Beyond Answer Engines: LLMs as Reasoning Partners in Data Structures and Algorithms Education

Saad Zafar Khan[*]
University of Calgary
Department of Computer Science
Calgary, Alberta, Canada
saadzafar.khan@ucalgary.ca

Desiree Leal[†]
University of Calgary
Electrical and Software Engineering
Calgary, Alberta, Canada
desiree.leal1@ucalgary.ca

Lucas Valença
University of Calgary
Electrical and Software Engineering
Calgary, Alberta, Canada
lucas.rodriguesvalen@ucalgary.ca

Ahmad Abdellatif
University of Calgary
Electrical and Software Engineering
Calgary, Alberta, Canada
ahmad.abdellatif@ucalgary.ca

Mea Wang
University of Calgary
Department of Computer Science
Calgary, Alberta, Canada
meawang@ucalgary.ca

Diwakar Krishnamurthy
University of Calgary
Electrical and Software Engineering
Calgary, Alberta, Canada
dkrishna@ucalgary.ca

Ronnie de Souza Santos
University of Calgary
Electrical and Software Engineering
Calgary, Alberta, Canada
ronnie.desouzasantos@ucalgary.ca

Abstract

Large Language Models (LLMs) are rapidly entering software engineering education. As students turn to them for Data Structures and Algorithms (DSA) tutoring, a critical question emerges: do they foster genuine problem-solving skills or merely supply polished answers? Moreover, their competence in DSA remains underexplored beyond benchmarks likely contaminated with training data. We evaluate four LLMs—GPT-4o, Claude 3.7 Sonnet, Llama 3.2, and DeepSeek R1—on 80 LeetCode Top Interview 150 problems and 100 recent Weekly Contest problems, assessing code correctness, maintainability, and explanatory reasoning as core software engineering (SE) values. Results show that while Claude and DeepSeek perform best on harder problems, all models' average success rates drop by ∼49% on novel problems, revealing a critical gap in translating theoretical insights into effective implementations. On the novel problems, DeepSeek attained the highest success rate (70%) with fewer prompt turns but produced the least maintainable code, whereas Claude solved fewer (50%) yet generated the most maintainable solutions, highlighting a trade-off between correctness and pedagogical value. These findings suggest that LLMs are capable reasoning partners yet insufficient for autonomous problem-solving. Thus, educators must carefully integrate these tools into curricula to emphasize students' critical reasoning and debugging skills, and develop assessments that leverage or withstand LLM assistance. Our work contributes a rigorous, multidimensional evaluation framework and practical recommendations for AI-assisted learning in SE and computer science (CS) education, underscoring critical evaluation over reliance on automated solutions.

[*]These authors contributed equally to this work.
[†]These authors contributed equally to this work.

ICSE '26, Rio De Janeiro, Brazil
2026. ACM ISBN xxxx-xxxx-xxxx-x/YYYY/MM
https://doi.org//nnnnnnn.nnnnnnn

CCS Concepts

• **Social and professional topics** → **Software engineering education**; Student assessment; • **Computing methodologies** → *Natural language processing*; *Machine learning*; • **Theory of computation** → Design and analysis of algorithms.

Keywords

Generative Artificial Intelligence (AI), Large Language Models (LLMs), Software Engineering Education, Data Structures and Algorithms, LeetCode, AI-assisted Learning

1 Introduction

LLMs are reshaping software engineering (SE) practice and education—offering on-demand tutoring, generating/adapting learning materials, and supporting novel pedagogy and assessment [20, 35, 66]. Their capacity to generate, explain, and debug code makes them powerful aids for both students and professionals. Institutions (e.g., universities) are accordingly experimenting with policies to integrate LLMs into teaching [12]. Despite this growing adoption, a critical question remains unanswered: do LLMs foster deep, transferable problem-solving skills, or merely a facade of understanding? Empirical evidence is still scarce on whether the outputs learners consume (solutions, explanations, repairs) exhibit the properties known to support conceptual grasp, debugging skill development, and transfer to novel problems.

How students use LLMs. Students increasingly turn to LLMs to explain concepts, debug errors, plan solution steps, and draft or refactor code, with usage shaped by course goals and policies [37]. Adoption is broad for rapid feedback, summarization, and scaffolding, yet students and instructors remain wary about reliability and the need for verification [11, 32]. Use patterns are not uniform: some learners engage selectively while others rely heavily, and trust in GenAI influences both frequency and the kinds of tasks attempted

[4, 62]. Evidence warns that unchecked reliance can displace critical analysis and weaken debugging and decision-making skills [78]. In response, instructors increasingly favour guided use and assessment redesigns over blanket bans [42]. These dynamics leave open whether current LLM use actually strengthens transferable reasoning and debugging—precisely the gap our study interrogates.

Why DSA is the right testbed. DSA is a curricular cornerstone, underpinning software design, performance, and scalability in both computer science (CS2023, "Algorithmic Foundations" [39]) and SE curricula (SE2014, "algorithms, data structures, and complexity" [6]). Unlike open-ended domains such as software design or system architecture, DSA problems are objectively gradable yet require explicit reasoning about correctness and efficiency across core structures including arrays, trees, graphs, and dynamic programming. DSA pedagogy deliberately cultivates metacognitive skills through incremental practice: systematic decomposition, complexity analysis, and trade-off reasoning [52]. These are precisely the skills that LLMs may either scaffold or undermine, making DSA an ideal setting to assess whether model assistance fosters durable understanding rather than polished but fragile solutions. In SE education, these same competencies manifest in performance debugging, trade-off analysis, and maintainable implementations, positioning DSA as a controlled yet SE-relevant lens for evaluating the risks and benefits of AI-assisted learning.

Benchmark competence ≠ classroom learning. Frontier models perform strongly on code synthesis (generating source code from a higher-level specification) and repository-level code repair (e.g., GPT-4o and Llama-3.1 on HumanEval/MBPP; Claude 3.7 Sonnet and DeepSeek R1 on SWE-bench Verified and contest-style tasks) [5, 21, 22]. However, benchmark success does not guarantee learning outcomes: students still need durable mental models, the ability to assess complexity and efficiency, diagnose subtle faults, and generalize patterns to novel problems.

Prior work and instructor reports identify three tensions. (1) Students may overrely on LLMs, removing the cognitive work of analysis and debugging [42]. (2) Answers that look polished can conceal algorithmic inefficiencies or errors that students struggle to detect [79]. (3) Educators must adapt assessment and classroom practice to harness LLMs without eroding integrity or skill development [35, 41]. Consequently, there is a pressing need for a rigorous DSA evaluation that goes beyond functional correctness to assess the pedagogical quality of LLM outputs—including their explanatory value, maintainability, and utility in debugging.

In our study, we evaluate four LLMs — GPT-4o, Claude 3.7 Sonnet, Llama 3.2, DeepSeek R1—on 80 Top Interview (TI) tasks and 100 recent weekly contest (WC) problems from LeetCode. We use a few-shots chain-of-thought prompting strategy and score (i) correctness, (ii) efficiency (asymptotic and empirical), (iii) explanation quality via a rubric (invariants, edge cases, reasoning about complexity), and (iv) debugging on seeded faults. We define novelty as problems released after each model's stated training cutoff and mark all items with per-model novelty flags for analysis. Our study answers the following Research Questions (RQ):

RQ1. To what extent do LLM-generated solutions satisfy functional correctness and maintainability in novel problems?On canonical TI items, models appear "polished" with near-perfect solutions ($\geq$ 97.5%), but on 100 temporally novel WC problems, success rates drop sharply—DeepSeek R1: 70%, Claude 3.7: 50%, GPT-4o: 47%, Llama 3.2: 34%—revealing novelty brittleness behind fluent, template-like narratives. In short, presentation quality stays high while true generalization falters.

RQ2. How does the explanatory depth and theoretical justification of LLM outputs align with SE pedagogical frameworks? As difficulty rises, we observe a clear trade-off: models that push correctness tend to produce denser, harder-to-follow code (lower MI), whereas models that prioritize clarity solve fewer hard items (lower success). Despite the low average prompts-to-success (1.4–1.7), correctness, not extra prompting, limits performance under harder, varied tasks.

RQ3. How effectively do LLMs diagnose and repair faulty code to support the development of a learners debugging skills? Autonomous debugging remains limited: models rarely fix issues in a single pass and typically require 1.4–1.7 prompts. GPT/Claude/Llama often deliver "black-box" fixes, while DeepSeek provides "glass-box" diagnostics with explicit counterexamples and rationale that better support learner understanding.

Our findings translate into concrete, straightforward choices for instructors, curriculum leads, and teaching teams: (1) Adopt *novelty-aware* tasks (recent or held-out items) to counter benchmark optimism and better elicit transferable reasoning. (2) Grade with *quality gates*—maintainability and complexity thresholds alongside correctness—so polished but precarious code is not rewarded. (3) Require *glass-box* artifacts (code to concept alignment notes, edge case checklists, and debugging rationales) that keep the cognitive work with students even when LLMs are present. (4) Position LLMs as *reasoning partners*, not answer engines, via debug-and-improve assignments that use model output to scaffold analysis, refactoring, and fault diagnosis. We package these patterns in an *Instructor Playbook* with rubrics, templates, and ready-to-run scripts (e.g., MI/CC calculators), all of which are provided in our publicly available replication package on Zenodo [38], to enable immediate adoption in DSA and follow-on SE courses while preserving core algorithmic competencies.

2 Related Work

We synthesize prior work on LLM benchmarks and educational deployments to identify persistent gaps in evaluating engineered code quality and pedagogical processes.

2.1 LLMs for SE

Code generation benchmarks such as HumanEval, MBPP, and APPS foreground single-turn functional accuracy (typically `pass@k`), which limits visibility into explanation quality and maintainability. Recent work introduces contamination-controlled test suites and live, temporally novel streams (e.g., LiveCodeBench, LiveBench) and broadens tasks to self-repair, execution, and unit-test reasoning [13, 17, 30, 33, 69, 72]. Breakpoint stresses long-horizon repair by corrupting interdependent functions across system call graphs, testing whether LLMs can reason beyond local fixes [28]. Similarly, BigCodeBench requires compositional reasoning over diverse libraries and multi-tool instructions, highlighting deficits in realistic, large-scale code generation [80]. LLM-as-judge methods promise scalable rubric-aware scoring, but recent judge benchmarks report order sensitivity, prompt drift, and between-judge variance, motivating calibration protocols and human-rater reliability checks [19, 34].

LLM-based repair increasingly rivals or exceeds traditional APR on Defects4J-scale suites; agentic critique–revise loops (e.g., CodeAct) improve multi-turn repair and tool use [56, 71]. Still, most studies treat debugging as a binary outcome, with limited analysis of rationale quality or pedagogy. We therefore grade process competence—fault localization, rationale quality, and repair under bounded feedback budgets—so that improvements reflect teachable debugging behaviours rather than single-endpoint success.

2.2 LLMs for SE Education

Large classroom deployments show that LLM assistants can accelerate progress and reduce cognitive load, yet risk over-reliance and uneven learning outcomes [36, 49]. In novice debugging, pedagogically shaped chatbots improve help-seeking but require scaffolds to avoid shallow fixes [75]. LLM-generated worked examples and explanations can be perceived as clearer than peer-produced variants, but explanation correctness and teachability vary [46]. They can also suggest alternative implementations and refactoring approaches, exposing students to diverse coding patterns [64]. Recent studies on AI-assisted pair programming and prompting competence further highlight mixed effects on motivation, anxiety, and strategy use [23, 43].

Surveys and reviews highlight personalization benefits alongside concerns over academic integrity and "laziness" [7, 64]. Integrity challenges persist because LLMs can solve typical homework at scale, with over-reliance diminishing independent problem-solving abilities [63]. Multiple strands explore rubric-aware auto-grading, concept-based rubrics, and cross-task judge generalization in education, alongside integrity concerns where LLMs can complete CS1-style assignments [14, 48, 49, 57]. These studies indicate that while LLMs can accelerate development speed, they risk introducing fragile implementations that students lack the expertise to identify [64]. Across studies, the consistent recommendation is human-in-the-loop calibration with transparent rubrics and reliability reporting before operational deployment in coursework [14, 48, 57]. This suggests that the pedagogical value of LLMs lies not in their ability to act as *answer engines*, but as *reasoning partners*, simulating design rationales and traceability between requirements and implementation.

2.3 Evidence for Research Gap

Prior work surfaces three recurring gaps that motivate our pedagogically grounded evaluation of LLMs:

(G1) Metric myopia. Leaderboards emphasize single-turn `pass@k`, under-measuring explanation quality and maintainability [13, 17, 30, 33, 65]. Our study extends this by grading outputs not only on correctness but also on explanation quality, readability, and maintainability using quantitative gates (MI, CC). This moves beyond binary correctness to assess whether solutions are educationally usable.

(G2) Generalization under uncertainty. Public benchmarks risk contamination; temporally novel, problem-stream designs are required to assess out-of-distribution robustness [61, 69, 72]. We operationalize this recommendation by combining canonical datasets (Top Interview 150) with temporally novel Weekly Contest problems, showing that success rates drop by nearly half under novelty. This provides direct empirical evidence of the fragility of LLM competence in authentic educational settings.

(G3) Process opacity in pedagogy. Prompt traces exist, but explanation and debugging behaviours remain underexplored in authentic educational settings [36, 58, 67, 68, 74–76]. We address this by analyzing explanation quality across six rubric dimensions and by contrasting "black-box" vs. "glass-box" debugging. In doing so, we show how reasoning visibility (e.g., DeepSeek's iterative traces) creates pedagogical value beyond correctness.

In contrast to correctness-only leaderboards, our work contributes an education-ready evaluation of DSA problem-solving. We (i) grade explanations and maintainability alongside correctness via a multidimensional evaluation rubric; (ii) probe generalization using temporally novel problem streams; and (iii) ensure measurement reliability through human ratings with inter-rater agreement (κ). We use bounded, scaffolded interactions to further illustrate how models improve under fault-driven feedback, aligning evaluation metrics with teachable practices.

3 Methods

The main goal of this study is to evaluate how LLMs can be effectively used as reasoning partners to solve DSA problems. The following subsections outline the datasets, prompting strategy, interaction protocols, and evaluation metrics that structure this investigation. To support transparency and reproducibility, all datasets, prompts, model interactions, and evaluation results are available in the accompanying Zenodo repository [38].

3.1 Datasets

To evaluate the LLMs' performance on DSA problems, we resorted to LeetCode as the source platform because it is one of the most widely used repositories of DSA problems by both students and professionals [17, 72, 73], ensuring familiarity and relevance. Its large curated pool of problems also provides standardized difficulty levels and diverse algorithmic categories, making it a practical benchmark for evaluating LLM performance in educational contexts. We used datasets, summarized in Table 1.

Top Interview 150 [45]. This review dataset contains questions for various "must do" coding DSA interview problems. From this dataset, we analyzed 80 (easy, medium and hard) problems.

Weekly and Biweekly Contests [44]. The contest dataset spans from January 4th to May 31st 2025 (WC 431 [1]–452 [2]) and includes 100 post training-cutoff problems across a range of DSA topics, including arrays, strings, stacks, dynamic programming, binary trees, and graphs. These problems ensured broad coverage across difficulty levels while minimizing data leakage, as they were released after each models' known training cutoffs. For each contest item we recorded the public release date and tagged per-model novelty by comparing against the model's stated pre-training cutoffs (Table 2); the full problem list and flags are included in the artifact.

Evaluation Subset. To rigorously evaluate LLM performance, we curated a stratified subset of problems from the Weekly Contest benchmark. From an initial pool of 100 problems, we drew a stratified random sample of 45 (15 Easy, 15 Medium, 15 Hard), balancing solved and unsolved instances to preserve outcome variability. Each sampled problem was attempted by all four models, yielding 180 explanation artifacts (45 per model) that support within-problem contrasts under our analysis plan.

Table 1: Summary of Datasets by difficulty

Dataset	Total Used	Easy	Medium	Hard
Top Interview 150 (subset)	80	15	47	18
Weekly Contests	100	22	42	36

Table 2: Studied LLMs: Versions, Training Parameters etc

Model	Version / Release	Params	Training Cutoff	Context Window
GPT	gpt-4o-2024-08-06	~200B [3]	2023.09	128k tokens
Claude	claude-3-7-sonnet-20250219	350B	2024.10	200k tokens
Llama	llama-3.2 (2024.09)	90B / 11B*	2023.12	128k tokens
DeepSeek	deepseek-r1 (2025-01-20)	671B	Pre-2025	128k tokens

3.2 Models & Prompting

We evaluate four state-of-the-art LLMs—GPT-4o, Claude 3.7 Sonnet, Llama 3.2, and DeepSeek R1—because they are widely used and show strong performance on software-engineering tasks relevant to DSA [22, 27, 47]. The selected LLMs span proprietary services that students commonly use and open-weight baselines valued by instructors for inspection and reproducibility; they include both general-purpose assistants and code/reasoning-tuned models. This coverage serves our study goals—generalization to novel contest problems (RQ1), explanation quality for formative feedback (RQ2), and self-repair under debugging prompts (RQ3)—while enabling stakeholder-relevant comparisons for classroom policy and tooling and letting us separate the effects of model scale from training emphasis when selecting assistants for labs. Model specifications appear in Table 2.

Prompting protocol. To approximate realistic student usage while preserving comparability, we employ few-shot chain-of-thought prompt (i.e., step-by-step reasoning with 2–3 provided examples), which have been shown to be the most effective prompting strategy [77], with expert-role conditioning [54]; instructions require stepwise reasoning, algorithmic justification and asymptotic analysis, consistent with LeetCode-style tasks that explicitly mandate runtime bounds. Each prompt contains task content—problem statement, examples, constraints, and base starter code—reproduced verbatim from the corresponding LeetCode problem page. All models are provided with this identical content, without task-specific demonstrations or fine-tuning, to approximate naive student behaviour [47]. Figure 1 highlights the prompt structure.

Our problem dataset is representative but not exhaustive; proprietary systems evolve and some training details are undisclosed. We interpret results as comparative evidence under a fixed-in-time snapshot and standardized prompting strategy.

You are an expert in algorithms and data structures, skilled in solving computational problems efficiently. Carefully analyze the following problem and provide your response in two structured parts:
Part 1: Implement an optimized solution for the given problem, considering efficiency and readability. The solution should be written in Python.
Part 2: Explain your approach in detail, including:
- The choice of algorithm and data structures.
- The reasoning behind these choices.
- The time and space complexity of the solution.

Problem Statement: {*Description*}.
Example inputs and outputs: {*Example 1*}, {*Example 2*}
Constraints: {*Constraints*}.
Base Code: {*Base Code*}.
If multiple approaches exist, compare them and explain why your chosen approach is preferable under the given constraints.

Figure 1: Prompt Structure

3.3 Interaction Protocol

We evaluate each model under a standardized, bounded interaction protocol that approximates instructor–student scaffolding while enabling fair cross-model comparison.

Step 0 (Initialization / first attempt). For each problem, we prompt the LLM with a single, fixed base prompt that contains the full problem statement and I/O format and asks for a complete, executable solution. We submit the model's generated code (verbatim, without manual edits) to the LeetCode online judge (OJ) and record the platform's verdict and feedback: Accepted (AC), Wrong Answer (WA), Runtime Error (RE), or Time Limit Exceeded (TLE), along with the number of public tests passed/total and the OJ error message.

Step 1 (Failure feedback). If the submission is not AC, we *re-prompt the same model* with a structured debugging message that includes the relevant OJ feedback from LeetCode (e.g., failing test IDs, example input/output, and the platform error summary). We never reveal the reference solution.

Step 2 (Hint provision). If the second attempt is still not AC, we add the problem's official LeetCode hints *verbatim* to the next prompt, targeting the observed failure mode.

Step 3 (Iterative failure feedback). On subsequent non-AC attempts, we return any *newly* failing cases and updated platform messages from LeetCode. Interactions are capped at five total model messages per problem with the sequence:

$$\textit{Base} \longrightarrow \textit{Fail-info} \longrightarrow \textit{Hint} \longrightarrow \textit{Fail-info} \longrightarrow \textit{Final attempt.}$$

We allow up to five *code submissions to LeetCode* per problem with *early stopping on first AC*. Let Attempts-to-Pass (AtP) $\in \{1, 2, 3, 4, 5, \infty\}$ denote the index of the first accepted submission (or ∞ if unsolved within budget). A problem is *Solved* iff AtP ≤ 5; otherwise it is *Unsolved*, and we record the terminal LeetCode verdict (WA/TLE/RE) and best partial progress.

3.4 Experimental Setup

We orchestrate a uniform prompt $\rightarrow$ submit $\rightarrow$ feedback$\rightarrow$ re-prompt loop for all models: each LLM is prompted, its code is submitted to the LeetCode OJ, and the platform's verdict and feedback (verdict class, number of public tests passed, error message) drive the next attempt, up to five total. Figure 2 summarizes this end-to-end pipeline. We randomized problem order per model, ran within fixed windows to limit UI/version drift, and log UI/OJ versions in the artifact.

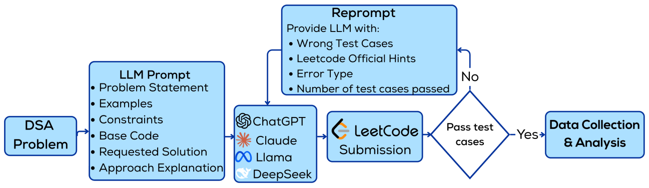

Figure 2: Experimental Setup

RQ1 (problem-solving performance). The base prompt elicits a complete solution. Because our protocol early-stops within five attempts, we *do not report pass@1*. Instead we measure: (1) AtP distribution for solved items; (ii) Partial Correctness—for unsolved items, the best fraction of public tests passed over the budget; and (iii) the resource profile of the *successful attempt only*: *Run time (ms)*, *Beats (run time) %*, *Memory (MB)*, and *Beats (memory) %*.
RQ2 (explanation quality). The prompt additionally requires a structured rationale, including chosen data structures/strategy and stated *time/space complexity* (Fig. 1). We log the raw complexity strings and score explanations with our rubric as in 3.5.2.
RQ3 (self-repair). Upon failure, we open a brief debugging loop: return minimal failing-case details (e.g., "passed 702/989; returns `false` on s = "gyye", $k = 3$; expected `true`—identify the fault and patch"). If the second attempt still fails, we provide the problem's all official hints verbatim from LeetCode and allow remaining attempts within the five-attempt budget. For example: "Rethink the problem with the following hints:

- Starting from each character, build the smallest special substring interval containing it.
- Use dynamic programming on the obtained intervals to check if it's possible to pick at least k disjoint intervals."

3.5 Evaluation Metrics

We conducted two evaluations: one assessing code quality, another assessing explanation quality, which are detailed below.

3.5.1 Code Quality Metrics. We evaluate LLM-produced code on three dimensions: maintainability, correctness, and efficiency. The detailed criteria are summarized in Table 3, followed by a description of the maintainability analysis pipeline.

Table 3: Code quality metrics and thresholds

Dimension	Metrics, Bands, and Interpretation
Maintainability	**(a) Maintainability Index (MI)** [18]: A composite metric that estimates how maintainable (i.e., readable and understandable) the code is. **Bands** [31]: 85–100 = highly maintainable, well-structured and easy to modify; 65–84 = moderately maintainable, some complexity or verbosity may slow comprehension; <65 = difficult to maintain, twisted logic or long methods. **(b) Cyclomatic Complexity (CC)** [51]: counts independent execution paths. **Bands**: Low $\leq$10 = simple, easy to test; 11–20 = moderate complexity; >20 = high complexity, hard to read, test, and refactor.
Correctness	**Test Case Pass Rate**: Fraction of LeetCode test cases passed, including edge cases, invalid inputs, and boundary conditions. A solution is correct if it consistently produces expected outputs across all categories.
Efficiency	**(a) Time Complexity (Big-O)**: asymptotic measure of worst-case runtime growth relative to input size. **(b) Space Complexity and Memory Usage Profiling**: evaluates extra memory beyond the input (auxiliary data structures) and overall memory consumption at runtime.

We augment evaluation with automated static analysis using *radon* [40]. For each model-produced Python solution, we compute: (i) Maintainability Index (MI) [18], and (ii) Cyclomatic Complexity (CC) [51] per function and $CC_{\max}$. A maintainability target of $MI \geq 70$ *and* $CC_{\max} \leq 10$ is used, reflecting pedagogy-oriented readability targets [18, 51]. For each (difficulty, model) cell, we report median MI and $CC_{\max}$, band distributions, and the proportion of solutions meeting the target. For between-model comparisons on MI and $CC_{\max}$ within each difficulty, we use paired Wilcoxon signed-rank tests [70] across problems (reporting median paired differences with 95% bootstrap CIs, 5,000 resamples, and Cliff's δ effect sizes) [15]. For the binary maintainability gate $\text{MI} \geq 70 \ \wedge \ CC_{\max} \leq 10$, we use McNemar's test [53] on matched problem pairs. To control multiplicity, we apply Benjamini–Hochberg (q=0.05) [9]. We also report Spearman MI–$CC_{\max}$ correlations.

3.5.2 Explanation Quality Metrics. Two researchers independently evaluated all 180 LLM outputs (per-dimension range 0.78–0.85). We score each explanation on six dimensions, each on a 1–5 ordinal scale (1 = insufficient, 3 = minimally adequate, 5 = exemplary) as in Table 4. To ensure alignment and calibration, a small subset of problems was jointly reviewed at the outset, and disagreements were resolved through discussion before proceeding with the full evaluation. To ensure fairness and consistency in scoring, we assessed inter-rater reliability using the weighted quadratic Cohen's kappa [16]. This choice reflects the ordinal nature of our rubric, where disagreements further apart on the scale are penalized more heavily. Inter-rater reliability was in substantial agreement, with an average Cohen's kappa of 0.82 across all evaluations. The summary can be seen in Table 5.
Drawing from constructivist learning theory [8], six different criteria were selected to comprehensively evaluate pedagogical effectiveness rather than technical correctness alone. These criteria also resonate with Bloom's taxonomy [10], as they progress from foundational understanding (e.g., clear examples, code-to-concept mapping) to higher-order reasoning (e.g., theoretical justification, detection of problems).
These metrics assess whether explanations facilitate deep understanding, systematic reasoning, and transfer learning, which are core competencies required for algorithmic problem-solving in professional software development contexts. Unlike metrics such as computational complexity, which measure algorithm properties, this criteria evaluates explanatory quality and instructional scaffolding essential for student learning outcomes.

Table 4: Rubric for Explanation Quality

Dimension	Description & Scale Anchors
Conceptual Depth	Measures how thoroughly the explanation covers the theory behind the algorithm and solution. *Scale:* 1 = vague restatement; 5 = articulates core ideas (invariants, subproblems/recurrence, trade-offs).
Step-by-Step Breakdown	Assesses whether the explanation is decomposed into clear, ordered steps. *Scale:* 1 = missing or muddled steps; 5 = clear, ordered reasoning with no gaps.
Theoretical Justification	Evaluates proofs or rationales explaining why the solution works. *Scale:* 1 = no rationale; 5 = concise proof sketch/loop invariant and time/space complexity.
Code-to-Concept Mapping	Checks how effectively the explanation links to the implementation. *Scale:* 1 = code and logic are disjoint; 5 = variables/control flow map cleanly to algorithmic roles.
Use of Examples	Examines whether illustrative examples clarify scenarios and edge cases. *Scale:* 1 = absent or irrelevant; 5 = well-chosen normal and edge case walkthrough.
Detection of Problems	Evaluates ability to identify faults, edge cases, or limitations. *Scale:* 1 = no issues identified; 5 = explicit diagnosis, cause, and principled fix.

3.5.3 Pedagogical validity of metrics. Our rubric's six dimensions align with well-established learning mechanisms in CS/SE education and cognitive science. Conceptual Depth and Code-to-Concept Mapping target schema construction and abstraction—central to novice program comprehension and developing competence [29, 60]. Step-by-Step Breakdown operationalizes systematic reasoning and decomposition; structured subgoaling improves problem-solving and transfer, and systematic strategies are associated with stronger novice debugging performance [24, 50]. Theoretical Justification elicits explicit rationales (e.g., invariants/properties), supporting correctness-oriented reasoning and higher-level algorithmic design as captured in novice assessments [26]. Use of Examples leverages the worked-example effect to foster schema induction, reduce cognitive load, and enhance pattern recognition and transfer in algorithmic problem solving [59]. Detection of Problems cultivates metacognitive monitoring and error diagnosis—processes foundational to effective problem solving and observed to differentiate more systematic novice behaviour [24, 25]. Finally, Maintainability metrics (MI/CC) provide pragmatic, widely used proxies for structural code qualities tied to understandability and maintenance effort in practice [55]. Collectively, these dimensions supply actionable indicators that AI-generated explanations are likely to support the cognitive processes—abstraction, decomposition, metacognition, and justification—linked to improved novice comprehension and algorithmic thinking.

4 Results and Discussion

This section presents the results of our multi-dimensional analysis. We structure our findings around the RQs, presenting the quantitative and qualitative evidence for each before discussing its implications for education.

4.1 RQ1: To what extent do LLM-generated solutions satisfy functional correctness and maintainability in novel problems?

To understand how LLMs generate and present DSA solutions, we evaluated both their correctness and code quality. Our analysis reveals two fundamental challenges that directly impact their educational utility: (1) inflated performance on familiar problems masking limited generalization, and (2) a systematic trade-off between solution correctness and maintainability that varies by model.

We evaluated model performance on the 80 questions from the Top Interview 150 dataset; GPT and DeepSeek solved all problems, Claude 98.8%, and Llama 97.5%. While superficially impressive, this uniformly high success rate strongly suggests data contamination—these problems have likely appeared in training corpora, allowing models to retrieve memorized solutions rather than demonstrate genuine problem-solving capability.

To isolate true generalization ability from memorization, we evaluated models on novel Weekly Contest problems published after each model's training cutoff. Performance collapsed dramatically. Success rates on problems fell to: DeepSeek 70% [60.4, 78.1], Claude 50% [40.4, 59.6], GPT 47% [37.5, 56.7], and Llama 34% [25.5, 43.7] (95% Wilson CIs). This represents an average performance drop of 48.8%. Moreover, detailed error analysis revealed that approximately 70% of failures stemmed from incorrect outputs rather than timeout or memory issues, indicating fundamental gaps in algorithmic reasoning rather than mere inefficiency. Many failing submissions still included fluent and well-structured reasoning, reinforcing that explanation detail is not a reliable indicator of correctness.

Table 5: Model Performance Metrics on Weekly Contests

Model	Easy	Medium	Hard	Total Success Rate	Average Prompts	Cohen's κ
GPT	21	23	3	47%	1.7	0.78
Claude	22	24	4	50%	1.5	0.82
Llama	15	18	2	34%	1.6	0.85
DeepSeek	22	32	17	70%	1.4	0.83
Average	–	–	–	–	–	**0.82**

Table 6: Static Analysis of Maintainability

					MI Dist. (%)			CC Dist. (%)			
Model	**n**	**MI Median**	CC_{avg}	$CC_{\max}$	**B1**	**B2**	**B3**	**B1**	**B2**	**B3**	**Pass (%)**
					Easy Problems						
GPT	15	88.59	4.50	5	53.3	33.3	13.3	93.3	6.7	0.0	73.3
Claude	15	84.19	4.50	5	46.7	46.7	6.7	93.3	6.7	0.0	86.7
Llama	15	80.63	5.33	6	46.7	13.3	40.0	93.3	6.7	0.0	53.3
DeepSeek	15	71.21	5.50	6	20.0	60.0	20.0	93.3	6.7	0.0	60.0
					Medium Problems						
GPT	15	77.61	7.50	8	21.4	64.3	14.3	78.6	21.4	0.0	57.1
Claude	15	74.72	8.50	10	26.7	73.3	0.0	60.0	33.3	6.7	60.0
Llama	15	68.18	7.00	8	6.7	46.7	46.7	80.0	13.3	6.7	26.7
DeepSeek	15	60.28	9.50	10	6.7	33.3	60.0	60.0	20.0	20.0	26.7
					Hard Problems						
GPT	15	69.37	8.50	9	0.0	64.3	35.7	57.1	42.9	0.0	21.4
Claude	15	75.66	9.50	10	6.7	80.0	13.3	53.3	33.3	13.3	13.3
Llama	15	60.57	10.50	11	6.7	26.7	66.7	46.7	53.3	0.0	13.3
DeepSeek	15	56.66	13.50	16	0.0	28.6	71.4	35.7	28.6	35.7	40.0

MI Dist Bins: B1 (High, $\geq$ 85), B2 (Moderate, 65-84), B3 (Low, < 65). CC Dist Bins: B1 (Low, $\leq$ 10), B2 (Moderate, 11-20), B3 (High, > 20).

Beyond correctness, we examined whether models maintain code quality under increasing difficulty, as readable, maintainable code is essential for learning; students must be able to understand, modify, and debug solutions to internalize concepts. Static analysis of fifteen randomly selected problems revealed systematic degradation across difficulty levels (Table 6). The median Maintainability Index (MI) dropped from 80.63 on Easy problems to 67.52 on Hard problems, while median maximum Cyclomatic Complexity (CC) rose from 6 to 11. Most critically, the proportion of solutions meeting both maintainability thresholds (MI $\geq$ 70, $CC_{\max} \leq 10$) fell from 68.3% on Easy to just 23.3% on Hard problems. This indicates that as algorithmic difficulty increases, generated code becomes systematically harder to read and maintain. As summarized in Table 7, LLM success is highly dependent on the complexity of the software task. Models show high reliability in simpler tasks like Arrays and Sorting, where they can generate standard solutions. However, correctness drops sharply in categories requiring complex state management, such as Dynamic Programming (DP) and Tree-based structures (success rates < 50%). This suggests that LLM fragility is tied to software architecture challenges where global state and complex data dependencies are central; this mirrors the maintainability degradation observed as problem difficulty increases.

When we compared model pairs on Hard problems, a clear Pareto trade-off emerged:

- DeepSeek maximized correctness (40.0% pass rate) but produced the least maintainable code (MI = 56.7, $CC_{\max}$ = 16)
- Claude sacrificed correctness (13.3% pass rate) but delivered the most maintainable code (MI = 75.7, $CC_{\max}$ = 10)

Table 7: LLM Competencies and Software Engineering Impact

SE Competency	Key Categories	Perf.	Design Impact
Basic Implementation	Array, Sort, Matrix	High	Low technical debt
Resource Optimization	Hash, Heap, B-Search	Med	Critical structure choice
Complex State / Logic	DP, Trees, Strings	Low	Architectural fragility

The results show a significant correctness–maintainability tradeoff in LLM outputs. As problem difficulty increases, models that maximize functional correctness (e.g. DeepSeek) systematically produce code with higher structural complexity and lower maintainability. Conversely, models that prioritize readability and modularity (e.g. Claude) tend to sacrifice correctness on complex items.

This distinction is critical for CS/SE education. The architectural choices made by the LLM—whether to prioritize a complex but correct solution or a simple but incomplete one—directly dictate its pedagogical use. An instructor might use a DeepSeek solution to teach advanced optimization and logic, while a Claude solution is much better suited to teaching fundamental skills such as refactoring, debugging, and maintaining code quality.

This tradeoff between correctness and maintainability highlights that the educational value of LLMs extends beyond whether a solution simply passes all test cases. For students, code is not simply a final product that is judged only by correctness, but also a medium of learning where readability, structure, and explanation all contribute. Thus, to fully understand their pedagogical utility, we must also look beyond code output to the way models communicate their implementation reasoning.

4.2 RQ2: How does the explanatory depth and theoretical justification of LLM outputs align with SE pedagogical frameworks?

To evaluate another pedagogical dimension, we next examine the explanatory quality of model outputs, focusing on how well LLMs scaffold understanding through conceptual depth, justification, and practical reasoning. Our analysis reveals a consistent textbook–tutor gap across models and difficulty levels. While LLMs function as excellent interactive textbooks, they largely fail as practical tutors, struggling to model critical skills of example generation and self-correction. As shown in Table 8, the models scored highly on "textbook" qualities. For instance, on Easy problems, average scores for Conceptual Depth (4.34/5) and Theoretical Justification (4.26/5) were strong, and remained robust on Hard problems (4.09/5 and 4.01/5, respectively). This indicates a clear ability to articulate algorithmic principles, map them to code, and explain efficiency in a structured manner.

Two independent graders scored all explanation criteria. Agreement was substantial across models (Cohen's $\kappa \in [0.78, 0.85]$, as in Table 5), supporting the stability of our reported rubric means.

However, this theoretical fluency breaks down when moving to practical, "tutor-like" skills. For GPT, Claude, and Llama, Use of Examples was a critical weakness, with scores rarely exceeding 2.1 and often falling below 1.5, indicating they merely rephrase examples from the prompt rather than generating novel ones. Telling is the decline in Detection of Problems on Hard problems, where scores for GPT (2.93), Claude (3.37), and Llama (2.77) reveal failure to self-critique when complexity increases. DeepSeek is a notable exception. It achieved a perfect 5/5 on Use of Examples across difficulties and consistently outperformed others on Detection of Problems (3.77 on Hard). This superior performance directly correlates with its visible, iterative reasoning traces, where it externalizes a process of proposing, testing, and refuting candidate solutions.

This sharp drop-off in performance suggests a form of non-linear brittleness. As problems shift from requiring formulaic explanation to genuinely abductive reasoning (e.g., discovering a tricky dynamic programming invariant or an off-by-one error in pointer arithmetic), most models continue to narrate theory but fail to operationalize diagnostic behaviours like generating counter-examples or reasoning about boundary conditions. This pattern mirrors a known challenge in human learning—the transition from passive explanation to active self-monitoring and test design.

For learners, LLMs act as excellent interactive textbooks—clarifying recurrences, invariants, and asymptotics—yet weak practical mentors: they rarely demonstrate testing discipline or error diagnosis. Uncritical use risks fluent-but-shallow understanding. For instruction, two interventions follow: (i) example-forcing prompts ("construct a failing test and explain why it fails") and (ii) diagnostic-first prompts ("state two plausible bugs and design a check for each"). These behaviours are not automatic byproducts of theoretical knowledge; they require explicit elicitation or models that natively externalize their reasoning.

These gaps motivated RQ3: if models fail to self-monitor, can they diagnose and repair faulty code when asked, and does that process itself aid comprehension? We therefore turn from explanation quality to debugging behaviour, measuring not only fix success but also the clarity of the diagnostic steps offered to learners.

4.3 RQ3: How effectively do LLMs diagnose and repair faulty code to support the development of a learners debugging skills?

Debugging capacity varied sharply between models. Table 8 shows that Detection of Problems scores were consistently among the lowest criteria, particularly on Hard problems (average = 3.21/5). This suggests that even when models were prompted with the failing test cases, they struggled to identify the specific source of error. Similarly, Table 5 shows that all models required multiple prompts to generate correct solutions, with averages ranging from 1.4 (DeepSeek) to 1.7 (GPT). This indicates that no model could reliably self-correct in a single pass, underscoring the difficulty of autonomous debugging.

Explanatory metrics reinforced these limitations: GPT, Claude, and Llama all received low scores on Use of Examples (1.10-2.13), rarely documenting how they used proved examples or generating additional test cases to verify their fixes. In contrast, DeepSeek's Detection of Problems scores were the highest, achieving 5.00, 4.07, and 3.87 on Easy, Medium and Hard problems respectively. Its explanations frequently included synthetic inputs, step-by-step traces of failing executions, and justification of why a repair worked. Figure 3 illustrates this: DeepSeek required 630 seconds to generate a corrected solution but produced a detailed trace of its reasoning process. Qualitative analysis revealed two debugging styles:

Table 8: Explanation quality scores across criteria, difficulty levels, and models across two graders

Criterion	Easy				Medium				Hard			
	GPT	Claude	Llama	DeepSeek	GPT	Claude	Llama	DeepSeek	GPT	Claude	Llama	DeepSeek
Conceptual Depth	4.17	4.50	4.03	4.67	4.10	4.20	4.00	4.37	4.10	4.17	3.70	4.37
Step-by-Step Breakdown	4.30	4.53	3.80	4.67	4.17	4.30	3.60	4.43	3.93	3.97	3.27	4.10
Theoretical Justification	4.03	4.30	3.90	4.47	4.13	4.23	3.70	4.50	4.03	4.20	3.50	4.30
Code-to-Concept Mapping	4.77	4.97	4.63	4.97	3.93	4.03	3.57	4.47	3.93	4.23	3.80	4.43
Use of Examples	1.87	1.50	1.67	5.00	1.47	1.30	1.10	5.00	2.13	2.13	1.37	5.00
Detection of Problems	4.60	4.93	4.47	5.00	3.60	3.73	3.10	4.23	2.93	3.37	2.77	3.77

We are getting the wrong answer for [5,2,3,1] (expected 2, but getting 3). Let's simulate the current algorithm on [5,2,3,1]. Initial: [5,2,3,1] with indices 0:5, 1:2, 2:3, 3:1. Heap: (5+2=7, 0,1), (2+3=5, 1,2), (3+1=4, 2,3) → min is 4 at (2,3). …

The problem: we have two entries for (1,2)—one with sum 5 (from initial) and one with sum 6 (after merge). We want the new one (6) to be considered. The old one (5) is stale because the value at index 2 changed. However, when we pop the old one $(5, 1, 2)$, we check: $alive[1]$ and $alive[2]$ are True? $next\_ptr[1] = 2$ (adjacent) and $prev\_ptr[2] = 1$ (adjacent), so we would merge (1,2) again. But that is incorrect because the value at index 2 is now 4, so the sum should be $2 + 4 = 6$, not 5. The heap entry $(5, 1, 2)$ is stale; we are not updating the heap when a node's value changes. Thus the heap contains pairs that are no longer valid after updates.

How to fix? We cannot update heap entries in `heapq` (no decrease–key), so we must avoid using stale pairs. …

How do we avoid processing stale pairs for the same indices? We can note that after a node is updated, any pair that has that node must be reinserted with the new value? But we don't have a way to remove the old ones.

Alternatively, we can store more state in the heap: not just the sum, but also the values that were used to compute the sum? Actually, even that is not sufficient because the same pair of indices might appear multiple times with different sums.

Instead, we can use a lazy heap: we accept that the heap has stale entries. We just skip them. …

Figure 3: DeepSeek Reasoning Snapshot During Code Repair

(1) GPT, Claude, and Llama operated as "black-box" debuggers: when presented with failing test cases, they produced corrected code but offered minimal reasoning. GPT sometimes structured its answers into problem identification, fix description, and validation, but still obscured the deeper reasoning process. This efficiency-oriented style produces correct outputs but yields low educational value, since it answers the *what* but hides the *why*.
(2) DeepSeek consistently acted as a "glass-box" tutor. Its superior scores on Detection of Problems and Use of Examples reflect a visible self-corrective process: fault localization, explanation of inefficiency, and justification for repairs. This transparency directly supports pedagogical goals: explicit reasoning aligns with Cognitive Apprenticeship by externalizing expert thought processes, with an example shown in Figure 3. For novice learners, such "glass-box" reasoning is invaluable for modeling systematic debugging.

The broader implication is that correctness alone is insufficient for debugging pedagogy. Black-box fixes may benefit advanced DSA learners seeking quick corrections, but they deprive novices of critical metacognitive scaffolding. DeepSeek's reasoning transparency highlights the potential LLMs have to be models of expert diagnostic reasoning, bridging the gap between conceptual knowledge and coding practice.

4.4 Implications for CS/SE Education

As LLMs become increasingly integrated into programming workflows, educators must prepare students not only to solve problems independently, but also to leverage these tools effectively while avoiding their documented limitations. This requires a fundamental shift from traditional algorithmic education that emphasizes the memorization of standard algorithms toward a more reflective approach that focuses on critical evaluation, optimization skills, and the ability to anticipate and handle edge cases.

Additionally, the varied performance of different LLM models across problem types and complexity levels suggests that educators should develop nuanced guidelines for appropriate model selection depending on specific learning objectives. SE and CS departments should also consider developing clear policies regarding acceptable LLM use in coursework that recognize these tools as valuable resources, while still ensuring that tudents develop fundamental problem-solving capabilities.

Our results highlight a set of challenges and opportunities for how LLMs should be integrated into computing education. We synthesize these implications around three recurring themes: (i) the challenge of attaining correctness under novelty, (ii) the pedagogical trade-off between correctness and maintainability, and (iii) the importance of transparency in debugging and reasoning. Based on our findings, we outline the following suggestions for designing CS/SE curricula.

4.4.1 Integrate LLMs as reasoning partners. Our results show that while LLMs can clearly articulate theoretical underpinnings (RQ2), they fail to consistently implement correct or efficient solutions (RQ1). To reduce black-box reliance, mandate students to externalize plans (e.g., data structure choices, expected complexity, anticipated edge cases) before coding that must align with the submitted solution. Such scaffolding reduces reliance on black-box outputs and helps learners connect conceptual reasoning to implementation. Furthermore, exposing students to the thinking process that DeepSeek demonstrated (RQ3) can provide a detailed approach to DSA selection, implementation, and verification of test cases. This suggests potential pedagogical value in demonstrating the importance of thorough problem analysis and iterative refinement in algorithmic thinking.

4.4.2 Re-evaluate Educational Priorities. Our results show that LLMs perform well on familiar benchmarks, but struggle to solve novel and harder problems, often failing correctness, efficiency, or maintainability criteria (RQ1-RQ2). This highlights the need for curricula and assessments that prioritize efficiency, robustness, and diagnostic clarity rather than correctness alone.

First, novelty-aware and LLM-resistant evaluations are essential. Prefer post-cutoff or freshly authored items; include hidden edge cases (e.g. non-textual images), and enforce explicit asymptotic (efficiency) targets. Second, balance correctness with maintainability through clear quality gates. We operationalize this with thresholds of MI≥70 and $CC_{\max}$≤10 for full style/maintainability credit, with partial credit for refactorings that improve MI/CC without breaking tests (see Playbook 5). Third, explicitly assess efficiency. In our Weekly Contest data, 30.65% of failures were due to time-limit exceeded (TLE) violations rather than purely incorrect DSA implementations. Students should therefore be required to analyze time/space complexity, not just produce working code. Exercises can be designed to provide a slow baseline for students to refine toward a target complexity.

4.4.3 Leverage debugging transparency. DeepSeek's transparent reasoning demonstrated pedagogical value by externalizing the diagnostic process of fault localization, test case generation, and repair justification (RQ3). In contrast, GPT, Claude, and Llama often corrected faults without explaining how. For novice learners, transparent reasoning provides metacognitive scaffolding aligned with Cognitive Apprenticeship theory, where expert thought processes are made explicit. Thus, educators should simulate this transparency by requiring students to describe their debugging rationale, reinforcing deeper critical reasoning.

4.4.4 Shape policy and practice in LLM use. Given the diversity of model performance, departments should establish policies for appropriate pedagogical LLM use. For example, instructors may allow models as conceptual scaffolds but restrict their use during high-stakes exams, or pair long-reasoning assistants (e.g., DeepSeek) with refactoring assistants (e.g., Claude) in instructional labs to highlight complementary strengths. Clear guidelines can prevent over-reliance while still recognizing the value of LLMs as learning tools.

We present our recommendations for instructors in an Instructor Playbook (Section 5), which includes policy templates, assignment blueprints, grading rubrics, and example artifacts [38].

5 Instructor Playbook—LLMs as Reasoning Partners (Data Structures & Algorithms)

Modern DSA courses face a practical dilemma: students now arrive with powerful assistants, yet most guidance and assessments still assume a pre-LLM classroom. Our playbook reframes LLMs from "answer engines" into *reasoning partners* under a bounded, auditable protocol—so instructors can harness explanation, planning, and debugging support without eroding core competencies or assessment fairness. It is built directly from our study's multi-dimensional evidence (correctness, code quality, explanatory depth, and self-repair) and packaged for immediate classroom use.

The playbook ships with ready-to-adopt materials that drop into a standard DSA syllabus:

- *Policy templates* that define LLMs as constrained, transparent assistants (bounded interactions, attribution, transcripts).
- *Assessment blueprints* that reward plan→code alignment, explanation quality, and maintainable implementations—not just final correctness.
- *Rubrics* for explanations, code quality/efficiency, and debugging competence that TAs can apply consistently.
- *Instructor/TA checklists* for quick integrity triage (budget compliance, plan freeze, transcript presence).
- *Task sourcing guidance* (novelty registry, topic tags) to keep problems fresh and resistant to memorized solutions.

How it helps a modern DSA curriculum.

- *Centers reasoning, not harvest.* Students must externalize a short plan (invariants, cases) before coding, then justify complexity and reflect on edge cases. This turns LLM use into a scaffold for thinking rather than a shortcut to answers.
- *Balances speed with quality.* Grading emphasizes readability and structure alongside correctness, so "passes tests but brittle" code no longer earns full credit.
- *Makes debugging teachable.* Fault-driven activities require locating and repairing issues with short rationale; instructors see *how* students reason about failures, not only whether they fix them.
- *Surfaces actionable signals.* Transcripts and mini-defenses provide fast indicators of authorship, grasp of trade-offs, and transfer to novel problems—useful for feedback loops and integrity checks.

To keep adoption simple, we provide plug-and-play patterns that align with common DSA learning outcomes:

- *Explain→Code (E→C):* plan-first submissions graded on plan–code alignment and complexity justification.
- *Debug&Improve (D&I):* intentionally flawed or borderline solutions that students must localize, repair, and test—making the debugging process visible and graded.
- *Contest Simulation (CT):* post-cutoff or parameterized variants under light time/attempt budgets to elicit transfer to novel problems.

Instructors adopt a simple weekly cadence (details in the toolkit):

(1) *Weekly labs:* short E→C tasks with transcripted, bounded LLM support; TA spot-checks with mini-defenses.
(2) *Periodic D&I labs:* students fix seeded faults and explain the repair; graders use one rubric across sections.
(3) *Two time-boxed contests:* novelty-aware items to measure transfer; plan cards and attempt caps keep assistance fair.
(4) *Summative checks:* LLM-restricted or LLM-free exams (paper coding, proof sketches) to verify individual mastery.

Using the playbook yields course-level analytics that inform teaching decisions:

- *Where students stall:* rubric trends highlight weak edges (e.g., example use, edge-case reasoning) even when code passes.
- *Trade-offs in practice:* patterns in readability vs. correctness guide which assistant to recommend for which activity.
- *Transfer readiness:* novelty-aware results separate memorized competence from adaptable problem solving.

These insights mirror the study's findings and give instructors concrete levers (task design, model policy, grading focus) rather than abstract recommendations

The appendix/companion artifact includes: syllabus-ready policy text; concise rubrics; lab/contest templates; TA checklists; and lightweight automation to streamline submission checks. Instructors can adopt a single lab first, then scale to a full 10–13 week rollout as needed.[1]

6 Threats to Validity

We organize threats as internal, construct, external, and reliability/replicability, with concrete mitigations applied throughout.

[1]All materials, including sample tasks and scripts, are archived with a persistent DOI.

6.1 Internal Validity

Data leakage and memorization. Canonical LeetCode items (e.g., Top Interview 150) may appear in pretraining corpora. To assess generalization under novelty, we added 100 post–training-cutoff Weekly/Biweekly Contest problems (Jan–May 2025). We release problem IDs and full transcripts to support independent verification.

Protocol-induced bias. Our bounded, scaffolded interaction (fail-info → hint → final) could favour models that benefit from staged feedback. We fixed the escalation schedule and exchange caps uniformly across models; ablations of scaffolding are left to future work.

Prompt sensitivity. Outputs are prompt-dependent. We used a single compact template for all tasks/models and publish it verbatim with conversations to enable prompt-robustness studies.

6.2 Construct Validity

Adequacy of code-quality metrics. Cyclomatic Complexity (CC), Maintainability Index (MI), and pass-rate correctness are proxies and may miss facets such as readability or evolvability. We pair static analysis with functional acceptance and efficiency, and report disagreements (e.g., higher CC with stronger coverage) to avoid over-interpreting any single metric.

Pedagogical value measurement. Explanation quality is rubric-scored (conceptual depth, stepwise clarity, theory, code–concept mapping, examples, fault diagnosis) and is partly subjective. Multiple raters were used; we report Cohen's κ and adjudicate disagreements with rubric anchors. Rubric and annotated examples are released.

Platform performance as pedagogical signal. While LeetCode acceptance measures functional correctness rather than direct learning gains, our multi-dimensional evaluation provides strong, theory-grounded proxies for educational utility. By triangulating correctness with explanation quality (via our validated rubric) and debugging transparency, we assess the pedagogical potential of LLM outputs—a necessary precursor to classroom impact studies.

6.3 External Validity

Task/domain coverage. LeetCode emphasizes algorithmic kernels under interview-style constraints; results may not transfer to systems programming or multi-module design. We stratified by topic (arrays, graphs, DP, etc.) and difficulty (easy/medium/hard) and included contest items.

Language/toolchain scope. We requested Python solutions; other languages (C++/Java) and IDE/tooling may shift efficiency trade-offs and failure modes. Replication across languages and environments is recommended.

6.4 Reliability and Replicability

Stochasticity and reproducibility. LLMs are nondeterministic and web UIs hide temperature. We repeated attempts where feasible under a fixed interaction budget and report acceptance (Solved/Unsolved). Performance and explanation depth varied across attempts and models; a more rigorous design would average multiple trials per problem, which is beyond scope and noted for future work. We release prompts, all turns, final code, and platform outcomes to enable exact procedural replication.

Sample size and selection. Our 180-problem corpus spans topics and difficulties but is not exhaustive. We mitigated selection bias via stratified sampling and inclusion of unseen contest items; larger preregistered samples would increase power.

Platform evolution. LeetCode may update hidden tests/constraints, affecting verdicts. We log contest IDs, timestamps, failing I/O exemplars, and acceptance statuses to anchor comparisons.

Key mitigations include (i) mixing canonical with post-cutoff contest problems, (ii) a transparent, bounded scaffolding protocol and fixed prompt template, (iii) time-bounded runs with documented versions, (iv) triangulation of code, efficiency, and explanation quality with inter-rater agreement, and (v) full artifact release.

7 Conclusion and Future Work

We contribute a multi-dimensional evaluation framework that extends beyond correctness to encompass explanation quality, maintainability, and debugging competence. When confronted with novel problems that better approximate authentic coursework, all four LLMS' exhibited sharp performance declines, with success rates dropping by approximately 49% and frequent breakdowns in plan-to-code fidelity, robust debugging, and efficient implementation. Code maintainability also degraded in parallel with problem difficulty. This disconnect between conceptual understanding and practical implementation was striking — models articulated strong explanations (mean Conceptual Depth = 4.12/5) yet struggled to apply them effectively (Use of Examples ≤ 2.13/5 for most models). These findings frame LLMs as valuable reasoning partners rather than autonomous problem-solvers.

Ultimately, this work argues for a pedagogical shift: instead of asking "How can we stop students from using LLMs?", we should be asking, "How can we design curricula that cultivate the uniquely human skills of critical judgment, deep debugging, and rigorous evaluation that these powerful tools currently lack?"

Looking ahead, we will (i) validate these patterns in situ through multi-institution deployments and track longitudinal transfer from foundational DSA courses into later SE experiences (e.g., testing, capstone); (ii) study human–AI teaming in pair programming and peer review, assessing whether an LLM can act as a "third partner" that flags edge cases and maintainability risks; (iii) operationalize two concrete pedagogical tools—a plan-to-code consistency checker and an interactive refactoring tutor—to directly address observed failure modes; and (iv) evaluate equity and accessibility to ensure that AI-augmented pedagogy benefits learners with diverse preparation levels and needs. By moving beyond correctness-centric benchmarks to a pedagogically-grounded evaluation, this work provides the evidence and the tools needed to thoughtfully integrate LLMs as partners in the cultivation of durable algorithmic reasoning skills.

References

[1] 2025. *LeetCode Weekly Contest 431*. https://leetcode.com/contest/weekly-contest-431/ Accessed 8 September 2025; contest date from standings resource.

[2] 2025. *LeetCode Weekly Contest 452*. https://leetcode.com/contest/weekly-contest-452/ Accessed 8 September 2025; contest date from standings resource.

[3] Asma Ben Abacha, Wen wai Yim, Yujuan Fu, Zhaoyi Sun, Meliha Yetisgen, Fei Xia, and Thomas Lin. 2025. MEDEC: A Benchmark for Medical Error Detection and Correction in Clinical Notes. arXiv:2412.19260 [cs.CL] https://arxiv.org/abs/2412.19260

[4] Matin Amoozadeh, David Daniels, Daye Nam, Aayush Kumar, Stella Chen, Michael Hilton, Sruti Srinivasa Ragavan, and Mohammad Amin Alipour. 2024. Trust in Generative AI among students: An Exploratory Study. arXiv:2310.04631 [cs.HC] https://arxiv.org/abs/2310.04631

[5] Anthropic. 2025. *Claude 3.7 Sonnet and Claude Code*. https://www.anthropic.com/news/claude-3-7-sonnet

[6] Mark Ardis, David Budgen, Gregory W. Hislop, Jeff Offutt, Mark Sebern, and Willem Visser. 2015. SE 2014: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering. *Computer* 48, 11 (2015), 106–109. doi:10.1109/MC.2015.345

[7] R. Azoulay, T. Hirst, and S. Reches. 2025. Large Language Models in Computer Science Classrooms: Ethical Challenges and Strategic Solutions. *Applied Sciences* 15, 4 (2025), 1793. doi:10.3390/app15041793

[8] Steve Olusegun Bada. 2015. Constructivism Learning Theory: A Paradigm for Teaching and Learning. *IOSR Journal of Research & Method in Education (IOSR-JRME)* 5, 6 Ver. I (2015), 66–70. doi:10.9790/7388-05616670

[9] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)* 57, 1 (1995), 289–300. doi:10.1111/j.2517-6161.1995.tb02031.x

[10] Benjamin S. Bloom, Max D. Engelhart, Edward J. Furst, Walter H. Hill, and David R. Krathwohl. 1956. *Taxonomy of Educational Objectives: Handbook I, Cognitive Domain*. David McKay Company, New York. Handbook I of *Handbook: The Classification of Educational Goals*.

[11] Ritvik Budhiraja, Ishika Joshi, Jagat Sesh Challa, Harshal D. Akolekar, and Dhruv Kumar. 2024. "It's not like Jarvis, but it's pretty close!" - Examining ChatGPT's Usage among Undergraduate Students in Computer Science. In *Proceedings of the 26th Australasian Computing Education Conference (ACE 2024)*. ACM, 124–133. doi:10.1145/3636243.3636257

[12] C.K.Y. Chan. 2023. A Comprehensive AI Policy Education Framework for University Teaching and Learning. *International Journal of Educational Technology in Higher Education* 20, 1 (2023), 38. doi:10.1186/s41239-023-00408-3

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[14] Yucheng Chu, Hang Li, Kaiqi Yang, Harry Shomer, Hui Liu, Yasemin Copur-Gencturk, and Jiliang Tang. 2025. A LLM-Powered Automatic Grading Framework with Human-Level Guidelines Optimization. arXiv:2410.02165 [cs.AI] https://arxiv.org/abs/2410.02165

[15] Norman Cliff. 1993. Dominance Statistics: Ordinal Analyses to Answer Ordinal Questions. *Psychological Bulletin* 114, 3 (1993), 494–509.

[16] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. doi:10.1177/001316446002000104

[17] Tristan Coignion, Clément Quinton, and Romain Rouvoy. 2024. A Performance Study of LLM-Generated Code on Leetcode. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024)*. ACM, 79–89. doi:10.1145/3661167.3661221

[18] D. Coleman, D. Ash, B. Lowther, and P. Oman. 1994. Using metrics to evaluate software system maintainability. *Computer* 27, 8 (1994), 44–49. doi:10.1109/2.303623

[19] Giuseppe Crupi, Rosalia Tufano, Alejandro Velasco, Antonio Mastropaolo, Denys Poshyvanyk, and Gabriele Bavota. 2025. On the Effectiveness of LLM-as-a-judge for Code Generation and Summarization. arXiv:2507.16587 [cs.SE] https://arxiv.org/abs/2507.16587

[20] Paul Denny, James Prather, Brett A. Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N. Reeves, Eddie Antonio Santos, and Sami Sarsa. 2024. Computing Education in the Era of Generative AI. *Commun. ACM* 67, 2 (Jan. 2024), 56–67. doi:10.1145/3624720

[21] Aaron Grattafiori et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[22] DeepSeek-AI et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs.CL] https://arxiv.org/abs/2501.12948

[23] Guangrui Fan, Dandan Liu, Rui Zhang, and Lihu Pan. 2025. The impact of AI-assisted pair programming on student motivation, programming anxiety, collaborative learning, and programming performance: a comparative study with traditional pair programming and individual approaches. *International Journal of STEM Education* 12, 1 (2025), 16. doi:10.1186/s40594-025-00537-3

[24] Sue Fitzgerald, Gary Lewandowski, Renée McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education* 18, 2 (2008), 93–116. doi:10.1080/08993400802114508

[25] John H. Flavell. 1976. Metacognitive aspects of problem solving. In *The Nature of Intelligence*, Lauren B. Resnick (Ed.). Lawrence Erlbaum Associates, 231–235.

[26] David Ginat and Eti Menashe. 2015. SOLO Taxonomy for Assessing Novices' Algorithmic Design. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 452–457. doi:10.1145/2676723.2677311

[27] Kunal Handa, Drew Bent, Alex Tamkin, Miles McCain, Esin Durmus, Michael Stern, Mike Schiraldi, Saffron Huang, Stuart Ritchie, Steven Syverud, Kamya Jagadish, Margaret Vo, Matt Bell, and Deep Ganguli. 2025. *Anthropic Education Report: How University Students Use Claude*. https://www.anthropic.com/news/anthropic-education-report-how-university-students-use-claude

[28] Kaivalya Hariharan, Uzay Girit, Atticus Wang, and Jacob Andreas. 2025. Breakpoint: Scalable evaluation of system-level reasoning in LLM code agents. arXiv:2506.00172 [cs.LG] https://arxiv.org/abs/2506.00172

[29] Orit Hazzan, Tami Lapidot, and Noa Ragonis. 2011. *Guide to Teaching Computer Science: An Activity-Based Approach*. Springer. doi:10.1007/978-0-85729-443-2

[30] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. *arXiv preprint arXiv:2105.09938* (2021).

[31] Tjaša Heričko and Boštjan Šumak. 2023. Exploring Maintainability Index Variants for Software Maintainability Measurement in Object-Oriented Systems. *Applied Sciences* 13, 5 (2023). doi:10.3390/app13052972

[32] Irene Hou, Sophia Metille, Zhuo Li, Owen Man, Cynthia Zastudil, and Stephen MacNeil. 2024. The Effects of Generative AI on Computing Students' Help-Seeking Preferences. arXiv:2401.02262 [cs.HC] https://arxiv.org/abs/2401.02262

[33] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. arXiv:2403.07974 [cs.SE] https://arxiv.org/abs/2403.07974

[34] Hongchao Jiang, Yiming Chen, Yushi Cao, Hung yi Lee, and Robby T. Tan. 2025. CodeJudgeBench: Benchmarking LLM-as-a-Judge for Coding Tasks. arXiv:2507.10535 [cs.CL] https://arxiv.org/abs/2507.10535

[35] E. Kasneci, K. Sessler, S. Küchemann, M. Bannert, D. Dementieva, F. Fischer, et al. 2023. ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and Individual Differences* (2023). doi:10.1016/j.lindif.2023.102274

[36] Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, et al. 2024. CodeAid: Evaluating a Classroom Deployment of an LLM-based Programming Assistant that Balances Student and Educator Needs. In *CHI '24*. doi:10.1145/3613904.3642773

[37] Hieke Keuning, Isaac Alpizar-Chacon, Ioanna Lykourentzou, Lauren Beehler, Christian Köppe, Imke de Jong, and Sergey Sosnovsky. 2024. Students' Perceptions and Use of Generative AI Tools for Programming Across Different Computing Courses. arXiv:2410.06865 [cs.CY] https://arxiv.org/abs/2410.06865

[38] Saad Zafar Khan, Desiree Leal, Lucas Valença, Ahmad Abdellatif, Mea Wang, Diwakar Krishnamurthy, and Ronnie de Souza Santos. 2025. Replication Package for: Beyond Answer Engines: LLMs as Reasoning Partners in Data Structures and Algorithms Education. https://zenodo.org/records/17229516

[39] Amruth N. Kumar, Rajendra K. Raj, Sherif G. Aly, Monica D. Anderson, Brett A. Becker, Richard L. Blumenthal, Eric Eaton, Susan L. Epstein, Michael Goldweber, Pankaj Jalote, Douglas Lea, Michael Oudshoorn, Marcelo Pias, Susan Reiser, Christian Servin, Rahul Simha, Titus Winters, and Qiao Xiang. 2024. *Computer Science Curricula 2023*. Association for Computing Machinery, New York, NY, USA.

[40] Michele Lacchia. 2023. Radon: Code Metrics in Python (v6.0.1). https://pypi.org/project/radon/. Python package, MIT License, computes code metrics including cyclomatic complexity, raw metrics, Halstead metrics, and Maintainability Index.

[41] K. Y. Lau and S. Sotiriadis. 2023. Learning to program with large language models: A case study with ChatGPT. *Proceedings of the 28th ACM Conference on Innovation and Technology in Computer Science Education* 1 (2023). doi:10.1145/3587102.3588830

[42] Sam Lau and Philip Guo. 2023. From "Ban It Till We Understand It" to "Resistance is Futile": How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools such as ChatGPT and GitHub Copilot. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1* (Chicago, IL, USA) *(ICER '23)*. Association for Computing Machinery, New York, NY, USA, 106–121. doi:10.1145/3568813.3600138

[43] D. Lee and E. Palmer. 2025. Prompt engineering in higher education: a systematic review to help inform curricula. *International Journal of Educational Technology in Higher Education* 22, 7 (2025). doi:10.1186/s41239-025-00503-7

[44] LeetCode. 2025. LeetCode Contest. https://leetcode.com/contest/. Accessed: 2025-02-28.

[45] LeetCode. 2025. Top Interview 150. https://leetcode.com/studyplan/top-interview-150/. Accessed: 2025-02-28.

[46] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. 2023. Comparing Code Explanations Created by Students and Large Language Models. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V.1*. 124–130. doi:10.1145/3587102.3588785

[47] Nero Li, Shahar Broner, Yubin Kim, Katrina Mizuo, Elijah Sauder, Claire To, Albert Wang, Ofek Gila, and Michael Shindler. 2025. Investigating the Capabilities of Generative AI in Solving Data Structures, Algorithms, and Computability Problems. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V.1*. 659–665. doi:10.1145/3641554.3701946

[48] Yi Liu et al. 2025. Evaluating LLMs for Automated Scoring in Formative Assessments of a Programming Course. *Applied Sciences* 15, 5 (2025), 2787. doi:10.3390/app15052787

[49] Wenhan Lyu, Yimeng Wang, Tingting (Rachel) Chung, Yifan Sun, and Yixuan Zhang. 2024. Evaluating the Effectiveness of LLMs in Introductory Computer Science Education: A Semester-Long Field Study. 63–74 pages. doi:10.1145/3657604.3662036

[50] Lauren E. Margulieux and Richard Catrambone. 2016. Improving problem solving with subgoal labels in expository text and worked examples. *Learning and Instruction* 42 (2016), 58–71. doi:10.1016/j.learninstruc.2015.12.002

[51] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. doi:10.1109/TSE.1976.233837

[52] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander. 2008. Teaching debugging skills in the 21st century. *Proceedings of the 13th annual conference on Innovation and technology in computer science education* (2008). doi:10.1145/1384271.1384387

[53] Quinn McNemar. 1947. Note on the Sampling Error of the Difference between Correlated Proportions or Percentages. *Psychometrika* 12, 2 (1947), 153–157. doi:10.1007/BF02295996

[54] Ethan R. Mollick and Lilach Mollick. 2023. *Assigning AI: Seven Approaches for Students, with Prompts*. Technical Report. The Wharton School Research Paper. doi:10.2139/ssrn.4475995 Available at SSRN: https://ssrn.com/abstract=4475995 or http://dx.doi.org/10.2139/ssrn.4475995.

[55] P. Oman and J. Hagemeister. 1992. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992*. 337–344. doi:10.1109/ICSM.1992.242525

[56] Yicheng Ouyang, Jun Yang, and Lingming Zhang. 2024. Benchmarking Automated Program Repair: An Extensive Study on Both Real-World and Artificial Bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 440–452. doi:10.1145/3650212.3652140

[57] Aditya Pathak, Rachit Gandhi, Vaibhav Uttam, Arnav Ramamoorthy, Pratyush Ghosh, Aaryan Raj Jindal, Shreyash Verma, Aditya Mittal, Aashna Ased, Chirag Khatri, Yashwanth Nakka, Devansh, Jagat Sesh Challa, and Dhruv Kumar. 2025. Rubric Is All You Need: Improving LLM-Based Code Evaluation With Question-Specific Rubrics. In *Proceedings of the 2025 ACM Conference on International Computing Education Research V.1*. 181–195. doi:10.1145/3702652.3744220

[58] Haritz Puerto, Martin Tutek, Somak Aditya, Xiaodan Zhu, and Iryna Gurevych. 2024. Code Prompting Elicits Conditional Reasoning Abilities in Text+Code LLMs. *arXiv preprint arXiv:2401.10065* (2024).

[59] Alexander Renkl. 2014. Toward an instructionally oriented theory of example-based learning. *Cognitive Science* 38, 1 (2014), 1–37. doi:10.1111/cogs.12086

[60] Anthony Robins, Jennifer Rountree, and Nathan Rountree. 2003. Learning and teaching programming: A review and discussion. *Computer Science Education* 13, 2 (2003), 137–172. doi:10.1076/csed.13.2.137.14200

[61] Oscar Sainz, Jon Campos, Iker García-Ferrero, Julen Etxaniz, Oier Lopez de Lacalle, and Eneko Agirre. 2023. NLP Evaluation in trouble: On the Need to Measure LLM Data Contamination for each Benchmark. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 10776–10787. doi:10.18653/v1/2023.findings-emnlp.722

[62] Ana Stojanov, Qian Liu, and Joyce Hwee Ling Koh. 2024. University students' self-reported reliance on ChatGPT for learning: A latent profile analysis. *Computers and Education: Artificial Intelligence* 6 (2024), 100243. doi:10.1016/j.caeai.2024.100243

[63] Marielle Justine Sumilong. 2025. Instructional affect and learner motivation in generative AI-restrictive and permissive classrooms. *Frontiers in Education* Volume 10 - 2025 (2025). doi:10.3389/feduc.2025.1626802

[64] Wannita Takerngsaksiri, Cleshan Warusavitarne, Christian Yaacoub, Matthew Hee Keng Hou, and Chakkrit Tantithamthavorn. 2024. Students' Perspective on AI Code Completion: Benefits and Challenges. arXiv:2311.00177 [cs.SE] https://arxiv.org/abs/2311.00177

[65] Lun Wang, Chuanqi Shi, Shaoshui Du, Yiyi Tao, Yixian Shen, Hang Zheng, Yanxin Shen, and Xinyu Qiu. 2025. Performance Review on LLM for solving leetcode problems. arXiv:2502.15770 [cs.SE] https://arxiv.org/abs/2502.15770

[66] Shen Wang, Tianlong Xu, Hang Li, Chaoli Zhang, Joleen Liang, Jiliang Tang, Philip S. Yu, and Qingsong Wen. 2024. Large Language Models for Education: A Survey and Outlook. arXiv:2403.18105 [cs.CL] https://arxiv.org/abs/2403.18105

[67] Xuezhi Wang, Jason Wei, Dale Schuurmans, et al. 2022. Self-Consistency Improves Chain of Thought Reasoning in Language Models. *arXiv preprint arXiv:2203.11171* (2022).

[68] Jason Wei, Xuezhi Wang, Dale Schuurmans, et al. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *arXiv preprint arXiv:2201.11903* (2022).

[69] Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Ben Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Sreemanti Dey, Shubh-Agrawal, Sandeep Singh Sandha, Siddartha Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. 2025. LiveBench: A Challenging, Contamination-Limited LLM Benchmark. arXiv:2406.19314 [cs.CL] https://arxiv.org/abs/2406.19314

[70] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83.

[71] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*.

[72] Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Huifeng Sun, Siyue Wu, Jian Hu, and Xiaolong Xu. 2025. LeetCodeDataset: A Temporal Dataset for Robust Evaluation and Efficient Training of Code LLMs. arXiv:2504.14655 [cs.LG]

[73] Jessica Xing. 2021. Here's what job seekers need to know about LeetCode, the coding-skills platform millions of developers use to ace the notoriously difficult technical interviews at firms such as Apple, Amazon, and Google. *Business Insider* (Nov. 2021). https://www.businessinsider.com/leetcode-coding-test-apple-amazon-google-technical-interview-prep-job-2021-11

[74] Yuankai Xue, Hanlin Chen, Gina R. Bai, Robert Tairas, and Yu Huang. 2024. Does ChatGPT Help with Introductory Programming? An Experiment of Students Using ChatGPT in CS1. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET '24)*. 331–341. doi:10.1145/3639474.3640076

[75] Stephanie Yang, Hanzhang Zhao, Yudian Xu, Karen Brennan, and Bertrand Schneider. 2024. Debugging with an AI Tutor: Investigating Novice Help-seeking Behaviors and Perceived Learning. In *ICER '24*. doi:10.1145/3632620.3671092

[76] Shunyu Yao, Dian Bosma, Jeffrey Zhao, et al. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. *arXiv preprint arXiv:2305.10601* (2023).

[77] Adam Yuen, John Pangas, Md Mainul Hasan Polash, and Ahmad Abdellatif. 2025. Prompting Matters: Assessing the Effect of Prompting Techniques on LLM-Generated Class Code. In *Proceedings of ICSME 2025, NIER Track*. Case Room 3, ICSME 2025.

[78] Chunpeng Zhai, Santoso Wibowo, and Lily D. Li. 2024. The effects of over-reliance on AI dialogue systems on students' cognitive abilities: a systematic review. *Smart Learning Environments* 11 (2024). doi:10.1186/s40561-024-00316-7

[79] Kyrie Zhixuan Zhou, Zachary Kilhoffer, Madelyn Rose Sanfilippo, Ted Underwood, Ece Gumusel, Mengyi Wei, Abhinav Choudhry, and Jinjun Xiong. 2024. The teachers are confused as well: A Multiple-Stakeholder Ethics Discussion on Large Language Models in Computing Education. arXiv:2401.12453 https://arxiv.org/abs/2401.12453

[80] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, and et al. 2025. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. *arXiv preprint arXiv:2406.15877* (2025).