

## Conceptualizing Agentic Programming with Scratch: Enhancing Computational Thinking on a Higher Level of Abstraction

**Orit Hazzan**

Education in Science and Technology  
Technion, Israel Institution of Technology  
Haifa, Israel  
[oritha@technion.ac.il](mailto:oritha@technion.ac.il)

**Noa Ragonis**

Computer Science Studies  
Beit Berl College  
Beit Berl, Israel  
[noarag@beitberl.ac.il](mailto:noarag@beitberl.ac.il)

### Abstract

Agentic programming is becoming increasingly central to contemporary software and AI development. Rather than viewing programs as scripted procedures, agentic programming frames them as semi-autonomous systems that perceive their environment, pursue goals, make decisions, and adapt over time.

This paper proposes a pedagogical lens that uses Scratch, a block-based visual programming environment, to make agentic programming ideas more concrete and teachable. We argue that Scratch offers an intuitive space in which learners can conceptually model agentic behavior without relying on advanced AI infrastructure. Sprites function as agents, variables hold internal states, sensing blocks provide perception, broadcasts enable communication, and control structures create sense–think–act cycles that resemble agentic reasoning. Thus, Scratch becomes a shared language through which computer science and software engineering educators and practitioners can discuss and explore agentic systems.

We illustrate how familiar Scratch activities can be reframed to highlight agency, autonomy, decision-making, and interaction, thereby strengthening computational thinking on a higher level of abstraction. This reframing enables students to view programs as purposeful digital collaborators rather than passive executors of instructions, building mental models that connect traditional coding practices with emerging agentic approaches. In a future work, we plan to develop AgentScratch, which will include a Cognition Palette whose basic building blocks simulate key processes of agentic software development (Hazzan and Ragonis, 2026).

### 1. Introduction

Traditional programming is based on explicitly writing algorithms that execute predefined instruction sequences. In this paradigm, the programmer is responsible for specifying the logic, rules, and decision processes, and the computer simply follows these.

Over the last few decades, a shift has begun with the growth of machine learning approaches. A further conceptual step is emerging with the current development of experiences with Generative AI (GenAI) environments, particularly with agentic programming. Here, AI systems do not merely assist with writing code but operate as semi-autonomous agents that can plan, reason, select tools, generate and revise code, test their own outputs, and iteratively work toward goals. In this sense, agentic systems behave more like collaborators who analyze a problem, consider possible solutions, and take action rather than simply executing instructions.

The shift from viewing programs as linear, predetermined procedures to seeing them as autonomous, goal-directed entities introduces a level of conceptual abstraction and presents challenges for learners (Jin et al., 2024; Gonzalez et al., 2025). Learners must reason about perception, internal state, planning, coordination, and adaptation – capabilities that signal the emergence of a new, up-to-date form of computational thinking (CT): not only sequencing, loops, and conditionals, as well as problem decomposition, abstraction, and generalization, but also thinking in terms of goals, interactions, dependencies, feedback, and emergence, which represent higher levels of abstraction.

We suggest that when agentic ideas are grounded in familiar, playful contexts, such as guiding a sprite through a maze, coordinating sprites that communicate, or designing a simulation that reacts to player decisions, as done in Scratch programming, these invisible ideas become visible. Students can experience what it means for a program to “think,” “decide,” “reflect” or “respond,” rather than merely

executing instructions. Through such experiences, learners develop the contemporary CT required to conceptualize the foundational elements of agentic programming.

Although full agentic programming cannot be implemented within Scratch, we propose using Scratch as a conceptual bridge (Ejersbo et al., 2014). By engaging learners in the development of Scratch projects, educators can deliberately foreground ideas, such as autonomy, goals, perception, communication and adaptation. In this way, learners begin to encounter the core concepts of agentic programming in an accessible, manipulable environment that scaffolds their understanding of abstract notions. In a follow up work, we propose a Cognition Palette whose basic building blocks simulate key processes of agentic software development, providing the conceptual foundation for what we envision as AgentScratch, a block-based development environment for agentic programming (Hazzan and Ragonis, 2026).

## 2. Agentic Programming: Concepts and Challenges

Agentic programming represents a paradigm shift in contemporary software development, in which programs are constructed around autonomous or semi-autonomous agents rather than solely around predefined algorithmic procedures. In this approach, LLM-driven agents do more than generate code: they interpret high-level goals, decompose tasks, plan multi-step actions, interact with external tools (such as search engines or debuggers), incorporate feedback, and iteratively refine their outputs (Yao et al., 2022; Schick et al., 2023; Wang et al., 2025; Chowa et al., 2026). Consequently, programming becomes an orchestrated cycle of reasoning and action, in which decision-making and adaptation are integral to the computational process. Recent ACM and HCI literature characterize this shift as a move away from direct instruction toward collaborative human–AI problem-solving, where control is distributed between the programmer and the agent (Hoda, 2025; Schömbis et al., 2025; Wadinambiarachchi et al., 2025). Rather than writing complete algorithms, developers increasingly focus on specifying intentions, constraints, and evaluation criteria, while agents accept responsibility for generating, adapting, and evaluating code artifacts throughout the development process (Jin et al., 2024, Shah et al., 2025).

From a computer science *education* perspective, agentic programming introduces substantial challenges because it fundamentally reshapes what it means to “write a program.” When learners interact with agent-like entities, programming activity shifts from constructing explicit algorithms to formulating goals, interpreting feedback, and supervising autonomous behavior (Jin et al., 2024, Gonzalez et al., 2025). Empirical studies indicate that students may produce functional solutions with the assistance of agents while holding only a partial understanding of the underlying logic and control structures (Shah et al., 2025; Wermelinger, 2023; Kazemitabaar et al., 2023). This creates instructional challenges not primarily around misuse but around how learners conceptualize core agentic ideas, such as autonomy, decision-making, responsibility, and control. Therefore, educators are required to rethink learning objectives and assessment practices to distinguish between what the human learner understands and performs independently and what is delegated to the agent. More broadly, teaching with agentic systems demands new pedagogical approaches that support productive human–agent collaboration, including helping students learn how to specify intentions, evaluate agent behavior, verify outcomes, and reflect on the division of labor between themselves and the agent (Jin et al., 2024, Vaithilingam et al., 2022). These challenges highlight the need for educational models that make the underlying concepts of agentic programming explicit and cognitively accessible.

Agentic programming also introduces multiple layers of abstraction simultaneously, including goal specification, autonomous decision-making, hidden reasoning processes and tool-mediated execution. Unlike traditional programming, where causality can often be traced directly through explicit code paths, the actual execution of programming in agentic systems is distributed across interactions between agent reasoning, prompts, external tools, and iterative feedback loops (Yao et al., 2022; Schick et al., 2023). As a result, learners may struggle to understand why a system behaves as it does because outcomes often emerge from complex, partially opaque processes rather than from transparent, line-by-line logic. ACM and CHI studies report that this opacity of causality presents a significant cognitive challenge, particularly for novice programmers, as essential reasoning and decision-making processes occur behind the scenes, reducing traceability and explainability (Jin et al., 2024, Vaithilingam et al.,

2022). Broader reviews of LLM-based agents similarly emphasize that educational applications must address issues of transparency, overreliance, and effective integration into learning environments, precisely because agents' intermediate reasoning and evidentiary bases are not always visible to learners (Wang et al., 2025; Chowa et al., 2026). Together, these challenges complicate learners' ability to form accurate mental models of agentic behavior, underscoring the need for pedagogical approaches that externalize agent reasoning and make core agentic concepts more accessible to learners.

From the CT perspective, agentic programming foregrounds an expanded and up-to-date set of practices that extend well beyond traditionally agreed-upon skills. In agentic contexts, learners are required to engage more deeply in goal specification, decompose into subgoals, plan and monitor multi-step processes, evaluate generated solutions, iteratively refine outcomes, and coordinate actions, often in collaboration with autonomous agents (Jin et al., 2024; Schömb's et al., 2025). These competencies reflect the growing prominence of human-agent collaboration in real-world programming practices (Hoda, 2025; Wadinambiarachchi et al., 2025). Together, these developments signal the need to reconceptualize CT in ways that make agency, autonomy, and intentional behavior explicit and teachable in the classroom. In this paper, we discuss CT dimensions of agentic programming that can be introduced and scaffolded through accessible programming environments, such as Scratch.

### **3. Scratch as a Learning Environment for Simulating Contemporary Agentic Programming**

Scratch is a block-based visual programming environment developed by the MIT Media Lab with the explicit goal of enabling learners to create interactive stories, games, and simulations while lowering the syntactic barriers to programming (Resnick et al., 2009; Maloney et al., 2010). Its design emphasizes snap-together blocks, immediate visual feedback, and an event-driven execution model, allowing learners to focus on computational ideas, behaviors, and interactions rather than syntax. Central to Scratch's philosophy is the notion of programming for all, where creative expression, experimentation, and sharing are core components of learning (Resnick et al., 2009).

A substantial body of research has demonstrated that Scratch supports the development of foundational CT concepts and practices, including decomposition, abstraction, and debugging, through the design and refinement of meaningful computational artifacts (Brennan & Resnick, 2012). Systematic reviews of Scratch-based learning research further indicate consistent positive effects on learners' CT development across K–9 contexts, with particular strengths in supporting iterative problem-solving and conceptual understanding (Hazzan et al., 2020; Zhang & Nouri, 2019; Talan, 2020; Ragonis et al., 2022A; Ragonis et al., 2022B; Stewart & Baek, 2023). While Scratch is most commonly associated with children aged 8–16 years, empirical studies show that its pedagogical value extends beyond young learners. In teacher education, Scratch has been shown to positively influence attitudes toward programming, self-efficacy, and conceptual views of computing (Yükseltürk, 2017).

In recent years, Scratch has increasingly been explored as an entry point for introducing ideas from AI, machine learning, and data-oriented reasoning (Kazemitabaar et al., 2023). Studies have described extensions that connect Scratch to external services, enabling learners to experiment with data streams, pattern recognition, and algorithmic decision-making (Wald et al., 2019).

Importantly, Scratch's core constructs – sprites with internal states, sensing blocks, message passing, and iterative control structures – naturally lend themselves to modelling aspects of agentic behavior. Although Scratch does not support full agentic programming in the sense of LLM-driven autonomous agents, its design enables learners to conceptualize sprites as agents – entities that perceive, decide, act, and interact with their environment. *This design positions Scratch as a promising pedagogical environment for introducing the essential ideas of agency, autonomy, and feedback that underlie agentic programming in a concrete and developmentally appropriate manner.*

There are several environments that share the pedagogical spirit of Scratch, including Snap! (which extends Scratch with more advanced programming constructs), MIT App Inventor (which emphasizes event-driven, real-world application development), Cognimates (which integrates Scratch-like blocks with capabilities such as speech recognition, image recognition, and robotics), and Tynker (which focuses on game-based learning). The approach presented in this paper is conceptually aligned with

these environments in its use of block-based programming as a medium for developing CT. However, Scratch is selected as the focal environment due to its widespread global adoption and its well-established role in supporting the learning of programming foundations across all age groups, including higher education.

#### 4. Mapping Agentic Programming into Scratch

Following our pedagogical suggestion, in this section we describe the conceptual bridge between agentic programming and Scratch by mapping core agentic programming ideas onto concrete Scratch constructs. As discussed in the previous sections, while Scratch cannot support full LLM-driven agentic systems, it can be used as an environment for conceptualizing agentic behavior. Therefore, the mapping presented here is pedagogical rather than technical. The idea is to ground agentic programming concepts in familiar Scratch programming practices, enabling learners to reason about agency, autonomy, and interaction through hands-on design. As stated above, in parallel, we plan to develop AgentScratch to supports LLM-driven agentic behavior.

The proposed mapping is first presented by identifying core agentic concepts and their corresponding Scratch constructs and subsequently, by analyzing the accomplishment of agentic conceptions by their Scratch analogs.

##### 4.1. Core Agentic Concepts and Their Scratch Analogues

Table 1 summarizes the conceptual mapping underling how Scratch can naturally support agentic ideas through its design primitives. This reflects how agentic programming commonly characterizes systems as composed of agents who pursue goals, perceive their environment, make decisions, act, communicate, and adapt over time.

| Agentic Programming Concept  | Scratch Representation                                                                                                    |
|------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| Goals                        | The target of the entire simulation                                                                                       |
| Agent                        | Sprite with its own scripts and local variables                                                                           |
| Environment                  | Stage, stage scripts, other sprites, global variables                                                                     |
| Perception                   | Sensing blocks (e.g., touching?, distance, key pressed?)                                                                  |
| Internal state / memory      | Sprite-specific variables and lists, clones                                                                               |
| Decision making / planning   | Control structures (if, if-else, repeat, forever), time                                                                   |
| Action                       | Motion, looks, sound, pen                                                                                                 |
| Communication                | Scratch actions such as broadcast / when I receive / when touch                                                           |
| Prioritizing competing goals | Rule ordering, conditional precedence, variable-based priorities                                                          |
| Reflection / monitoring      | Lists or variables used as logs or counters, inputs from the user gotten by the operation 'ask' and the variable 'answer' |
| Tool use                     | Custom blocks (procedures/functions)                                                                                      |

*Table 1 – Conceptual mapping between agentic programming concepts and Scratch representation*

This mapping makes it explicit that Scratch embodies many of the structural elements required to reason about agentic systems, even in the absence of AI capabilities. Hence, explicit instruction that addresses these analogies can promote a meaningful understanding of agentic programming and scaffold the abstract concepts that agentic programming introduces.

##### 4.2. Accomplishing Agentic Conceptions and Their Scratch Analogues

This section describes the three main elements of agentic programming and their corresponding implementation in Scratch.

- a) *Modeling Sense–Think–Act Cycles in Scratch:* A central idea in agentic programming is the continuous sense–think–act cycle. In Scratch, this cycle can be explicitly modeled using forever loops that repeatedly:
  - sense the environment (using sensing blocks);
  - evaluate internal state and conditions;
  - select and execute actions.

By designing sprites around such sense-think-act loops, learners shift from writing linear scripts to designing autonomous behavior. **Such loops do not precompute the sequence of behaviors**; rather, behaviors emerge **dynamically from** interactions among the system state or the environment, perception, and control logic. This reflects a fundamental property of agentic systems and supports learners' understanding of autonomy and adaptation.

b) *Single-Agent and Multi-Agent Designs*: Scratch supports both single-agent and multi-agent perspectives. In single-agent designs, a sprite autonomously navigates an environment (e.g., a maze or simulation). In multi-agent designs, multiple sprites operate concurrently, each with its own goals and decision logic, while coordinating through message passing, or global variables values. The multi-agent Scratch scripts provide a powerful analogy to agentic programming, since learners can reason about:

- parallel execution;
- coordination and negotiation;
- shared versus local state;
- emergent system's behavior.

These properties of Scratch foreground interaction and dependency – key concepts in contemporary agentic systems, as they underpin the autonomy of such systems.

c) *Reflection, Explainability, and Metacognition*: As agentic-oriented designs emphasize reflection on decision making, Scratch learners can be encouraged to log decisions, record reasons for actions, or visualize internal state changes. Such practices externalize otherwise invisible processes and support explainability, metacognition, and debugging. Here are several examples of such externalization activities:

- display state changes visually (e.g., speech bubble, costume change, color effect),
- externalizes decision making,
- create a list of variables and present their changing values,
- observe project behavior, change one rule, re-run, and compare decision logs.

Pedagogically, this aligns with the need to make agent reasoning visible, which was identified earlier as a major abstraction barrier in learning agentic programming (Rombaut et al., 2024).

## 5. Illustrating the Mapping with a Concrete Scratch Simulation

In this section, we describe a simulation that models the motion vectors of two billiard balls before, during, and after a collision, until both balls come to a complete stop, while accounting for friction and table boundaries. This demonstration is inspired by a project developed by a student during his training to become a physics teacher, as part of a course on CT. The project simulated a billiards game as a computational process of collisions between bodies in physics, taking into account the objects involved, the forces acting on them, and the angle of impact.

Figure 1 presents a screenshot of the simulation. Figure 2 shows part of the Scratch blocks that set up the screen and simulate the forever loop that governs the simulation. Table 2 outlines the representation of agentic programming concepts in the simulation. As can be seen, even a relatively simple simulation can be described using agentic terminology.

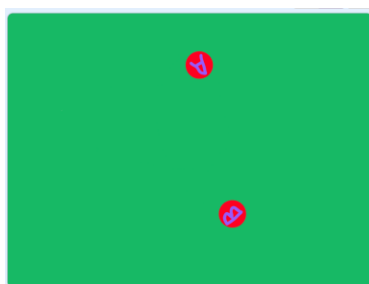


Figure 1 - A simulation of two billiard balls during a collision

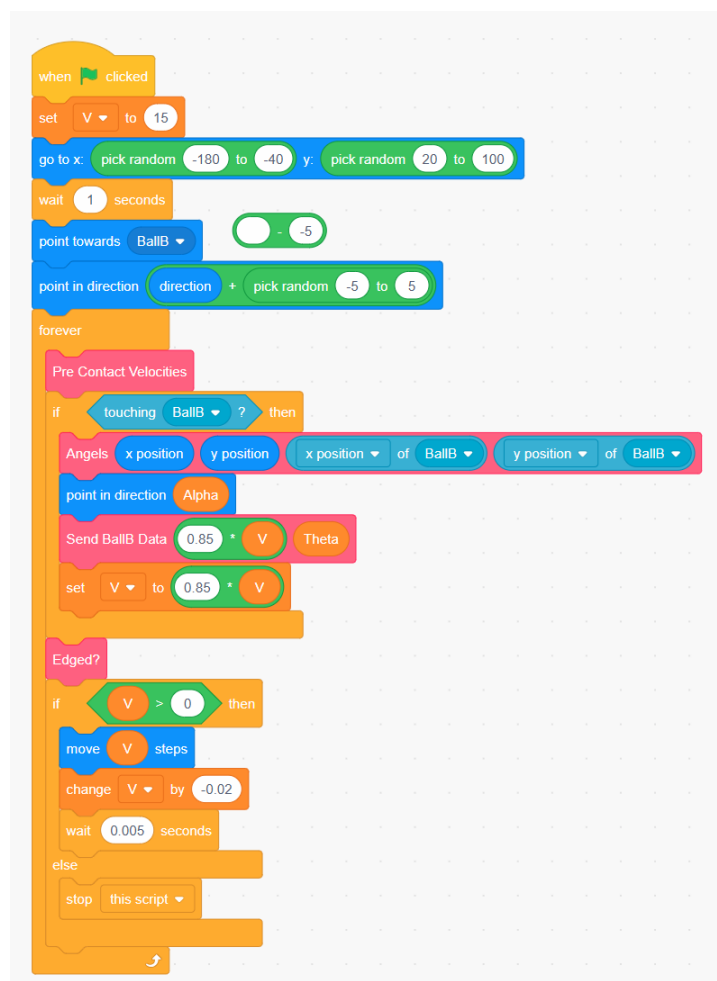


Figure 2 - Part of the Scratch blocks that creates the simulation of the two balls<sup>1</sup>

## 6. Computational Thinking Shifts and Pedagogical Implications

To examine the shift in CT skills required for agentic programming, we refer to the analytical components of agentic programming in Scratch, as presented in Section 4. Table 3 summarizes these analytical components and lists the fundamental CT and the agentic CT skills relevant to their implementation.

By foundational CT skills we refer to more traditional algorithm-centered CT skills that are now frequently established in programming curricula. They are typically emphasized in introductory programming and early CT frameworks, where the focus is placed on single-program, rule-based, and locally traceable logic.

By agentic CT skills, we refer to CT skills involved in designing, reasoning about, and coordinating autonomous agents and their interactions with environments and other agents. These agentic CT skills emphasize intentionality, interaction, monitoring, coordination, and reflection. They encompass CT skills that are often underrepresented or implicit in traditional curricula but become salient and necessary in agentic contexts.

We note that the distinction presented in Table 3 does not represent a hierarchy of difficulty, but rather a shift in CT orientation prompted by agentic programming.

<sup>1</sup> Created by Ali Zur (with his permission).

| Agentic Programming Concept  | Representation in the Simulation of the Collision                                                                                                                                                                                                            |
|------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Goals                        | The simulation models the motion vectors of two billiard balls, before, during, and after collision while accounting for friction and table boundaries until both balls come to a complete stop.                                                             |
| Agent                        | The simulation include two objects: ball A is in motion, with a nonzero velocity, whereas ball B is stationary prior to the collision.                                                                                                                       |
| Environment                  | The environments includes: A felt-covered billiard table; ball A with initial velocity and direction prior to the collision and ball B is stationary beforehand; a variety of variables                                                                      |
| Perception                   | Ball A accounts for friction with the felt-covered table, detects and handles collisions with ball B, and manages interactions with the table boundaries. The code uses <i>touching?</i> and variables values.                                               |
| Internal state / memory      | Ball A has 11 variables, for example: gV2, gTheta, and Right Attack.<br>Ball B has four variables, for example: gV2 and gTheta.                                                                                                                              |
| Decision making / planning   | Ball A includes a <i>loop</i> that runs while its velocity remains <i>greater than zero</i> and controls the simulation of the two ball's motion. This is a <i>forever</i> loop using <i>if</i> and <i>if-else</i> structures.                               |
| Action                       | The two balls move on the table according to the scripts.                                                                                                                                                                                                    |
| Communication                | Ball B responds to the collision message from A by receiving its updated angle and velocity. The communication use <i>broadcast</i> .                                                                                                                        |
| Prioritizing competing goals | Figure 1 presents a part of the rule ordering.<br>The simulation implements several basic behaviors but does not account for others, such as the spin of the ball, which affects both the friction with the felt and the motion vectors after the collision. |
| Reflection / monitoring      | This simulation does not include inputs; inputs could be added, for example, for friction.                                                                                                                                                                   |
| Tool use                     | The simulation includes a variety of custome blocks such: <i>Edged?</i> , <i>Send Ball B Data(V,Theta)</i>                                                                                                                                                   |

Table 2 – Representation of agentic programming concepts in the simulation of the collision.

| Agentic Programming Conceptions               | Computational Thinking Skills                                                                                                                           |                                                                                                                                                                                                                                          |
|-----------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                               | Foundational Computational Thinking Skills                                                                                                              | Agentic Computational Thinking Skills                                                                                                                                                                                                    |
| Core Agentic Concepts                         | Skills needed to identify agents, environments, goals, and states, and to abstract these concepts into programmable representations                     |                                                                                                                                                                                                                                          |
| Modeling Sense–Think–Act Cycles               | Skills needed to design continuous control scripts that integrate perception, decision logic, and action, highlighting adaptive and dynamic computation |                                                                                                                                                                                                                                          |
|                                               | <ul style="list-style-type: none"> <li>Algorithmic thinking beyond linear sequencing</li> <li>Conditional reasoning and decision-making</li> </ul>      | <ul style="list-style-type: none"> <li>Iterative process design</li> <li>Monitoring system state and feedback</li> <li>Temporal reasoning (reasoning over time and repeated cycles)</li> </ul>                                           |
| Single-Agent and Multi-Agent Designs          | Skills for developing system-level thinking, requiring learners to reason about distributed control, inter-agent dependencies, and collective outcomes  |                                                                                                                                                                                                                                          |
|                                               | <ul style="list-style-type: none"> <li>Decomposition into interacting components</li> <li>Coordination and communication</li> </ul>                     | <ul style="list-style-type: none"> <li>Parallelism and concurrency</li> <li>Reasoning about interactions and emergent behavior</li> <li>Responsibility allocation and delegation</li> </ul>                                              |
| Reflection, Explainability, and Metacognition | Skills for making computational processes explicit, supporting learners' ability to explain, justify, and revise agent behavior.                        |                                                                                                                                                                                                                                          |
|                                               | <ul style="list-style-type: none"> <li>Evaluation and debugging</li> <li>Iterative refinement and improvement</li> </ul>                                | <ul style="list-style-type: none"> <li>Tracing and explaining computational behavior</li> <li>Reflective reasoning about design choices</li> <li>Justification of decisions</li> <li>Awareness of limitations and uncertainty</li> </ul> |

Table 3 – Computational thinking skills: From traditional to agentic programming

The analysis presented in Table 3 does not imply that Scratch itself is an agent-native environment or that its existing computational vocabulary fully captures the concepts required for agentic programming. More generally, we propose that a programming environment designed for a different programming paradigm may not fully represent the concepts and computational processes underlying agentic programming. Therefore, a potential compatibility gap may exist between Scratch's current

programming abstractions and the conceptual requirements of agentic programming. This concern motivates our proposal for AgentScratch, an agent-oriented, block-based programming environment inspired by Scratch. Rather than simply repurposing Scratch for agentic programming, AgentScratch is intended to retain the accessibility and hands-on nature of block-based programming while introducing constructs that explicitly represent agentic concepts and processes. Specifically, AgentScratch introduces learners to LLM-based agents while constraining their design and behavior through a predefined set of visual, conceptual blocks. In this way, learners can explore agentic concepts without being overwhelmed by the technical complexity of directly programming LLM-based agents (Hazzan and Ragonis, 2026).

## 7. Conclusion and Summary

This study presents a pedagogical mapping that identifies structural and behavioral analogies between agentic programming and Scratch, enabling learners to experience agentic thinking within a familiar block-based programming environment. Rather than attempting to replicate agentic programming architectures, the proposed approach focuses on concretizing abstract agentic concepts, such as autonomy, goals, decision-making, interaction, and adaptation, through accessible Scratch design practices.

As mentioned above, though Scratch provides a powerful medium for conceptually modeling agentic concepts, it cannot be considered an agentic programming environment in the technical sense. Agentic programming involves systems in which agents can autonomously interpret goals, plan multi-step actions, dynamically select and use external tools, and iteratively refine their behavior based on feedback, often supported by large-scale models and access to external resources. In contrast, Scratch operates within a closed, deterministic, and predefined execution model: all behaviors must be explicitly programmed in advance, decision-making is limited to rule-based logic, and there is no intrinsic support for autonomous goal interpretation, dynamic tool invocation, or learning from data (unless such capabilities are explicitly defined and programmed). Hence, while Scratch enables the representation of core agentic programming concepts as presented in this paper, it is limited to scripted simulations of agency rather than authentic autonomous processes.

Yet, from a pedagogical perspective, we view Scratch as a pedagogical foundation whose familiar block-based interaction can be extended with constructs explicitly designed to represent agentic processes. Our vision of AgentScratch intend to address the compatibility gap, identified in Section 6, while retaining the accessibility and hands-on nature of block-based programming, it introduces building blocks that make agent-specific concepts (such as goals, states, perception, planning, interaction, and autonomous action) explicit and manipulable. In this way, the proposed environment does not merely repurpose a tool designed for another programming paradigm; it adapts the pedagogical affordances of Scratch to the conceptual requirements of agentic programming, providing learners with a direct environment in which to construct, observe, and reason about agentic computational processes.

The rapid emergence of GenAI is fundamentally reshaping the programming landscape, including the growing prominence of agentic systems. Although the future trajectory of these technologies remains uncertain, it is already clear that the nature of CT required in this context is changing significantly. Programming increasingly demands reasoning about intentional behavior, coordination, evaluation, and responsibility, in addition to traditional algorithmic constructs (Wing in Hoffmann, 2026).

The analysis in this paper highlights that responding to these challenges does not require abandoning well-established educational tools or prematurely introducing opaque technologies. Instead, it demonstrates how a dynamic and widely used environment, such as Scratch, can function as a powerful pedagogical mediator. By grounding potentially intimidating and abstract ideas in concrete, comprehensible programming experiences, the proposed approach supports the development of up-to-date CT skills while preserving learners' sense of agency, understanding, and control over their learning.

However, it is important to clarify the limitations of this mapping. Scratch cannot natively host AI agents, access large-scale models, or perform autonomous tool selections. Therefore, its role is not to implement agentic programming but to serve as a conceptual preparation environment and, where relevant, as an execution or visualization layer for externally generated agentic designs. This distinction

reinforces the central claim of this study: Scratch provides a *didactic medium* through which the essential ideas of agentic programming can be introduced, explored, and understood. Thus, the work presented here represents a pedagogical step toward supporting learners and educators in navigating the evolving realities of programming in the age of GenAI. In Ragonis et al. (2025) one such pedagogy is presented. Extending these insights, we envision AgentScratch, a block-based programming environment designed to support learners in developing agentic thinking and programming skills by making key agentic concepts and computational processes explicit and manipulable (Hazzan and Ragonis, 2026).

## References

- Brennan, K., & Resnick, M. (2012). Using artifact-based interviews to study the development of computational thinking in interactive media design. *Proceedings of the American Educational Research Association (AERA)*.
- Chowa, S. S., Alvi, R., Rahman, S. S., Rahman, M. A., Raiaan, M. A. K., Islam, M. R., ... & Azam, S. (2026). From language to action: a review of large language models as autonomous agents and tool users. *Artificial Intelligence Review*.
- Ejersbo, L. R., Leron, U., & Arcavi, A. (2014). Bridging intuitive and analytical thinking: Four looks at the 2-glass puzzle. *For the Learning of Mathematics*, 34(3), 2-7.
- Gonzalez, E., Chan, J., & Weintrop, D. (2025). Quack! Configuring large language models to serve as rubber duck coding assistants. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 2 (SIGCSE TS 2025)*. Association for Computing Machinery. <https://doi.org/10.1145/3641555.3705235>
- Hazzan, O., Ragonis, N., & Lapidot, T. (2020). Computational thinking. In *Guide to Teaching Computer Science: An Activity-Based Approach* (pp. 57-74). Cham: Springer International Publishing.
- Hazzan, O. & Ragonis, N. (2026). The Cognition Palette: Teaching Programming in the Age of AI Agents Using Block-Based Environments, *BLOG@CACM, Communications of the ACM*. <https://cacm.acm.org/category/artificial-intelligence-machine-learning/>
- Hoda, R. (2025). Toward Agentic Software Engineering Beyond Code: Framing Vision, Values, and Vocabulary. *arXiv preprint arXiv:2510.19692*.
- Hoffmann, L. (2026). The rise of computational thinking in the age of AI, Opinion - Communications of the ACM, January 2026.
- Jin, H., Lee, S., Shin, H., & Kim, J. (2024, May). Teach ai how to code: Using large language models as teachable agents for programming education. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (pp. 1-28).
- Kazemitabaar, M., Chow, J., Ma, C. K. T., Ericson, B. J., Weintrop, D., & Grossman, T. (2023, April). Studying the effect of AI code generators on supporting novice learners in introductory programming. In *Proceedings of the 2023 CHI conference on human factors in computing systems* (pp. 1-23).
- Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch programming language and environment. *ACM Transactions on Computing Education*, 10(4), Article 16.
- Ragonis, N., Bukai, A., & Hazzan, O. (2022A). Selecting examples for CS courses: the case of a computational thinking MOOC. *ACM Inroads*, 13(3), 22-28.
- Ragonis, N., & Hazzan, O. (2022B, December). A MOOC on Computational Thinking for All: Pedagogical Principles, Challenges, and Their Application. In *2022 International Conference on Computational Science and Computational Intelligence (CSCI)* (pp. 1943-1949). IEEE.
- Ragonis, N., Rosenberg-Kima, R. B., & Hazzan, O. (2025). A computational thinking course for all preservice K-12 teachers: implementing the four pedagogies for developing computational

- thinking (4P4CT) framework. *Educational technology research and development*, 73(1), 301-329.
- Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., & Kafai, Y. (2009). Scratch: Programming for all. *Communications of the ACM*, 52(11), 60–67.
- Rombaut, B., Masoumzadeh, S., Vasilevski, K., Lin, D., & Hassan, A. E. (2024). *Watson: A Cognitive Observability Framework for the Reasoning of Foundation Model-Powered Agents*. <https://arxiv.org/abs/2411.03455>
- Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Hambro, E., ... & Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36, 68539-68551.
- Schömb, S., Zhang, Y., Goncalves, J., & Johal, W. (2025). From conversation to orchestration: HCI challenges and opportunities in interactive multi-agentic systems. In *Proceedings of the 13th International Conference on Human-Agent Interaction (HAI '25)* (pp. 1–11). Association for Computing Machinery.
- Shah, A., Chernova, A., Tomson, E., Porter, L., Griswold, W. G., & Soosai Raj, A. G. (2025, February). Students' Use of GitHub Copilot for Working with Large Code Bases. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1* (pp. 1050-1056).
- Stewart, W. H., & Baek, Y. (2023). Analyzing computational thinking studies in Scratch programming: A review of elementary education literature. *International Journal of Computer Science Education in Schools*, 6(1), 35–58.
- Talan, T. (2020). Investigation of the studies on the use of Scratch software in education: A systematic review. *Journal of Education and Future*, 18, 109–134.
- Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022, April). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts* (pp. 1-7).
- Wald, I., Zuckerman, O., et al. (2019). Teaching children machine learning through gesture recognition in Scratch. *Proceedings of the Interaction Design and Children Conference*.
- Wang, H., Gong, J., Zhang, H., Xu, J., & Wang, Z. (2025). *AI agentic programming: A survey of techniques, challenges, and opportunities*. University of Leeds. *arXiv preprint arXiv:2508.11126*
- Wermelinger, M. (2023, March). Using github copilot to solve simple programming problems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (pp. 172-178).
- Wadinambiarachchi, S., Waycott, J., Rogers, Y., & Wadley, G. (2025). Imagining design workflows in agentic AI futures. In *Proceedings of the 37th Australian Conference on Human-Computer Interaction (OZCHI '25)* (pp. 1–17). Association for Computing Machinery.
- Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., & Cao, Y. (2022, October). React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.
- Yükseltürk, E. (2017). An investigation of the effects of programming with Scratch on the views of preservice information technology teachers. *British Journal of Educational Technology*, 48(2), 537–550.
- Zhang, L., & Nouri, J. (2019). A systematic review of learning computational thinking through Scratch in K–9. *Computers & Education*, 141, 103607.