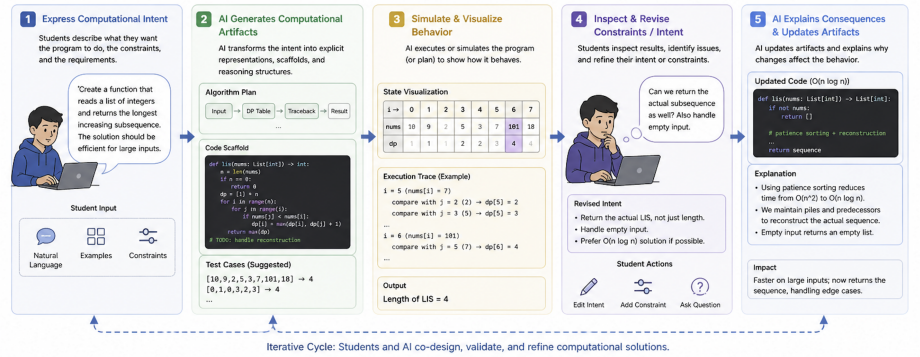

Fig. 1. Overview of the proposed computational design partner framework. Students express computational intent, AI generates inspectable computational artifacts, students simulate and inspect behavior, revise constraints, and receive explanations of behavioral consequences.

works, why it works, and how it might fail [1,17]. Current AI coding interfaces often move from a natural language prompt directly to code, leaving few opportunities for students to inspect intermediate reasoning.

We therefore propose a shift from using generative AI as an answer generator toward designing it as a computational design partner. In this paradigm, students express computational intent through natural language, examples, diagrams, partial code, or expected behaviors. The AI then transforms this intent into inspectable intermediate representations, such as problem decompositions, state machines, pseudocode, execution traces, simulations, test cases, or code scaffolds. Prior work on program visualization and educational programming environments shows that making computational structures visible can support learner understanding [22,9,18,26].

This paper presents a design framework for generative AI-supported CS education. We argue that future learning interfaces should help students externalize intent, convert that intent into explicit reasoning constraints, inspect program behavior, and iteratively revise their computational designs. Our goal is not to replace programming practice, but to help students learn how to design, reason about, and evaluate computational systems with AI as an interactive partner.

2 Related Work

2.1 Generative AI for Programming and CS Education

Generative AI tools have become increasingly capable of supporting programming tasks. Students can use these systems to generate code snippets, translate natural language into functions, explain errors, produce examples, and receive step-by-step guidance [21,11,6,1]. These systems can reduce frustration and help learners overcome syntax-level obstacles.

However, current AI coding tools also raise educational concerns. When AI produces complete solutions, students may skip the reasoning process that programming assignments are designed to teach [1,17]. A generated answer may pass tests while still leaving the learner unable to explain why the solution works, how it could fail, or how it should be adapted to a different problem. This is especially problematic for novice learners, who often lack the expertise to judge whether generated code is appropriate [20,12].

Existing interactions also tend to center on a narrow input-output pattern: the learner writes a prompt, and the AI returns code or explanation. While useful, this pattern does not necessarily expose intermediate reasoning. It also provides limited support for comparing alternative designs, visualizing program behavior, or diagnosing mismatches between a student's mental model and the actual execution of the program [22,25]. Recent studies on students' use of ChatGPT and Copilot suggest that students value these tools, but also raise questions about overreliance, verification, and the need for pedagogically grounded integration [17,8,2,10].

Therefore, we argue that the educational value of generative AI should be evaluated not only by whether it helps students produce correct code, but also by whether it helps them build durable computational understanding.

2.2 Learning Computational Thinking Through Intermediate Representations

Computer science education has long relied on intermediate representations to support learning. Flowcharts, pseudocode, state diagrams, memory diagrams, execution traces, test cases, block-based interfaces, and visual simulations help students understand concepts that are difficult to grasp from code alone [9,14,19,18,26,22]. These representations are especially important because many CS concepts are dynamic and invisible. Recursion unfolds through call stacks; loops depend on changing state; references involve hidden object identity; concurrency involves interleavings that are hard to observe directly.

However, creating high-quality intermediate representations is time-consuming for instructors and difficult for students. Static diagrams may not reflect a learner's own code, and generic examples may not address the specific misunderstanding a student has. Generative AI creates an opportunity to generate personalized intermediate representations from a student's current intent, code, or mistakes [21,11,13].

The challenge is designing systems that generate these representations in a way that supports learning rather than replacing reasoning. Learning sciences research suggests that scaffolding, self-explanation, cognitive load, and adaptive feedback are central to supporting deeper understanding [28,4,24,5]. We therefore propose that generative AI should act as a computational design partner: a system that helps students externalize, inspect, and revise computational ideas.

3 Computational Design Partner Framework

3.1 Design Goals

Based on the research gaps identified above, we propose three design goals for future generative AI-supported CS learning interfaces.

- **Enable Computational Intent Expression for Learners.** Lower the barrier for students to express what they are trying to build before they know how to fully implement it. The system should allow students to describe goals through natural language, examples, diagrams, sketches, partial code, or expected behaviors. Rather than requiring students to immediately write syntactically correct code, the system should help them externalize incomplete computational ideas. This goal builds on the broader role of external representations in design and learning [23,3,22].
- **Support Controllable Code Generation via Reasoning Constraints.** Transform learner intent into explicit reasoning constraints that guide generation. Instead of relying solely on one-shot natural language prompts, the system should leverage student-provided goals, edge cases, invariants, state transitions, and test conditions to guide generated plans, scaffolds, and code. This can improve controllability and reduce the risk of treating AI-generated code as a black-box answer [1,17,21].
- **Facilitate Iterative Human-AI Computational Design.** Support a continuous learning loop in which students can inspect generated plans, simulations, traces, or code scaffolds; revise their constraints; and observe how changes affect behavior. The system should enable students to progressively refine computational systems while maintaining agency, interpretability, and responsibility for design decisions. This goal aligns with prior work on scaffolding, self-explanation, and formative feedback [28,4,5,15].

3.2 Framework Overview

Our framework treats generative AI as a medium for computational design communication rather than a direct code-generation tool. Instead of moving from a prompt directly to final code, students move through an iterative learning loop that makes computational reasoning visible and revisable.

The loop consists of five stages: students first express their computational intent through natural language, examples, diagrams, partial code, or expected behaviors. The AI then translates this intent into explicit reasoning constraints, including inputs, outputs, states, events, assumptions, edge cases, and test conditions. Based on these constraints, the system generates inspectable intermediate artifacts such as decompositions, pseudocode, state machines, execution traces, simulations, test cases, or code scaffolds. Students inspect these artifacts, identify mismatches between expected and actual behavior, and revise their constraints or code. Finally, the AI explains how each revision changes the program's behavior.

This process shifts CS learning from "write code until it passes" toward designing computational systems whose structure, assumptions, and behavior can be explained, tested, and refined.

Enable Computational Intent Expression for Learners Students often begin with an incomplete idea rather than a formal specification. Traditional programming environments require them to immediately translate this idea into code, which can be difficult when they do not yet know the relevant syntax or abstraction [20]. AI coding assistants reduce this barrier, but they often jump too quickly from informal intent to final implementation [17,1].

In our proposed framework, the first role of AI is to help students make their intent explicit. For instance, when a student describes a simple game mechanic, the system can identify entities such as player, enemy, obstacle, and goal; behaviors such as follow, avoid, stop, and return; and conditions such as distance thresholds or collision events. These extracted elements can be shown to the student as an editable computational intent map.

This process treats learner expression as structured computational input. In CS education, learner expressions can become structured computational prompts that guide subsequent reasoning and implementation. Instead of using AI to hide the design process, the system uses AI to make the learner's assumptions and goals visible.

Enabling Controlled Code Generation Through Computational Constraints Existing AI coding tools often generate code directly from a prompt. However, prompts can be ambiguous, and generated code may contain hidden assumptions [6,1]. To address this issue, our framework redefines student inputs as constraints and control signals within the generation process.

For example, a student learning sorting algorithms may specify that the algorithm should work in-place, preserve equal elements' order, or prioritize readability over performance. A student learning object-oriented design may specify that each object should manage its own state. A student learning networking may specify that updates must remain consistent across clients.

The system can convert such inputs into generation-oriented computational constraints, including but not limited to: input-output expectations, invariants, state transitions, edge cases, test cases, data structure constraints, performance or readability goals, and allowed or disallowed operations.

These constraints determine which aspects of the solution must be preserved and which aspects can be AI-generated. This transforms AI generation from an unconstrained answer into guided exploration within defined boundaries. By anchoring generation to explicit computational constraints, the system can reduce trial-and-error prompting while helping students understand the assumptions behind a solution.

Facilitating Iterative Human-AI Computational Design Programming is not a one-shot process. Students learn by testing ideas, encountering failures,

and refining their understanding [20,12,25]. Therefore, AI-supported CS learning interfaces should support iteration at the level of reasoning, not only code editing.

After the AI generates a plan, scaffold, or simulation, the student should be able to inspect it and revise specific constraints. For example, if an enemy AI moves too aggressively, the student can add a stopping distance constraint. If a recursive function produces unexpected results, the student can inspect the call stack and revise the base case. If a data structure fails for an edge case, the student can add a test and observe how the algorithm responds.

The system should then explain the consequence of each revision. This feedback should connect local changes to broader CS concepts. For example, changing a loop condition should be explained in relation to boundary cases; changing a variable assignment should be explained in relation to reference semantics; changing a state transition should be explained in relation to system behavior.

Through this process, students learn to design computational systems rather than merely request code.

4 Implementation

To instantiate the proposed framework, we envision a prototype learning interface for introductory programming and game development courses. The system is organized around an intent-first workflow rather than a code-first workflow. Students begin by describing a computational goal through natural language, examples, partial code, or behavioral constraints. The interface then uses a generative AI backend to extract key computational elements, including entities, inputs, outputs, states, events, conditions, and assumptions.

Instead of immediately returning final code, the system generates inspectable intermediate representations. Depending on the learning objective, these representations may include problem decompositions, pseudocode, state machines, execution traces, test cases, or code scaffolds. For example, when a student describes an enemy character that should chase the player, avoid obstacles, and return home, the system can first generate a state machine with states such as Idle, Chase, AvoidObstacle, and ReturnHome. Students can inspect the transitions, revise constraints, and ask questions about possible behaviors before code is generated.

The interface includes three main panels: an intent panel, a computational representation panel, and a feedback panel. The intent panel allows students to edit goals, constraints, and examples. The representation panel visualizes AI-generated artifacts such as state diagrams, traces, or scaffolds. The feedback panel explains how student revisions change the behavior of the program. Through this structure, the system positions generative AI as an interactive partner for computational design rather than a tool for directly producing answers.

5 Preliminary Evaluation Plan

Because this paper proposes a design framework rather than a completed deployed system, we outline a formative evaluation plan for future work. The initial goal would be to identify usability issues, learning opportunities, and breakdowns in the interaction loop rather than to make broad claims about learning outcomes. This follows formative evaluation traditions in HCI and educational technology, where early studies can identify usability issues and inform iterative design [15,16].

We plan to conduct a formative study with students enrolled in introductory programming or game development courses. Participants would complete programming tasks under two conditions: a conventional AI coding assistant condition and a computational design partner condition. Tasks would involve concepts such as state machines, recursion, data structures, or debugging.

Participants would be asked to: 1) express intent for a programming task; 2) inspect an AI-generated computational representation; 3) predict program behavior; 4) revise constraints or code; and 5) explain the effect of their revision.

We would measure both process and learning outcomes, including quality of problem decomposition, ability to explain generated code, accuracy of behavior prediction, debugging transfer to similar tasks, quality of abstraction, perceived agency and confidence, cognitive load, and usability. These measures are motivated by prior work showing relationships among tracing, explaining, and writing code, as well as learning sciences research on self-explanation and cognitive load [12,25,4,24].

The evaluation would focus on whether students better understand the computational structure of a solution, not merely whether they finish the task faster. In particular, we are interested in whether students can explain why an implementation works, identify assumptions embedded in generated code, revise constraints appropriately, and transfer debugging strategies to new problems.

6 Limitations and Future Work

Although the proposed framework highlights the potential of generative AI as a computational design partner, several limitations remain.

First, generative AI may produce incorrect or misleading intermediate representations. A wrong state machine, execution trace, or test case could harm learning if students trust it without critique. Future systems should therefore include transparency mechanisms, uncertainty indicators, and opportunities for students to challenge or verify AI outputs.

Second, balancing support and over-scaffolding is difficult. If the AI provides too much structure, students may still become passive. If it provides too little, novice learners may remain stuck. Future research should explore adaptive scaffolding strategies that adjust based on learner expertise, task difficulty, and demonstrated understanding.

Third, different CS topics require different representations. A state machine may be suitable for event-driven programming, while memory diagrams are more useful for references, and execution traces may be better for recursion. Future work should investigate how AI systems can select appropriate representations based on the learning objective.

Finally, evaluation should extend beyond short-term task success. Longitudinal studies are needed to determine whether computational design partner interfaces improve transfer, debugging ability, and independent problem solving. Future iterations may also explore collaborative learning scenarios where multiple students jointly define constraints, compare alternative designs, and negotiate system behavior with AI support.

7 Conclusion

In this paper, we propose a shift in how generative AI should be designed for future computer science education. Rather than treating AI primarily as a code generator, we argue that AI should serve as a computational design partner that helps students express, inspect, and refine computational reasoning.

Our framework supports a learning loop in which students express intent, AI forms computational constraints, generates inspectable scaffolds, students inspect behavior, and then revise their design with feedback on consequences. This approach reframes programming education from producing correct code toward designing computational systems.

By adapting the logic of intent expression and controllable generation to CS education, we show how generative AI can support deeper learning processes such as problem decomposition, abstraction, algorithmic reasoning, debugging, and evaluation thinking. The future of CS education should not be defined by students asking AI for answers, but by students learning to design, reason about, and evaluate computational systems with AI as an interactive partner.

References

1. Becker, B.A., Denny, P., Finnie-Ansley, J., Luxton-Reilly, A., Prather, J., Santos, E.A.: Programming is hard – or at least it used to be: Educational opportunities and challenges of ai code generation. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1. pp. 500–506. SIGCSE 2023, ACM, New York, NY, USA (2023). https://doi.org/10.1145/3545945.3569759
2. Budhiraja, R., Joshi, I., Challa, J.S., Akolekar, H.D., Kumar, D.: "it's not like jarvis, but it's pretty close!" – examining chatgpt's usage among undergraduate students in computer science (2024), https://arxiv.org/abs/2311.09651
3. Buxton, B.: Sketching User Experiences: Getting the Design Right and the Right Design. Morgan Kaufmann, San Francisco, CA, USA (2007)
4. Chi, M.T.H., de Leeuw, N., Chiu, M.H., LaVancher, C.: Eliciting self-explanations improves understanding. Cognitive Science **18**(3), 439–477 (1994). https://doi.org/10.1207/s15516709cog1803_3

5. Corbett, A.T., Anderson, J.R.: Knowledge tracing: Modeling the acquisition of procedural knowledge. User Modeling and User-Adapted Interaction **4**(4), 253–278 (1995). https://doi.org/10.1007/BF01099821
6. Finnie-Ansley, J., Denny, P., Becker, B.A., Luxton-Reilly, A., Prather, J.: The robots are coming: Exploring the implications of openai codex on introductory programming. In: Proceedings of the 24th Australasian Computing Education Conference. pp. 10–19. ACE '22, ACM, New York, NY, USA (2022). https://doi.org/10.1145/3511861.3511863
7. Grover, S., Pea, R.: Computational thinking in k–12: A review of the state of the field. Educational Researcher **42**(1), 38–43 (2013). https://doi.org/10.3102/0013189X12463051
8. Joshi, I., Budhiraja, R., Tanna, J., Jain, H., Desai, M., Rallapalli, S., Srivastava, S., Chava, S.: Chatgpt in the classroom: An analysis of its strengths and weaknesses for solving undergraduate computer science questions. In: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1. pp. 625–631. SIGCSE 2024, ACM, New York, NY, USA (2024). https://doi.org/10.1145/3626252.3630803
9. Kölling, M., Quig, B., Patterson, A., Rosenberg, J.: The bluej system and its pedagogy. Computer Science Education **13**(4), 249–268 (2003). https://doi.org/10.1076/csed.13.4.249.17496
10. Kosar, T., Ostojić, D., Zorjan, S., Mernik, M.: Computer science education in chatgpt era: Experiences from an experiment in a programming course for novice programmers. Mathematics **12**(5) (2024). https://doi.org/10.3390/math12050629
11. Leinonen, J., Hellas, A., Sarsa, S., Reeves, B., Denny, P., Prather, J., Becker, B.A.: Using large language models to enhance programming error messages. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1. pp. 563–569. SIGCSE 2023, ACM, New York, NY, USA (2023). https://doi.org/10.1145/3545945.3569770
12. Lister, R., Fidge, C., Teague, D.: Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. In: Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education. pp. 161–165. ITiCSE '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1562877.1562930
13. MacNeil, S., Tran, A., Hellas, A., Kim, J., Sarsa, S., Denny, P., Bernstein, S., Leinonen, J.: Comparing code explanations created by students and large language models (2023), https://arxiv.org/abs/2304.03938
14. Maloney, J., Resnick, M., Rusk, N., Silverman, B., Eastmond, E.: The scratch programming language and environment. ACM Transactions on Computing Education **10**(4) (2010). https://doi.org/10.1145/1868358.1868363
15. Nielsen, J.: Heuristic evaluation. In: Nielsen, J., Mack, R.L. (eds.) Usability Inspection Methods, pp. 25–62. John Wiley & Sons, New York, NY, USA (1994)
16. Nielsen, J., Landauer, T.K.: A mathematical model of the finding of usability problems. In: Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems. pp. 206–213. CHI '93, ACM, New York, NY, USA (1993). https://doi.org/10.1145/169059.169166
17. Prather, J., Reeves, B.N., Denny, P., Becker, B.A., Leinonen, J., Luxton-Reilly, A., Powell, G., Finnie-Ansley, J., Santos, E.A.: "it's weird that it knows what i want": Usability and interactions with copilot for novice programmers. ACM Transactions on Computer-Human Interaction **31**(1) (2023). https://doi.org/10.1145/3617367

18. Price, T.W., Barnes, T.: Comparing textual and block interfaces in a novice programming environment. In: Proceedings of the Eleventh Annual International Conference on International Computing Education Research. pp. 91–99. ICER '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2787622.2787712
19. Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., Kafai, Y.: Scratch: Programming for all. Communications of the ACM **52**(11), 60–67 (2009). https://doi.org/10.1145/1592761.1592779
20. Robins, A., Rountree, J., Rountree, N.: Learning and teaching programming: A review and discussion. Computer Science Education **13**(2), 137–172 (2003). https://doi.org/10.1076/csed.13.2.137.14200
21. Sarsa, S., Denny, P., Hellas, A., Leinonen, J.: Automatic generation of programming exercises and code explanations using large language models. In: Proceedings of the 2022 ACM Conference on International Computing Education Research. pp. 27–43. ICER '22, ACM, New York, NY, USA (2022). https://doi.org/10.1145/3501385.3543957
22. Sorva, J., Karavirta, V., Malmi, L.: A review of generic program visualization systems for introductory programming education. ACM Transactions on Computing Education **13**(4) (2013). https://doi.org/10.1145/2490822
23. Sutherland, I.E.: Sketchpad: A man-machine graphical communication system. In: Proceedings of the May 21–23, 1963, Spring Joint Computer Conference. pp. 329–346. AFIPS '63, ACM, New York, NY, USA (1963). https://doi.org/10.1145/1461551.1461591
24. Sweller, J.: Cognitive load during problem solving: Effects on learning. Cognitive Science **12**(2), 257–285 (1988). https://doi.org/10.1207/s15516709cog1202_4
25. Venables, A., Tan, G., Lister, R.: A closer look at tracing, explaining and code writing skills in the novice programmer. In: Proceedings of the Fifth International Workshop on Computing Education Research Workshop. pp. 117–128. ICER '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1584322.1584336
26. Weintrop, D., Wilensky, U.: Comparing block-based and text-based programming in high school computer science classrooms. ACM Transactions on Computing Education **18**(1) (2017). https://doi.org/10.1145/3089799
27. Wing, J.M.: Computational thinking. Communications of the ACM **49**(3), 33–35 (2006). https://doi.org/10.1145/1118178.1118215
28. Wood, D., Bruner, J.S., Ross, G.: The role of tutoring in problem solving. Journal of Child Psychology and Psychiatry **17**(2), 89–100 (1976). https://doi.org/10.1111/j.1469-7610.1976.tb00381.x