World Journal of Educational Research
ISSN 2375-9771 (Print) ISSN 2333-5998 (Online)
Vol. 13, No. 3, 2026
www.scholink.org/ojs/index.php/wjer

Original Paper

A Code-Agency-Centered Evidence Framework for Programming Education in the Age of AIGC

Wei Xin[1*] & Qinxin Liao[2]

[1] School of Information Engineering, Gannan University of Science and Technology, Ganzhou, Jiangxi 341000, China, 18846916579@163.com

[2] College of Chemistry, Beijing Normal University, Beijing, 100875, China, lqx20140101@163.com

[*] Corresponding Author

Received: May 26, 2026 Accepted: June 19, 2026 Online Published: July 7, 2026

doi:10.22158/wjer.v13n3p148 URL: http://dx.doi.org/10.22158/wjer.v13n3p148

Abstract

Generative artificial intelligence (AIGC) has made executable code easier to produce, but it has also weakened the evidential link between submitted code and students' actual programming competence. In programming education, the visibility of code products now contrasts with the relative invisibility of competence formation: students may submit code that runs without being able to explain its logic, judge its risks, revise its defects, or take responsibility for its consequences. This paper develops a Code-Agency-centered evidence framework for AIGC-supported programming education. Code Agency is deliberately bounded as educational ownership over code, expressed through four capacities: understanding code logic, judging code quality and risk, modifying code under constraints, and assuming responsibility for code behavior. The framework organizes learning through three progressive stages--independent implementation, AI-mediated critique and revision, and responsible engineering delivery--while treating AI access as a governance condition coupled with evidence requirements. The evidence cycle comprises execution-model construction, debugging and verification, AI output review and revision, and on-site defense with feedback adjustment. A LiteOS task-management case is used as a diagnostic setting because RTOS runtime states are partly invisible, scheduling behavior is temporally complex, and AI-generated code may appear correct while remaining poorly understood. The framework contributes a bounded and assessable account of code ownership for programming education in the age of AIGC.

Keywords

AIGC, generative AI, programming education, Code Agency, evidence of code ownership, assessment design, LiteOS

1. Introduction

Generative AI has quickly moved from a specialized technology into everyday programming practice. In computing and engineering courses, students can now request functions, debugging explanations, refactoring suggestions, test cases, or complete solutions and receive plausible code within seconds. This shift creates a problem deeper than academic dishonesty or tool management. It changes the evidential status of student-submitted code. When executable code can be externally generated, teachers can no longer assume that a working artifact transparently reveals the learner's understanding, judgment, revision ability, or responsibility.

The central challenge is therefore evidential rather than merely disciplinary. In traditional programming assessment, the final code product often supported an artifact-to-competence inference: if students submitted working code, teachers could tentatively infer that they had interpreted the problem, constructed logic, reasoned about execution, debugged errors, and revised the implementation. AIGC destabilizes that inference. The code product becomes increasingly visible, while competence formation becomes less directly observable. Students may submit runnable and well-structured code without having formed the execution model, evaluative judgment, or maintenance responsibility that the artifact appears to signal.

This paper argues that programming education in the age of AIGC requires a shift from code-product assessment to evidence of code ownership. Code ownership does not require students to write every line without assistance. It means that students retain educational agency over the artifact they submit: they can understand how the code works, judge whether it is adequate and safe, modify it under new constraints, and account for its behavior and consequences. These requirements are captured here by the construct of Code Agency.

The paper develops a Code-Agency-centered evidence framework. It makes three contributions. First, it bounds Code Agency as a four-part construct--understanding, judgment, modification, and responsibility--so that it does not become a catch-all term for programming competence. Second, it reframes AI regulation as an evidence-design problem: the more AI support a task permits, the stronger the required evidence of student ownership should be. Third, it uses LiteOS task management as a diagnostic case in which invisible runtime states, scheduling complexity, and plausible but fragile AI-generated code place productive pressure on students' claims of code ownership.

2. Literature Review and Research Gap

2.1 AIGC and Programming Education

Studies of generative AI in computing education show a dual pattern. On one side, AI tools can provide scalable help, explain errors, generate examples, and support students who may not receive timely instructor feedback (Kazemitabaar et al., 2024; Liffiton et al., 2023). Classroom deployments such as CodeAid and CodeHelp show that guardrails can be designed to provide support without simply handing students final answers. On the other side, code generation tools can solve many introductory

tasks and can therefore undermine the use of conventional assignments as evidence of individual learning (Becker et al., 2023; Finnie-Ansley et al., 2022).

Recent empirical research confirms generative AI has uneven effects on new programmers.Groothuijsen's 2024 study noted engineering students used ChatGPT for debugging, code explanation and solution writing, while instructors observed poorer coding ability and weaker knowledge mastery. Prather et al. (2024) found some beginners accelerated learning with AI, yet others faced worsening metacognitive struggles. Zviel Girshin (2024) similarly warned of overreliance and shallow understanding of core programming basics among novice learners.Collectively, these works show the key educational challenge lies not in AI availability, but in designing proper teaching approaches to let AI support rather than replace students' independent learning.

2.2 Code Understanding and Execution Models

True programming proficiency involves far more than writing grammatically correct code. Learners have to grasp control logic, variable changes, memory allocation, function calls, data storage formats and runtime exceptions. Research by Fincher et al. (2020) and Sorva (2013) on notional machines argues that students need a clear abstract model of program execution. With this mental model, they can predict outputs, trace through code line by line, fix bugs and explain how code works. Without such a framework, students often only notice superficial code patterns or see a small set of test cases run smoothly, falsely believing they have fully mastered the relevant knowledge.

AIGC intensifies this issue because it can supply not only code but also fluent explanations of code. When generated code appears clean, students may read it as authoritative even when it contains requirement mismatches, missing boundary cases, inefficient logic, unsafe assumptions, or misleading comments. The problem is not that AI explanations are always wrong. It is that students may borrow the appearance of understanding without acquiring the execution model that would let them test, question, or repair the code. In programming education, execution-model construction is therefore not an optional preliminary exercise; it is part of the evidential basis for judging whether submitted code is educationally owned by the student.

2.3 AI-Generated Code, Human Judgment, and Revision

AI literacy research provides a useful but broad starting point for thinking about student interaction with AI systems (Almatrafi et al., 2024; Long & Magerko, 2020). Broader educational research has also emphasized that large language models create both learning opportunities and risks that require careful pedagogical design (Kasneci et al., 2023). However, programming education requires a more specific focus. Students must not merely know that AI systems can be biased, inaccurate, or limited; they must be able to evaluate concrete AI-generated code, identify defects, revise implementations, and verify behavior. This focus aligns with recent arguments that generative AI requires students to develop evaluative judgment rather than passive acceptance of machine output (Bearman et al., 2024).

Assessment research in the GenAI era similarly points toward redesign rather than prohibition. The AI Assessment Scale, for example, treats AI use as something that can be structured across levels of

permitted support and aligned with human input and critical thinking (Furze et al., 2024). In computing education, recent risk-management work likewise suggests that generative AI should be handled through explicit educational strategies rather than through blanket rejection (Humble, 2024). In programming education, however, a generic AI-permission scale is not enough. The assessment question must be tied to code-specific evidence: Can students explain execution, diagnose faults, evaluate AI output, revise code under constraints, and defend the resulting artifact?

2.4 Research Gap: From Code Artifacts to Evidence of Code Agency

The literature has identified the capabilities and risks of AI code generation, the importance of notional machines, and the need for AI literacy and assessment redesign. Yet the deeper issue remains under-theorized: AIGC changes the validity of a familiar assessment inference. Existing studies often discuss AI use, code understanding, academic integrity, and assessment as separate concerns. Less attention has been given to an integrated account of evidence validity under AI-mediated code production: if code can be externally generated, what forms of evidence justify the claim that students understand, judge, revise, and take responsibility for it?

This paper addresses that gap by proposing Code Agency as a bounded construct and by developing an evidence framework for making student code ownership observable. The framework is conceptual rather than empirical. It does not claim that a particular intervention has improved learning outcomes. Instead, it clarifies the evidence logic that programming tasks and assessments need when AI-mediated code production weakens the traditional link between artifact and competence.

3. Code Agency as a Bounded Construct

A central risk in proposing a new educational construct is conceptual overextension. If Code Agency is used as a broad synonym for programming competence, AI literacy, debugging ability, academic integrity, and engineering responsibility all at once, it loses explanatory value. This paper therefore defines Code Agency narrowly as educational ownership over code in AI-mediated programming contexts. It is expressed through four observable capacities: understanding code logic, judging code quality and risk, modifying code under constraints, and assuming responsibility for code behavior.

These capacities are developmental in logic, although not always linear in classroom practice. Understanding means that students can enter the logic of the code: they can explain purpose, structure, state changes, and execution flow. Judgment means that they can evaluate whether the code is reliable, appropriate, and aligned with requirements rather than accepting either human-written or AI-generated code at face value. Modification means that they can revise the code under new constraints and explain the consequences of those revisions. Responsibility means that they can defend the submitted artifact, disclose relevant AI support, and accept responsibility for maintenance and failure. Code Agency therefore names a compact relation between learner and code: understanding, judgment, modification, and responsibility.

Table 1. Bounded dimensions of Code Agency

Dimension	Agency meaning	Typical evidence
Understanding	The student can explain the purpose, structure, state changes, and execution flow of the code.	Execution-model diagram, trace table, code walkthrough
Judgment	The student can evaluate whether code is reliable, appropriate, and aligned with requirements.	Defect analysis, test rationale, requirement-code comparison
Modification	The student can revise code under new constraints and explain the consequences of the change.	Revised implementation, change log, local modification task
Responsibility	The student can defend, disclose support for, maintain, and take responsibility for submitted code.	AI-use disclosure, oral defense, maintenance explanation

This well-defined framework delineates the applicable boundaries of Code Agency. It does not mean a total ban on students' use of AI tools. Students can still show their independent control over coding even with AI support.The concept also differs from AI literacy. In this paper, the ability to assess and adjust AI-generated code serves as a learning method to build Code Agency, instead of standing as an independent core concept on its own.Furthermore, it covers more than just academic honesty. The key concern is not merely whether students abide by rules, but whether their submitted codes can reliably reflect their actual learning progress. Lastly, Code Agency cannot represent the full scope of engineering competency. When AI takes part in code writing and creates confusion over work ownership, student comprehension and assessment credibility, this concept offers specific criteria to evaluate students' learning performance through code.

4. The Code-Agency-Centered Evidence Framework

4.1 Design Problem

The framework begins with a central design problem: when AI systems can generate plausible or executable code, programming instruction must generate additional evidence that students educationally own the code they submit. Ownership neither means legal copyright nor requires students to write code entirely on their own. Instead, it describes students' active mastery of their coding work: they ought to evaluate whether the code meets requirements,comprehend the program logic, adjust the code within given limits, and bear full accountability for how the program operates.

For this reason, the framework treats assessment validity as a core concern. If coursework permits AI assistance but grading is based only on final code outputs, students will copy pre-written code rather than actually learn. A complete prohibition of AI, with no foundational coding exercises to back it up,

isolates classroom learning from actual software development practice. Rather than allowing or banning AI tools, one possible solution would be to redesign programming assignments and focus on process-based evaluation. Although it is reasonable to allow students to use AI, instructors must have tangible evidence that indicates the understanding, critical judgment and work in revising code by students during their learning process.

4.2 Three Design Tensions

Three instructional tensions arise from this design problem. The first tension is to keep productive struggle intact without making AI use inherently illegitimate. Novice programmers still require tasks that involve reading, tracing, predicting and building core logic, especially when the instructional objective is construction of program execution models. Nonetheless, an outright prohibition of AI fails to acknowledge its extensive application in real-world programming practice. This framework clearly divides coding learning situations into two groups. The first is self-directed exploration, which is essential to learning including following program logic and error debugging. The second represents AI-assisted heuristic learning wherein students ask conceptual advice to AI after having tried on their own.

The second tension is that AI tools should be adopted and the critical judgment of the students should not be lost. Students can use AI-generated code as a benchmark to compare against their own work, identify gaps in their solutions, experiment with other forms of coding and adjust the outcome based on it. This approach holds that the educational value of AI does not stem from its use alone. The only thing that matters is whether students can still exercise critical judgment over AI-generated code.

The third tension lies in evaluating code artifacts without ignoring the processes that give them meaning. Code artifacts must be interpreted in relation to requirements, constraints, tests, maintainability, resource use, and potential failures. This tension is especially salient in engineering and embedded systems courses, where code may appear locally correct while violating timing, concurrency, or resource assumptions. This approach therefore extends assessment beyond visible program outputs and connects student work to the real-world consequences of code execution.

4.3 AI Access-Evidence Coupling

This framework is underpinned by the core regulatory principle of AI access–evidence coupling, which embeds a full evidentiary chain for AI usage throughout all instructional tasks. Within this framework, the greater the AI assistance privileges granted to students, the more robustly they must demonstrate deep mastery and conceptual understanding of the knowledge underpinning their learning outcomes.

This principle shifts AI governance away from an all-or-nothing approach and centers on the collection of learning evidence. The use of AI is educationally viable only if assessment designs can effectively evaluate students' code comprehension, independent judgment, revision work, and sense of accountability. In this way, AI access is inseparably tied to the supporting evidence students submit to demonstrate their Code Agency.

Table 2. AI Access-Evidence Coupling principle

AI access condition	Typical task type	Required evidence emphasis
No AI or minimal AI	Execution tracing, quizzes, hand-written core logic	Understanding evidence: trace tables, state predictions, code explanation
Limited AI explanation or hints	Concept review, debugging discussion, pseudocode planning	Understanding and judgment evidence: comparison of student reasoning and AI suggestion
AI-assisted critique or partial generation	Code review, project extension, refactoring	Judgment and modification evidence: defect report, revised code, test log, prompt record
Disclosed AI collaboration	Integrated project or engineering task	Full ownership evidence: process record, verification evidence, defense, maintenance rationale

4.4 Four-Stage Evidence Cycle

The framework operationalizes the generation of evidence in four interrelated steps: (1) execution-model construction, (2) debugging and verification, (3) AI output review and revision, and (4) on-site defense with feedback adjustment. These stages form an iterative cycle, as evidence generated at each stage serves as the basis for work in the next stage.

The predicted execution model built in the first stage guides what students observe during program execution. Discrepancies between predictions and observed outcomes define the scope of debugging and verification work in the second stage. Verified diagnostic findings from the debugging stage form the basis for reviewing and revising AI-generated code in the third stage. Finally, the defense stage tests whether students can integrate the complete evidence trail into an accountable, coherent explanation of their work.

In the opening stage of execution model construction, students must predict program behavior before turning to AI-generated explanations. Students may produce state diagrams, trace runtime sequences, or articulate expected control flow. These artifacts establish the baseline against which all subsequent execution evidence is interpreted.

Debugging and verification follow, centering on a comparison between predicted and observed program behavior. Where actual program execution diverges from the proposed model, students must account for discrepancies through log analysis, targeted testing, and causal reasoning. This work moves learners from conceptual understanding to evaluative judgment, as they assess whether runtime behavior confirms or challenges their initial model.

AI output review and revision place AI-generated code under critical scrutiny. Students identify requirement mismatches, unaddressed boundary cases, unsubstantiated assumptions, and misleading explanatory content, and then revise the code and verify the impacts of their changes. Critical judgment

is here translated into concrete, tested modifications.

The final stage, on-site defense and feedback adjustment, requires students to explain their decision-making process, respond to incremental revision prompts, and justify their testing or revision work with controlled access to AI. Accountability is made explicit in this phase: students must connect the final deliverable to the full evidence trail that produced it. Evidence collected during defense also feeds forward into instruction, informing task difficulty, AI access parameters, scaffolding design, and rubric criteria for the next cycle.

Taken together, these four stages are not intended as a linear checklist, but as an iterative feedback system for progressively fostering students' code agency.

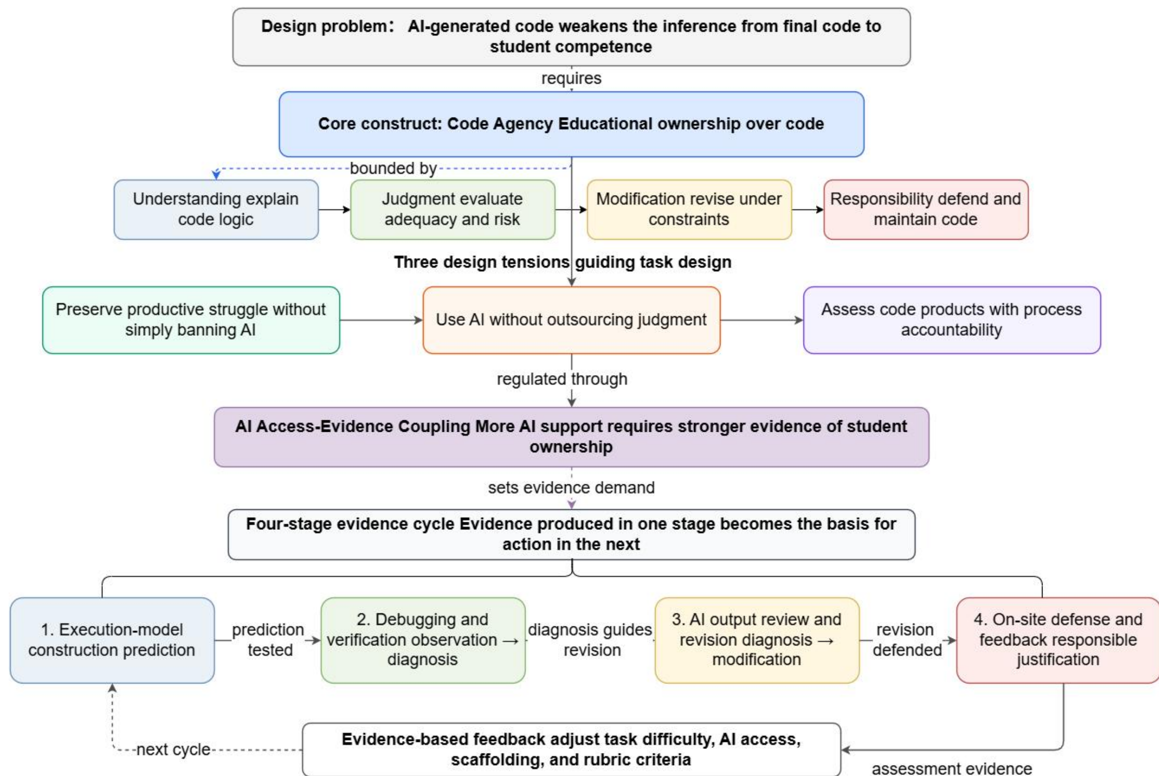

Figure 1. Code Agency Evidence Framework

4.5 Framework Mapping

Table 3 summarizes how the major components of the framework contribute to making Code Agency observable. Together, these components connect the theoretical construct, instructional tensions, AI governance principle, evidence cycle, and feedback mechanism into an integrated assessment design.

Table 3. Mapping the framework to evidence of Code Agency

Framework part	Function	Evidence produced
Code Agency	Defines bounded educational	Evidence of understanding, judgment,

	ownership over code	modification, and responsibility
Three design tensions	Guide task design around the tensions introduced by AI-mediated code production	Evidence that productive struggle is preserved, AI is evaluated, and engineering accountability is addressed
AI Access-Evidence Coupling	Regulates the relation between AI support and evidence requirements	AI-use record, prompt record, process evidence, verification evidence
Four-stage evidence cycle	Makes code ownership observable	Trace table, state diagram, test log, defect report, revised code, defense response
Feedback adjustment	Uses evidence to revise the next instructional cycle	Updated task difficulty, AI access level, rubric criteria, and support strategy

5. LiteOS Task Management as a Diagnostic Case

5.1 Why LiteOS Is Diagnostic

LiteOS task management is used here not as proof of an empirical intervention, but as a diagnostic pressure test for Code Agency. RTOS task behavior is partly invisible to students. A task may be created, delayed, blocked, awakened, preempted, or terminated according to scheduling rules that are not directly visible in the source code. Students may see serial output but still misunderstand the state transitions that produced it. This makes LiteOS useful not merely as an application domain, but because it exposes the gap between a code artifact and runtime understanding.

The case is also conceptually demanding. Priority-based scheduling, delay intervals, shared-state access, task synchronization, resource use, and error checking interact in ways that can make a program appear correct in a limited run while remaining fragile or misleading. An AI-generated LiteOS example may include plausible API calls and comments but still contain inappropriate priorities, missing return-value checks, unsafe shared-state assumptions, or incorrect explanations of scheduling order. These features make LiteOS more diagnostic than a simple function exercise because final code alone cannot establish whether students understand, judge, modify, or take responsibility for the runtime system.

5.2 Task Design

Course assignments should not merely require students to develop functional multitask programs. Instead, they must prompt students to produce full-process learning evidence that demonstrates their comprehensive thinking in understanding, evaluating, modifying and interpreting code. Typical practical tasks cover task creation and deletion, priority scheduling, delay and block processing, synchronization and mutual exclusion, message transmission, resource allocation, and runtime monitoring.

The practical learning process consists of four structured steps. First, students predict task state transitions according to given priority levels and delay durations, and explicitly document their presumed program execution logic. Second, students develop and run basic multitask programs,

observe serial output results, and compare the actual operating performance with their prior predictions. Third, they identify inconsistencies between predicted and practical outcomes, optimize the code, and verify program performance through operation logs, timing records and boundary tests. Fourth, students analyze defective LiteOS sample codes generated by AI, and complete on-site code modification or academic defense.

Table 4. LiteOS task design aligned with the evidence cycle

Evidence-cycle stage	LiteOS learning task	Code Agency dimension
Execution-model construction	Predict ready, running, blocked, delayed, and terminated task states under priority and delay conditions.	Understanding
Debugging and verification	Diagnose missing output, wrong scheduling order, incorrect delay behavior, or unsafe shared-state access.	Understanding; Judgment
AI output review and revision	Review AI-generated LiteOS code containing hidden scheduling, synchronization, or resource-management defects.	Judgment; Modification
On-site defense and feedback adjustment	Modify priority, task period, synchronization rule, or alarm condition and explain the result.	Modification; Responsibility

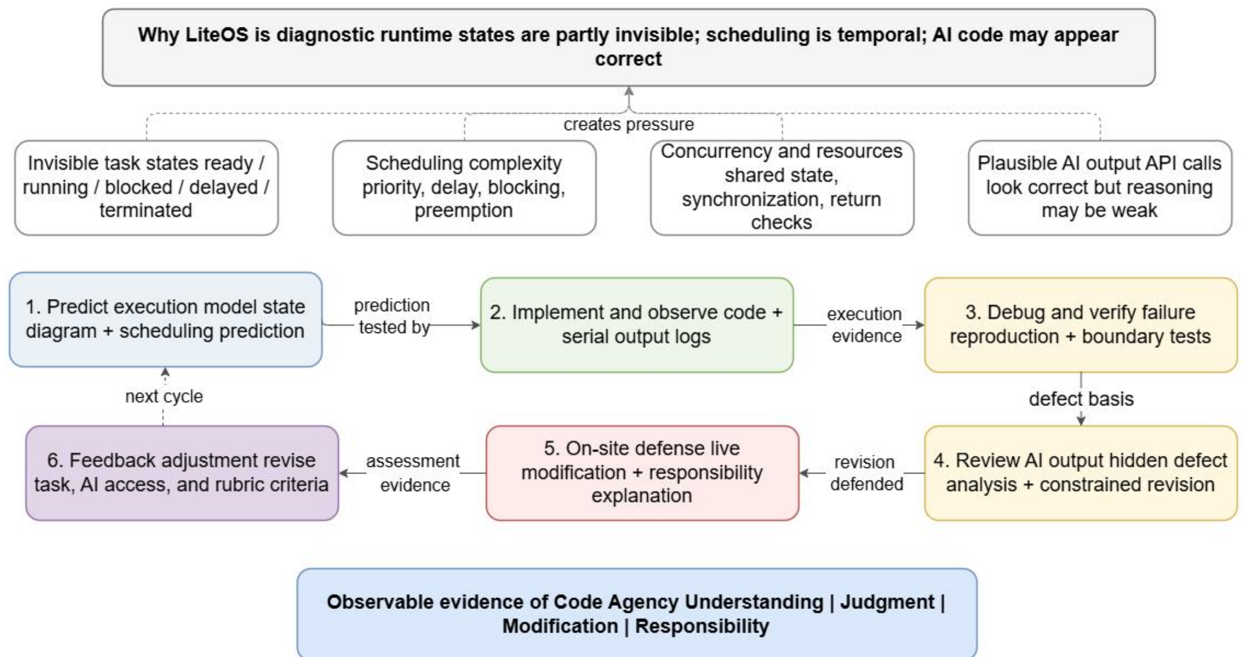

Figure 2. LiteOS Diagnostic Evidence Cycle

5.3 Evidence Requirements

The evidence package should extend beyond the final code artifact. To support a defensible claim of Code Agency, students should submit a requirement interpretation, a task-state diagram, a scheduling prediction table, serial-output logs, a debugging report, boundary-test results, an AI-output review report when AI is used, a revised implementation, and a brief defense statement explaining how these materials support the final program. If AI assistance is permitted, the prompt record and the student's evaluation of AI-generated output should also be included as part of the assessment evidence.

These requirements are not intended to increase workload indiscriminately. Their purpose is to make otherwise invisible learning processes visible and to protect the validity of assessment claims. A trace table can show whether students can reason about task states; serial-output logs can show whether they can connect execution evidence to scheduling claims; boundary tests can show whether they can evaluate reliability beyond a successful demonstration; an AI review report can show whether they can identify plausible but defective code; and an on-site modification can show whether they can maintain the code under new constraints. Each evidence type therefore compensates for a limitation of final-code assessment and helps clarify what the submitted artifact can, and cannot, prove about student learning.

5.4 Assessment Logic

The assessment logic follows directly from these evidence requirements. Final code should receive only part of the grade because technical correctness is not equivalent to educational ownership. A technically correct program is insufficient if the student cannot explain why a scheduling order occurs, identify defects in an AI-generated example, or modify the program under a small new requirement. In such cases, the code may function as an artifact, but it does not provide sufficient evidence of Code Agency.

Conversely, a partially flawed program may still offer meaningful evidence of learning if the student can diagnose the fault, propose a defensible repair, and explain the verification process. The assessment target is therefore not code perfection alone, but the validity of the student's claim to Code Agency. In this logic, code correctness, process evidence, AI-use disclosure, revision quality, and defense performance are interpreted together to determine whether students understand, judge, modify, and take responsibility for the code they submit.

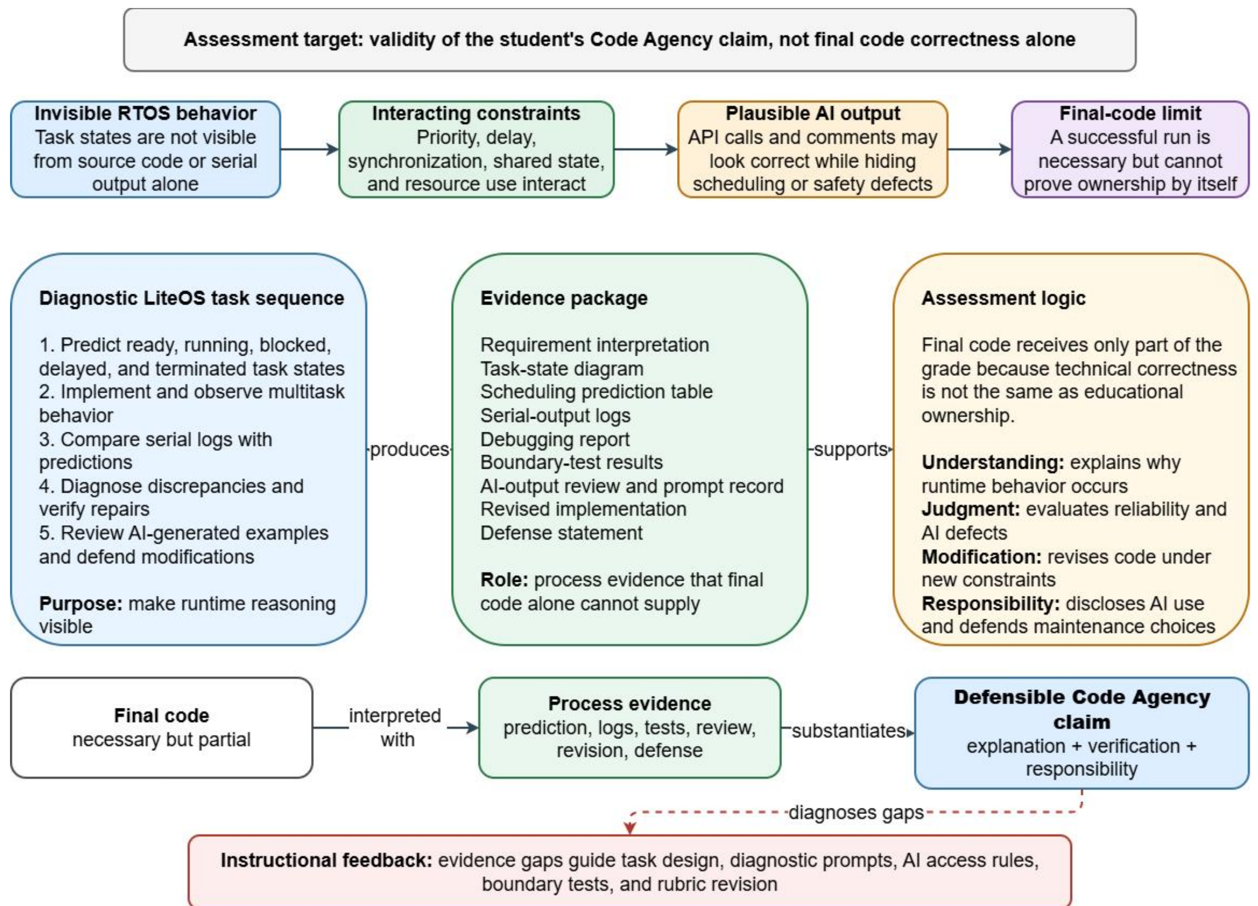

Figure 3. LiteOS Evidence Assessment Map

6. Discussion

6.1 Theoretical Contribution

The theoretical contribution of the paper is to reframe programming education in the age of AIGC as a problem of evidence validity. The framework shifts attention from whether code artifacts are correct to whether they can still support defensible inferences about student competence. This produces a first conceptual move: from artifact validity to evidence validity. A second move is from AI permission to evidence design. Rather than asking only whether AI should be allowed, the framework asks what evidence must accompany different forms of AI support. A third move is from broad competence language to bounded code ownership. By defining Code Agency through understanding, judgment, modification, and responsibility, the framework gives educators a more precise vocabulary for assessing learning when production, explanation, and repair can be partially externalized to AI.

6.2 Instructional Contribution

The instructional contribution is the AI Access-Evidence Coupling principle and the four-stage evidence cycle. Together they support flexible AI integration without abandoning assessment rigor. Teachers can differentiate tasks: some tasks restrict AI to protect execution-model construction, while others permit AI but require stronger process, verification, revision, and defense evidence. This logic

also makes feedback more actionable, because evidence from logs, reviews, tests, and defenses can be used to adjust task difficulty, AI access, and rubric criteria in the next instructional cycle.

6.3 Domain Contribution

The domain contribution lies in showing why embedded RTOS education is a particularly strong context for testing Code Agency. In such courses, runtime behavior is not always directly visible, and correctness depends on scheduling, timing, synchronization, and resource assumptions. These characteristics make it possible for AI-generated code to appear plausible while leaving essential reasoning unexamined. The LiteOS case demonstrates how a domain with invisible runtime behavior can place diagnostic pressure on students' understanding, judgment, modification, and responsibility.

7. Limitations and Future Research

This paper is a conceptual framework study supported by a diagnostic course case. It does not provide experimental evidence that the framework improves student learning outcomes. The LiteOS case illustrates assessment and instructional logic rather than reporting evaluated intervention data. Future research should test the framework through classroom experiments, quasi-experimental studies, or design-based research cycles, with particular attention to whether evidence requirements improve the validity of judgments about student code ownership.

Further work is also needed to develop and validate Code Agency rubrics. Such rubrics should assess understanding, judgment, modification, and responsibility without collapsing them into a single broad score. A four-level developmental scale--for example, foundational, intermediate, engineering, and innovative performance--could be explored, but only if each level is anchored in observable evidence rather than general claims of ability. Learning analytics may help collect evidence such as prompt logs, version histories, test results, and debugging traces, but these tools should support rather than replace instructor judgment. Cross-course studies could examine whether the framework applies beyond embedded RTOS education to introductory programming, software engineering, cybersecurity, and AI application development.

8. Conclusion

AIGC does not diminish the importance of programming education; it changes the evidence by which programming learning is judged. When AI can generate plausible and executable code, the final program alone can no longer serve as sufficient evidence of student competence. The key educational question therefore shifts from whether students can produce code to whether they can demonstrate ownership of the code they submit.

This paper proposed a Code-Agency-centered evidence framework to address this shift. Code Agency is defined as educational ownership over code, expressed through understanding, judgment, modification, and responsibility. The framework reframes AI governance as an evidence-design problem: the more AI support a task permits, the stronger the required evidence of student ownership

should be.

The LiteOS case illustrates why this framework is necessary. In embedded RTOS learning, runtime behavior is complex, partly invisible, and easily obscured by superficially correct AI-generated code. By linking final code to execution-model construction, debugging and verification, AI-output review and revision, and on-site defense, the framework provides a basis for reconnecting AI-mediated code production with valid claims about learning. Its contribution is to shift programming assessment from judging code artifacts alone to interpreting code through the evidence trail that makes Code Agency observable.

References

Almatrafi, O., Johri, A., & Lee, H. (2024). A systematic review of AI literacy conceptualization, constructs, and implementation and assessment efforts (2019-2023). *Computers and Education Open*, *6*, Article 100173.

Bearman, M., Tai, J., Dawson, P., Boud, D., & Ajjawi, R. (2024). Developing evaluative judgement for a time of generative artificial intelligence. *Assessment & Evaluation in Higher Education*, *49*(6), 893-905.

Becker, B. A., Denny, P., Finnie-Ansley, J., Luxton-Reilly, A., Prather, J., & Santos, E. A. (2023). 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* (Vol. 1, pp. 500-506). Association for Computing Machinery.

Denny, P., Prather, J., Becker, B. A., Finnie-Ansley, J., Hellas, A., Leinonen, J., Luxton-Reilly, A., Reeves, B. N., Santos, E. A., & Sarsa, S. (2024). Computing education in the era of generative AI. *Communications of the ACM*, *67*(2), 56-67.

Fincher, S., Jeuring, J., Miller, C. S., Donaldson, P., du Boulay, B., Hauswirth, M., Hellas, A., Hermans, F., Lewis, C., Muhling, A., Pearce, J. L., & Petersen, A. (2020). Notional machines in computing education: The education of attention. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (pp. 21-50). Association for Computing Machinery.

Finnie-Ansley, J., Denny, P., Becker, B. A., Luxton-Reilly, A., & Prather, J. (2022). 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). Association for Computing Machinery.

Furze, L., Perkins, M., Roe, J., & MacVaugh, J. (2024). The AI Assessment Scale (AIAS) in action: A pilot implementation of GenAI-supported assessment. *Australasian Journal of Educational Technology*, *40*(4), 38-55.

Groothuijsen, S., van den Beemt, A., Remmers, J. C., & van Meeuwen, L. W. (2024). AI chatbots in programming education: Students' use in a scientific computing course and consequences for

learning. *Computers and Education: Artificial Intelligence*, *7*, Article 100290.

Humble, N. (2024). Risk management strategy for generative AI in computing education: How to handle the strengths, weaknesses, opportunities, and threats? *International Journal of Educational Technology in Higher Education*, *21*, Article 61.

Kasneci, E., Sessler, K., Kuchemann, S., Bannert, M., Dementieva, D., Fischer, F., Gasser, U., Groh, G., Gunnemann, S., Hüllermeier, E., Krusche, S., Kutyniok, G., Michaeli, T., Nerdel, C., Pfeffer, J., Poquet, O., Sailer, M., Schmidt, A., Seidel, T., ... Kasneci, G. (2023). ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and Individual Differences*, *103*, Article 102274.

Kazemitabaar, M., Ye, R., Wang, X., Henley, A. Z., Denny, P., Craig, M., & Grossman, T. (2024). CodeAid: Evaluating a classroom deployment of an LLM-based programming assistant that balances student and educator needs. In *Proceedings of the CHI Conference on Human Factors in Computing Systems* (Article 650, pp. 1-20). Association for Computing Machinery.

Liffiton, M. H., Sheese, B. E., Savelka, J., & Denny, P. (2023). CodeHelp: Using large language models with guardrails for scalable support in programming classes. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research* (Article 8). Association for Computing Machinery.

Long, D., & Magerko, B. (2020). What is AI literacy? Competencies and design considerations. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (pp. 1-16). Association for Computing Machinery.

Prather, J., Denny, P., Leinonen, J., Becker, B. A., Albluwi, I., Craig, M., Keuning, H., Kiesler, N., Kohn, T., Luxton-Reilly, A., MacNeil, S., Petersen, A., Pettit, R., Reeves, B. N., & Savelka, J. (2023). The robots are here: Navigating the generative AI revolution in computing education. In *Proceedings of the 2023 Working Group Reports on Innovation and Technology in Computer Science Education* (pp. 108-159). Association for Computing Machinery.

Prather, J., Reeves, B., Leinonen, J., MacNeil, S., Randrianasolo, A. S., Becker, B. A., Kimmel, B., Wright, J., & Briggs, B. (2024). The widening gap: The benefits and harms of generative AI for novice programmers. In *Proceedings of the 2024 ACM Conference on International Computing Education Research* (Vol. 1, pp. 469-486). Association for Computing Machinery.

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education*, *13*(2), Article 8.

Zviel Girshin, R. (2024). The good and bad of AI tools in novice programming education. *Education Sciences*, *14*(10), Article 1089.