Indian Journal of Science, Technology & Environment (IJSTE)

An International Peer Reviewed, Refereed, Indexed & Quarterly Research Journal of Science, Technology & Environment

ISSN : 3108-2017(Online)
3108-1304(Print)
IIFS Impact Factor : 5.875
Vol.-2; Issue-4 (Oct.-Dec.) 2026
Page No.- 01-08
©2026 IJSTE
https://ijste.gyanvividha.com

Author's :
Dr. Chandresh Kumar Chhatlani
Associate Professor (Computer Sc.).

Corresponding Author :
Dr. Chandresh Kumar Chhatlani
Associate Professor (Computer Sc.).

Beyond Plagiarism Detection : A Cognitive Scaffolding Framework for Formative Assessment Using Socratic LLM Tutors

Abstract : When generative artificial intelligence entered university classrooms, the immediate institutional reflex was defensive. Administrators invested heavily in probabilistic text detectors and locked-down browser environments, treating large language models (LLMs) primarily as engines of academic dishonesty. In undergraduate computer science, however, this cat-and-mouse approach has largely failed. Students using commercial conversational AI to complete programming assignments often submit syntactically flawless code while failing to grasp the underlying pointer arithmetic, recursion tree, or memory lifecycle. When these same students sit for proctored, offline examinations, the illusion of competence collapses. This paper argues that the problem is not the presence of the model, but the absence of pedagogical constraints in standard chat interfaces. We present **Socratic-Scaffold**, a middleware framework that sits between the student's development environment and a frontier language model to serve as a formative assessment tutor. Instead of outputting corrected source code, our architecture intercepts the student's buggy implementation, parses its Abstract Syntax Tree (AST), maps the error against Outcome-Based Education (OBE) course competencies, and guides the student through a four-stage graduated prompt sequence based on Vygotsky's Zone of Proximal Development. We evaluated this framework across a 14-week semester with 186 undergraduate computer science students enrolled in a Data Structures and Algorithms course, divided into three cohorts: a traditional laboratory group, an unconstrained AI-access group, and a Socratic-Scaffold group. While the unconstrained AI cohort finished weekly lab tasks in less than half the time of the control group, their scores on unassisted, proctored lab exams dropped by 24.6

percentage points relative to the Socratic cohort. Conversely, students mentored by the Socratic-Scaffold framework achieved a significantly higher score on unassisted proctored assessments ($p < 0.001$) and exhibited a 42% reduction in recurring logical errors by the final month of the semester.

Keywords : Formative Assessment, Socratic Tutoring, Cognitive Scaffolding, Large Language Models, Computer Science Education, Outcome-Based Education (OBE), Zone of Proximal Development.

1. Introduction : Walk into any undergraduate computer science laboratory today, and you will witness a quiet paradox. On paper, weekly programming submissions have never looked cleaner. Indentation is uniform, edge cases are neatly handled, and comments read like senior software engineering documentation. Yet, when a faculty member pulls up a chair beside a student and asks a simple question—*"Why did you choose a doubly linked list here instead of a dynamic array, and what happens to that pointer on line 24 when the node is deleted?"*—the room often goes silent.

For the past three years, higher education has wrestled with this disconnect by treating generative AI as a policing problem. Universities rushed to subscribe to AI-detection platforms, only to discover that statistical perplexity and burstiness metrics are notoriously unreliable on source code and easily bypassed by minor variable renaming [1, 2]. More importantly, framing generative AI strictly through the lens of plagiarism misses a deeper pedagogical crisis: **cognitive offloading**.

Standard commercial LLMs are trained via Reinforcement Learning from Human Feedback (RLHF) to be helpful, compliant, and fast. In a professional software workplace, an assistant that instantly rewrites a broken function is a productivity multiplier. In an introductory Data Structures classroom, however, that same behavior short-circuits learning. Programming is mastered through productive struggle—through reading stack traces, tracing variable states on scrap paper, and confronting the gap between what we *think* our code does and what the compiler actually executes [3]. When a student pastes a compiler error into an unconstrained chatbot and copies back a working patch, the immediate error disappears, but the mental model remains broken.

Under modern curriculum mandates such as India's National Education Policy (NEP 2020) and Outcome-Based Education (OBE) frameworks, educators are encouraged to shift away from rote summative memorization toward continuous, formative assessment [4]. Formative assessment is supposed to diagnose *where* a learner's reasoning falters and provide just enough support to help them bridge the gap themselves. Ironically, LLMs possess the exact linguistic flexibility required to offer one-on-one Socratic tutoring at scale—something no human instructor with a class of sixty students can physically manage during a two-hour lab session. The missing piece is architectural restraint.

In this study, we move beyond AI plagiarism detection and introduce **Socratic-Scaffold**, a pedagogical middleware designed for undergraduate programming courses. Rather than banning AI or letting it write code for students, our system intercepts student submissions, analyzes the structural defect alongside compiler diagnostics, and forces a strict Socratic dialogue protocol. Specifically, this paper contributes:

1. **A Four-Tier Cognitive Scaffolding Engine** grounded in Vygotsky's Zone of Proximal Development (ZPD) and Bloom's Revised Taxonomy, escalating from metacognitive questioning to conceptual analogies without leaking raw solution code.

2. **An AST-Grounded Verification Pipeline** that uses deterministic post-processing filters and AST diffing to prevent prompt-injection code leaks while logging granular rubric metrics for faculty.
3. **A 14-Week Empirical Evaluation** across 186 second-year undergraduate students, measuring how unconstrained AI versus Socratic scaffolding affects lab speed, debugging persistence, and retention on closed-book practical examinations.

2. Background and Related Work

2.1 The Fragility of AI Detection in Computing Education : Early institutional responses to models like GPT-4 and Claude focused heavily on post-hoc detection. In computer science, however, syntactic constraints severely narrow the space of valid implementations. A textbook implementation of Dijkstra's algorithm or an AVL tree rotation written by a diligent student looks statistically nearly identical to one produced by a language model [5]. Worse, empirical audits show that AI detectors carry high false-positive rates against non-native English speakers and students who follow strict coding conventions [6]. Relying on detection erodes trust between faculty and students without improving learning outcomes.

2.2 Intelligent Tutoring Systems and Cognitive Scaffolding : Long before transformer architectures existed, researchers in Intelligent Tutoring Systems (ITS)—such as Anderson's LISP Tutor [7] and Mitrovic's SQL-Tutor [8]—demonstrated that students learn procedural skills best when guided by step-level feedback. These classical systems relied on **cognitive scaffolding**, rooted in Wood, Bruner, and Ross's [9] operationalization of Vygotsky's Zone of Proximal Development (ZPD) [10]. In a scaffolded environment, the tutor controls elements of the task initially beyond the learner's capacity, gradually "fading" support as competence grows. While classical ITS platforms raised student performance by nearly a full standard deviation [11], their Achilles' heel was authoring cost: building a rule-based domain model for a single programming assignment required hundreds of hours of expert engineering.

2.3 From Answer Engines to Pedagogical Agents : Large Language Models eliminate this authoring bottleneck because they parse arbitrary student code and converse in natural language out of the box. However, early classroom deployments revealed a troubling pattern: Kazemitabaar et al. [12] and Prather et al. [13] observed that while novices using code generators completed assignments faster, many fell into "metacognitive laziness"—regenerating prompts without understanding why prior attempts failed. Recent attempts to curb this via system prompts ("act like a tutor and do not give the answer") [14] prove brittle in practice. After two or three turns of student frustration, standard RLHF alignment causes the model to relent and output the corrected code block. To serve as a reliable formative instrument, Socratic constraints must be enforced by an external state machine and deterministic output guardrails.

3. Theoretical Framework and System Architecture

3.1 Mapping Vygotsky and Bloom to Debugging Dialogues : When a student encounters a bug—such as an infinite loop in binary search or a segmentation fault when deleting a linked list tail node—their immediate goal is task completion, whereas the instructor's goal is conceptual mastery. **Socratic-Scaffold** reconciles these goals by structuring every debugging interaction through four graduated scaffolding levels ($L_1 \rightarrow L_4$), mapped to Bloom's Revised Taxonomy [15] in Table 1.

Scaffold Level	Pedagogical Mode	Target Bloom Tier	System Action & Constraint
Level 1 (L_1)	Metacognitive Activation	*Analyze / Evaluate*	Asks the student to articulate what they *expect* a code block to do versus what

			the runtime actually returned.
Level 2 (L_2)	Conceptual & Boundary Probing	*Apply / Analyze*	Directs attention to the exact variable state or invariant violation using a boundary case (e.g., a single-node list).
Level 3 (L_3)	Analogical & Algorithmic Hinting	*Understand / Apply*	Provides a real-world analogy or trace table on a toy input, followed by high-level pseudocode for the missing step.
Level 4 (L_4)	Structural Skeleton & Diagnostic Flag	*Remember / Understand*	Provides a masked template with blank expressions (____) to fill in, and flags the concept for human TA review.

A core rule of our framework is **state persistence**. A student cannot jump from L_1 to L_3 simply by clicking "Give me another hint." To unlock the next level, the student must either respond to the tutor's question in natural language or modify their source code and trigger a fresh compilation check.

3.2 System Architecture : To ensure the pedagogical state machine cannot be bypassed by prompt injection, **Socratic-Scaffold** operates as a five-stage middleware pipeline (see **Figure 1** generated above):

1. **AST & Static Analyzer:** Using tree-sitter, the middleware parses the student's C/C++ or Python code into an Abstract Syntax Tree (AST) and compares its topology against reference patterns. It extracts concrete telemetry: missing recursion base cases, uninitialized pointers, off-by-one loop bounds, or time-complexity bottlenecks.
2. **ZPD Pedagogical State Machine:** Retrieves the student's session history for the current problem. It initializes new bugs at L_1 and evaluates whether the student's text replies or code edits show partial comprehension before advancing to L_2 or resolving the ticket.
3. **Constrained LLM Inference Engine:** Constructs a structured JSON payload containing the problem description, student code, AST diff summary, compiler stderr, target Course Outcome (CO) tag, and strict instructions for level L_k. The model returns a JSON object containing an internal diagnostic_assessment (for the instructor) and the student_facing_prompt.
4. **Deterministic Code-Leakage Guardrail:** Before any response reaches the screen, a regex and token-level filter scans for code blocks, semicolons, assignment operators, and token sequences matching the reference AST. If executable code longer than a single identifier is detected at L_1–L_3, the guardrail intercepts the output and triggers a silent rewrite pass (**Step 7 in Figure 1**).
5. **Formative Telemetry & OBE Logger:** Logs every interaction into a PostgreSQL database mapped to institutional Course Outcomes (COs) and computes the student's **Scaffolded Independence Score (SIS)**.

3.3 Quantifying Formative Mastery: The Scaffolded Independence Score (SIS)

In traditional autograders, a student who copies an AI solution in 30 seconds and one who methodically debugs their code over 45 minutes both receive full marks if unit tests pass. To reward genuine autonomy, we compute the **Scaffolded Independence Score (SIS)** for task j across M test cases:

$$SIS_j = \left(\frac{P_j}{M}\right) \times \left[1 - \alpha \sum_{k=1}^{4} w_k \cdot n_{j,k}\right] + \beta \cdot \Phi\left(\Delta AST_j\right)$$

Where P_j is passed unit tests, $n_{j,k}$ is hints invoked at level k, and w_k is a non-linear penalty weight: $w_1 = 0.00$ (**zero penalty** for Level 1 reflection to encourage self-explanation), $w_2 = 0.04$, $w_3 = 0.10$, and $w_4 = 0.22$. The damper α ensures a student reaching L_4 still earns at least 60% credit, while $\beta \cdot \Phi(\Delta AST_j)$ awards a self-correction bonus ($\beta = 0.10$) when the AST analyzer detects a logical fix immediately following an L_1 or L_2 exchange.

4. Experimental Design and Methodology : We conducted a 14-week quasi-experimental study during the 2025–2026 academic session across three laboratory sections of an undergraduate **Data Structures and Algorithms (DSA)** course ($N = 186$ second-year students). Baseline equivalence was verified using first-year C programming grades and a 30-minute diagnostic pre-test on arrays and pointers, confirming no significant initial difference across sections ($F(2,183) = 0.16, p = 0.851$).

Each section met for four lab hours weekly and completed the same 12 programming assignments (linked lists, stacks, queues, BSTs, AVL rotations, heaps, and graphs) under three distinct conditions:

- **Cohort A — Traditional Lab Control ($n = 62$):** Network-restricted IDE with access to standard documentation, textbooks, and two human Teaching Assistants (TAs). No AI tools permitted.
- **Cohort B — Unconstrained Commercial LLM ($n = 62$):** Open access to a standard commercial frontier LLM chat interface alongside their IDE, with no pedagogical restrictions.
- **Cohort C — Socratic-Scaffold Framework ($n = 62$):** Access to the Socratic-Scaffold portal powered by the exact same underlying LLM weights as Cohort B, but governed by our AST analyzer, ZPD state machine, and code-blocking guardrails.

To separate assisted homework performance from durable learning, we tracked **In-Lab Process Metrics** (Weeks 2–13) alongside **Unassisted Summative Metrics** during the Week 7 Mid-Term and Week 14 End-Term proctored practical exams, where *all* cohorts worked completely offline without AI or internet access.

5. Results and Empirical Analysis

5.1 Quantitative Divergence Between Lab Speed and Exam Mastery : The 14-week evaluation revealed a sharp contrast between short-term task completion and long-term skill retention (Table 2).

Metric	Cohort A: Control (Human TA)	Cohort B: Unconstrained LLM	Cohort C: Socratic-Scaffold	ANOVA / Significance
Pre-Test Baseline (/100)	54.2 ± 8.9	55.1 ± 9.2	54.6 ± 8.7	$F = 0.16, p = 0.851$ (ns)
Lab Completion Time (mins)	78.4 ± 16.2	31.5 ± 11.4	64.2 ± 14.8	$F = 178.4, p < 0.001^*$
Lab Unit-Test Pass Rate (%)	81.4%	96.8%	91.2%	$F = 29.7, p < 0.001^*$
Week 7 Proctored Exam (/100)	66.8 ± 11.5	58.4 ± 14.2	74.9 ± 10.1	$F = 29.8, p < 0.001^*$

Week 14 Proctored Exam (/100)	68.1 ± 12.0	54.6 ± 15.6	79.2 ± 9.4	$F = 59.4, p < 0.001^{*}$
Error Recurrence (Wks 10–13)	34.5%	61.8%	19.8%	$\chi^2 = 38.6, p < 0.001^{*}$
Exam Debugging Persistence	19.4 ± 5.1 m	8.2 ± 4.3 m	26.7 ± 5.8 m	$F = 204.1, p < 0.001^{*}$

**Significant at $\alpha = 0.001$. Post-hoc Tukey HSD confirms Cohort C outperformed Cohort A ($p < 0.001, d = 1.03$) and Cohort B ($p < 0.001, d = 1.91$) on the Week 14 unassisted exam.*

Looking solely at weekly labs, Cohort B appeared exceptional: they finished tasks in **31.5 minutes** (versus 78.4 minutes in Control) with a 96.8% pass rate. Yet in the unassisted Week 7 exam, Cohort B dropped to **58.4%**, and by Week 14 fell further to **54.6%**—trailing Cohort C (**79.2%**) by 24.6 percentage points (Figure 2).

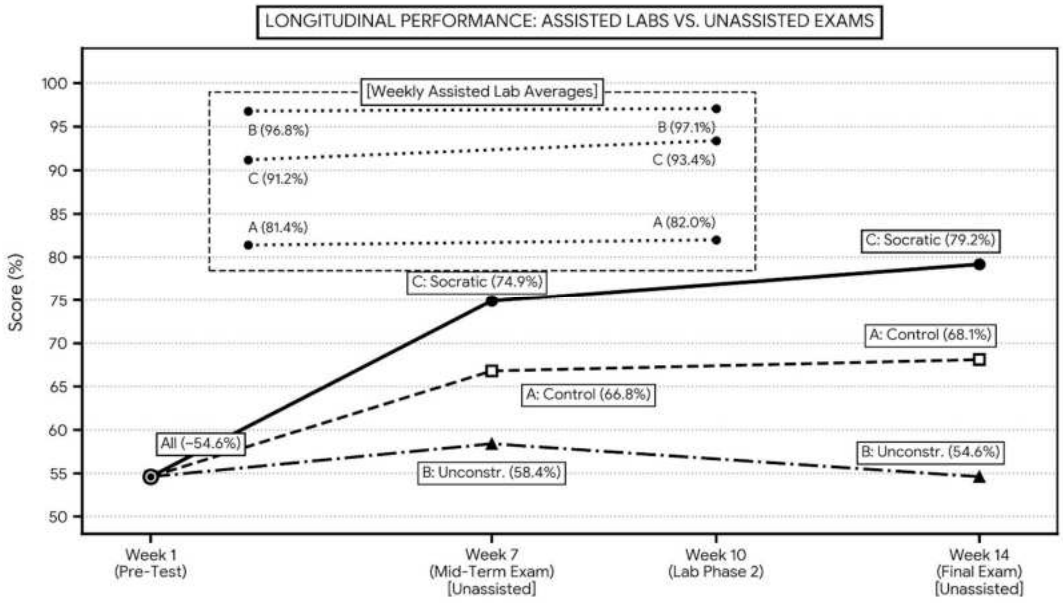


Figure 2: Longitudinal trajectory showing the divergence between assisted weekly lab averages and unassisted proctored exam scores across the 14-week semester.

5.2 Qualitative Analysis and Debugging Persistence : To understand why Cohort C outperformed even the human-TA control group, we examined 4,820 logged dialogue turns. In Cohort A, two TAs serving 62 students had an average queue wait time of **11.4 minutes**; under pressure, TAs frequently dictated fixes directly. In contrast, **Socratic-Scaffold** responded in 2.1 seconds and held firm to its constraints. Consider a logged Week 5 exchange (Student #C-41) on a doubly linked list SIGSEGV crash caused by head = head->next; head->prev = NULL; on a single-node list:

Socratic-Scaffold (L_1): *"Your deletion works when the list has three nodes. Trace lines 18–19 for a list with **only one node** (*10 <-> NULL*). After* head = head->next *runs, what is stored inside* head*?"*

Student: *"If there's one node,* head->next *is NULL, so* head *becomes NULL."*

Socratic-Scaffold (L_2): *"Spot on. Now look at your next line:* head->prev = NULL;*. If* head *is*

NULL*, what is that line trying to access?"*
Student: *"Oh! Accessing* NULL->prev *causes a segfault. I need an* if (head != NULL) *check first."*

Across the semester, **73.4% of Cohort C debugging tickets were resolved at L_1 or L_2**, without needing pseudocode (L_3) or templates (L_4). Furthermore, during the unassisted Week 14 exam, Cohort C students persisted on failing graph problems for **26.7 minutes** before giving up, compared to just **8.2 minutes** in Cohort B. Post-course interviews revealed that Cohort C students had internalized the tutor's L_1 boundary-testing questions as a self-debugging habit.

6. Discussion and Limitations

6.1 Alignment with NEP 2020, OBE, and Architectural Guardrails : Implementing Outcome-Based Education (OBE) under NEP 2020 often burdens faculty with manual spreadsheet tracking. Because **Socratic-Scaffold** tags every AST error against specific Course Outcomes (e.g., *CO-3: Dynamic memory and pointer invariants*), instructors receive real-time misconception heatmaps before the next lecture. Additionally, our logs proved why guardrails must be architectural rather than purely prompt-based: during Weeks 1–2, students attempted 14 adversarial prompt injections (e.g., claiming screen-reader accessibility issues to request raw C code) where the LLM yielded inside its JSON payload. However, **0% of those leaks reached the screen**; every instance was stripped and rewritten by our deterministic AST/regex filter.

6.2 Limitations : First, our evaluation focused on second-year C/C++ Data Structures, where AST structural comparison is clean; open-ended architecture or UI courses involve subjective trade-offs harder to verify statically. Second, AST parsing and guardrail checks add ~2.1 seconds of latency per turn, though prompt caching kept net API costs to $0.42 per student per month.

7. Conclusion : The arrival of frontier language models does not require turning university laboratories into surveillance states obsessed with AI plagiarism scores. Our 14-week study demonstrates that the real pedagogical threat is unmediated cognitive offloading. By interposing an AST-grounded **Cognitive Scaffolding Framework** between the learner and the LLM, educators can provide instant, individualized Socratic mentorship at scale—raising unassisted practical exam performance by 24.6 percentage points over unconstrained AI use while cultivating resilient, independent programmers. Future work will extend this framework to support multilingual scaffolding in Hindi and regional Indic languages for first-generation engineering learners.

References :

1. D. R. Cotton, P. A. Cotton, and J. R. Shipway, "Chatting and cheating: Ensuring academic integrity in the era of ChatGPT," *Innovations in Education and Teaching International*, vol. 61, no. 2, pp. 228–239, 2024.
2. V. Sadasivan et al., "Can AI-generated text be reliably detected?" *arXiv:2303.11156*, 2023.
3. M. Kapur, "Productive failure in learning math and computing," *Cognitive Science*, vol. 38, no. 5, pp. 1008–1022, 2014.
4. Ministry of Human Resource Development, *National Education Policy 2020*, Govt. of India, 2020.
5. S. Hoq et al., "Detecting ChatGPT-generated code submissions in a CS1 course," in *Proc. ACM SIGCSE*, 2024, pp. 526–532.
6. W. Liang et al., "GPT detectors are biased against non-native English writers," *Patterns*, vol. 4, no. 7, Art. no. 100779, 2023.
7. J. R. Anderson, A. T. Corbett, K. R. Koedinger, and R. Pelletier, "Cognitive tutors: Lessons learned," *Journal of the Learning Sciences*, vol. 4, no. 2, pp. 167–207, 1995.

8. A. Mitrovic, "Fifteen years of constraint-based tutors," *User Modeling and User-Adapted Interaction*, vol. 22, no. 1, pp. 39–72, 2012.
9. D. Wood, J. S. Bruner, and G. Ross, "The role of tutoring in problem solving," *J. Child Psychol. Psychiatry*, vol. 17, no. 2, pp. 89–100, 1976.
10. L. S. Vygotsky, *Mind in Society*. Harvard University Press, 1978.
11. K. VanLehn, "The relative effectiveness of human tutoring and intelligent tutoring systems," *Educational Psychologist*, vol. 46, no. 4, pp. 197–221, 2011.
12. M. Kazemitabaar et al., "Studying the effect of AI code generators on novice learners," in *Proc. ACM CHI*, 2023, pp. 1–23.
13. J. Prather et al., "Usability and interactions with Copilot for novice programmers," *ACM TOCE*, vol. 31, no. 1, pp. 1–31, 2024.
14. M. Liffiton et al., "CodeHelp: Using LLMs with guardrails in programming classes," in *Proc. Koli Calling*, 2023, pp. 1–11.
15. L. W. Anderson and D. R. Krathwohl, *A Taxonomy for Learning, Teaching, and Assessing*. Longman, 2001.

•