Preprints.org

Article

Not peer-reviewed version

The Code Council: Orchestrating Heterogeneous Large Language Models for Robust Programming Scaffolding

Daniel M. Muepu *, Yutaka Watanobe, Md Faizul Ibne Amin, Md. Shahajada Mia

Posted Date: 4 March 2026

doi: 10.20944/preprints202603.0350.v1

Keywords: large language models; multi-agent systems; automated debugging; intelligent tutoring systems; computing education; program repair

Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

The Code Council: Orchestrating Heterogeneous Large Language Models for Robust Programming Scaffolding

Daniel M. Muepu [1,2,*], Yutaka Watanobe [1], Md Faizul Ibne Amin [1] and Md. Shahajada Mia [1]

[1] The University of Aizu, Aizu-Wakamatsu, Fukushima 965-8580, Japan
[2] The University of Kinshasa, Kinshasa, D.R. Congo
* Correspondence: mdaniel108@gmail.com

Abstract

Recent advances in large language models (LLMs) have made it feasible to use them as automated debugging tutors, but it remains unclear how much can be gained by moving from single-model tutors to multi-agent councils with separated roles. We study this question in an offline simulation on 200 debugging cases drawn from an online judge, spanning 20 problems split into course-style and contest-style challenge tracks. We compare four single-model tutors based on current frontier models with four councils that assign models to Architect, Skeptic, Secretary, Pedagogue, and Mentor roles and operate in both Blind and Guided modes. Single-model tutors achieve near-perfect repair on course problems but perform less reliably on challenge cases and often rewrite large portions of student code, show non-negligible false positive rates, and leak full or near-full solutions in a substantial share of hints. Councils designed around measured model strengths improve both technical and pedagogical behaviour. On the challenge track, the best council raises patch success by 12.2 percentage points over the best single tutor, while reducing false positives, shrinking median patch size, improving hint localisation, and cutting solution leakage in Blind mode from about one fifth of hints to under ten percent. Councils also exhibit higher stability across reruns and produce hints that two independent instructors consistently rate as more useful and better scaffolded. Guided mode, where internal components see a reference solution, yields further technical gains but introduces leakage risks that require prompt tightening and a sanitising Secretary to control the flow of ground truth. Additional trap experiments with poisoned reference solutions show a mix of resistance and fail-safe collapse rather than systematic poisoning of hints. These results indicate that orchestration and information flow are powerful levers and that well-designed councils can provide more reliable and pedagogically aligned debugging support than strong single-model tutors alone.

Keywords: large language models; multi-agent systems; automated debugging; intelligent tutoring systems; computing education; program repair

---

1. Introduction

LLMs are starting to appear in computer science courses and may change the way engineering students learn programming [1,2]. At the moment, most universities do not yet include these tools in an official and systematic way [3]. However, LLMs already generate code, explain programs, and suggest fixes on demand [4]. This gives a preview of a learning environment where many students can receive help that feels personal and immediate, even when a human tutor is not available. In theory, such support could address Bloom's Two Sigma Problem, which suggests that average students can reach the level of the best students when they receive one-to-one tutoring [5].

In practice, turning this promise into real learning gains is not simple. An AI tutor that writes full solutions for students can damage the development of important problem solving skills [6]. The real value of an AI assistant in programming education comes from support for debugging [7,8]. Constructionism, a

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

central idea in learning sciences, states that students learn deeply when they struggle in a productive way, create artifacts, test them, and refine them over time [9,10]. In this view, the main task of an AI tutor is to help students understand and repair errors in their own code [11]. The tutor should not replace the creative and analytical work that the student needs to do before a first attempt [12].

At the same time, unregulated use of powerful generative models in graded courses introduces serious risks [13,14]. LLMs are probabilistic systems that focus on fluent language and user satisfaction rather than strict correctness [15]. In a pure completion mode, the model often acts like an oracle that simply gives the answer. This removes the need for the student to think through the problem and prevents the formation of robust mental models. In a debugging mode, another problem appears. The model may accept incorrect reasoning because the code passes a particular test case or gives a plausible explanation [16]. This type of hallucination can validate faulty logic. When this happens, the student builds an understanding of programming on weak or false ideas, which may be hard to correct later.

We argue that one main cause of these problems is the typical use of a single, monolithic model that must handle every part of the interaction with the learner. In most current systems, a single model instance receives the student code, tries to solve the problem, decides how to speak to the student, and chooses a teaching strategy [17,18]. Research in cognitive science suggests that complex reasoning works better when fast, intuitive processes and slow, careful checking processes remain distinct [19]. A single model interaction mixes these processes into one step. As a result, the system cannot properly examine its own answer before sending feedback to the student. In addition, the model faces a conflict between two goals. It is asked to be helpful, which often leads to immediate fixes, and at the same time it should support learning, which requires space for student reflection and self correction.

Current practice also ignores an important source of diversity in modern AI systems. Different base models often show different strengths [20]. Some models are strong in code synthesis and program repair [21]. Other models excel in conversation, explanation, and sensitive pedagogical dialogue [22]. Some models are faster and cheaper, which makes them suitable for routine checks [23]. Others are more expensive but more reliable, which makes them suitable for final validation. A single model that plays every role for every student cannot take advantage of this variety. Educational systems that rely on only one model miss the opportunity to assign each task to the type of model that can perform it best.

To address these limitations, we propose a new multi agent framework named The Code Council that explicitly supports orchestration of several models as well as several roles. Instead of depending on one opaque model, our approach distributes the work across specialized agents designed for educational debugging. An Architect agent focuses on the technical solution and produces a reference answer that serves as ground truth. A Skeptic agent then reviews this solution and searches for bugs, unsafe patterns, or hidden weaknesses. After this internal check, a Pedagogue agent compares the student code with the verified solution and plans an instructional strategy that respects the student's current level and errors. Finally, a Mentor agent turns this strategy into a supportive explanation in natural language. Each agent can run on a different underlying model. For example, a code-oriented model can drive the Architect or Skeptic role, while a dialogue-oriented model can drive the Mentor role. The framework also allows rotation of models across roles, so that we can study which assignment gives the most reliable and educationally useful behavior.

This design treats the AI tutor as a flexible council of specialists rather than a single voice. It introduces an internal review step before the system talks to the learner and mimics a slow, deliberate style of reasoning inside the pipeline. It also enables systematic experimentation. We can hold the structure of the council fixed and vary the models that occupy each role. In this way, we can test each candidate model as Architect, as Skeptic, as Pedagogue, and as Mentor, and observe where it adds the most value. This makes it possible to move from model-centric evaluation that focuses on benchmark scores to system-centric evaluation that focuses on stability, safety, and learning outcomes.

This paper contributes to research in automated support for computer science education in several ways.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

- First, we present a formal architecture for multi agent orchestration that enforces a student first workflow. In this workflow, the system always starts from the student's own attempt and focuses on debugging and scaffolding rather than full solution generation.
- Second, we conduct an ablation study that compares this architecture with standard single model baselines. We focus in particular on false positive validations in which an AI tutor incorrectly accepts faulty code.
- Third, we introduce a multi model evaluation protocol. In this protocol, different base models occupy different roles in The Code Council, and we measure how these assignments affect correctness, stability, and quality of feedback.
- Fourth, we design experiments that evaluate the framework in two modes, a Blind mode in which agents solve problems without external ground truth and a Guided mode in which they can access a reference solution.

Our empirical results show that clear role separation and careful model assignment encourage self correction inside the AI system. This leads to more reliable feedback, reduces the risk of reinforcing misconceptions, and increases the safety and robustness of LLM based coding tutors in real educational settings.

In the remainder of this paper, Section 2 reviews prior work on LLM-supported programming education, tutoring interfaces, and program repair, and clarifies how our approach differs from single-model educational modes and existing multi-role tutors. Section 3 describes the Code Council architecture, the role design (including sanitization and adversarial verification), and the experimental protocol. Section 4 presents the evaluation framework, including the experimental protocol, execution-based metrics, and stress-test regimes. Section 5 reports the empirical findings across execution-based repair outcomes, hint quality, solution leakage, and robustness under guided and trap conditions. Section 6 discusses implications for safe and effective programming education and outlines limitations and future directions. Finally, Section 8 concludes the paper.

2. Related Work

2.1. Large Language Models in Education

The rapid deployment of LLMs across educational settings has motivated a growing body of work on their opportunities and risks. Shahzad et al. [24] provide a broad review of LLM use in learning environments, outlining model types, training paradigms, and application scenarios in both digital and higher education. Their theoretical framework emphasises personalisation, ethical concerns, and adaptability as core design challenges when integrating LLMs into teaching–learning processes. While their focus is system-level and cross-domain, our work can be viewed as a concrete instantiation of these concerns in a tightly scoped scenario: code debugging support with explicit attention to safety, leakage, and robustness.

Within programming education specifically, Suzuki et al. conduct a systematic review of LLM applications in programming instruction [25]. Synthesising 25 empirical studies from 2022–2025, they report that LLMs are predominantly used in university-level introductory courses, with tutoring and explanation as the main use cases. Benefits include improved short-term performance and reduced instructor workload, but challenges related to overreliance, equity, and academic integrity remain unresolved. Our study builds directly on these observations by evaluating not only whether LLMs can repair code, but also *how* different orchestration strategies affect scaffolding style, solution leakage, and robustness to adversarial conditions.

Another line of work examines LLM tutors in non-programming domains. Vargas et al. evaluate an LLM-based English tutor for Brazilian learners, focusing on content adaptation, teacher moderation, and interactive corrective feedback [26]. Their findings underline the importance of human oversight and moderation when deploying generative AI as a tutor. We take a complementary perspective: rather than positioning humans in the loop, we investigate whether internal multi-agent coordination

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

(Architect, Skeptic, Secretary, Pedagogue, Mentor) can act as a built-in moderation layer for debugging tutors.

2.2. LLM Use and Student Learning Outcomes in Programming Education

A number of recent empirical studies investigate how students actually use general-purpose LLMs and what impact this has on learning. Jošt et al. analyse informal use of LLMs like ChatGPT and Copilot in an undergraduate React course [27]. They find a significant negative correlation between heavy LLM reliance for code generation and debugging and final grades, whereas using LLMs for explanations shows a weaker negative association. Lai and Lin similarly study an LLM-powered intelligent tutoring system for coding (ITS-CAL) and show that students who use the system in moderation achieve the highest pass rates, whereas overuse of hints is associated with weaker learning gains [28]. Both studies highlight a central tension: LLMs can support learning, but unregulated or excessively directive help risks undermining independent problem-solving.

Our results resonate with these findings at the system-design level. We do not measure long-term learning outcomes directly, but we explicitly quantify solution leakage and scaffolding style (full solutions, heavy hints, conceptual guidance). The council configurations that perform best technically also reduce the proportion of full-solution hints and increase conceptual guidance. In this sense, our work provides a low-level mechanism design perspective on the phenomena reported by Jošt et al. and Lai and Lin: by shaping internal roles and information flow, we can encourage behaviours that are more compatible with fostering student agency.

Cowan et al. propose a framework that “sits between” students and an LLM, rewriting prompts to improve the value of responses in a flipped interaction setting [29]. Their results show that such mediation can increase the pedagogical usefulness of responses compared to direct querying. Mueller et al. present an LLM-based programming tutor with a structured interface and history-based, focused feedback aligned with Hattie’s feedback model, finding that context-rich feedback is rated as more useful than feedback based on the current state alone [30]. Our work shares this concern with interaction design and context management but pushes in a different direction: rather than mediating student prompts or UI, we structure the *internal* interaction among multiple LLM roles and evaluate how this affects patch quality, hint localisation, and leakage.

Lee and Joe introduce a GPT-4o-based code review system for primary and secondary programming education [18]. Their system combines test-based checking and LLM assessment, controls when reviews are necessary, and explicitly aims to prevent AI-assisted cheating by avoiding direct answer revelation. They report improved error detection and semantically aligned feedback relative to existing tools. This work is close in spirit to ours in that it combines execution-based checks with LLM feedback and emphasises cheating prevention. However, their architecture remains essentially single-model with handcrafted logic around it, whereas we study multi-LLM councils, trap resistance, and guided mode as systematic instrumentations of internal disagreement and information flow.

2.3. LLMs for Programming Assistance and Program Repair

Beyond education, a substantial literature investigates LLMs for programming assistance, code generation, and program repair. Bucaioni et al. evaluate ChatGPT’s ability to solve programming problems in C++ and Java [31]. They find that ChatGPT performs well on easy and medium tasks but struggles with more complex problems and generates code that is often less efficient in runtime and memory than human-written solutions. Jiang et al. compare DeepSeek, ChatGPT, and Claude on scientific computing and scientific machine learning (ML) tasks, showing that reasoning-optimised models outperform non-reasoning variants on challenging problems, with differences in both accuracy and reasoning speed [32]. These studies position LLMs as powerful but fallible problem solvers whose performance depends heavily on task difficulty and model design.

Closer to our technical focus, Li et al. propose IDECoder, a framework that integrates IDE-derived static context with LLMs to improve repository-level code completion [33]. By leveraging accurate, real-time cross-file information from IDEs, IDECoder mitigates context-length limitations and improves

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026
doi:10.20944/preprints202603.0350.v1

multi-file completion accuracy. While their aim is not tutoring, their work underscores the importance of structured external context (IDE signals) for enhancing LLM-based coding tools. In contrast, our councils rely on structured *internal* context and role separation rather than IDE integration, but both approaches highlight that naïve "single prompt + single LLM" setups are suboptimal for complex coding tasks.

Zubair et al. provide a systematic literature review of LLM-based program repair [34]. They show that encoder–decoder architectures and open-access datasets dominate current approaches, and they catalogue fine-tuning strategies such as curriculum learning, iterative repair, and knowledge-intensified methods. Evaluation is typically phrased in terms of accuracy, exact match, or BLEU scores. Our work differs in two key ways. First, we focus on *tutoring* rather than purely automated repair, so we evaluate not just patch correctness but also hint localisation, leakage, scaffolding labels, and expert ratings. Second, we introduce multi-agent councils and trap experiments as new mechanisms and evaluation probes that complement the metrics emphasised in the program repair literature.

2.4. Positioning and Comparative Analysis with State-of-the-Art Educational AI Systems

Existing research has established that (i) LLMs can provide effective short-term support in programming education but raise risks of overreliance and academic integrity concerns [24,25,27,28], (ii) interface- and prompt-level mediation can substantially improve feedback quality [18,29,30], and (iii) LLMs are powerful tools for code generation and program repair but remain sensitive to context, model design, and evaluation protocols [31–34]. Despite this progress, much of the literature continues to treat an LLM debugging tutor as a monolithic assistant whose solution-generation capability and pedagogical behavior are governed primarily through prompting or runtime policies. This creates a persistent tension between helpfulness and scaffolding, particularly under correctness pressure, where the same model must both arrive at a correct fix and decide how much of that fix to reveal.

To situate our contribution, we compare *The Code Council* with representative educationally oriented AI systems and recent research prototypes reported in 2025 and early 2026. The key contrast is not the presence of an "educational mode" per se, but the *mechanism used to enforce scaffolding* while preserving technical validity. As summarized in Table 1, prior approaches commonly fall into four lines of work: (a) frontier LLM "study/learning" modes that enforce tutoring behavior through inference-time policies in a single sequential model, (b) education-tuned LLMs whose parameters are aligned toward pedagogical objectives while retaining a single-model architecture, (c) tool-augmented tutoring/debugging agents that rely on external compilation or tests for verification, and (d) multi-role tutor/evaluator/coach systems that separate responsibilities but often lack explicit information-flow constraints that prevent near-solution content from reaching the learner.

Table 1. Comparison of The Code Council with representative educational AI systems and research prototypes (2025–early 2026).

System / Line of Work	**Year**	**Primary architecture**	**Pedagogical safeguard**	**Robustness / resilience mechanism**	**Domain**
Frontier LLM "study/learning" modes [35–38]	2025	Single model (sequential dialogue)	Socratic prompting; hint-style policies	Policy compliance and self-restraint within one model	General / Code
Education-tuned LLMs [39,40]	2025	Single model (fine-tuned)	Instruction tuned for pedagogy	Training-time alignment; long-context grounding	General
Tool-augmented tutoring and debugging agents [41]	2025	Agent + tools	Stepwise tool calls; compiler/test feedback	External verification via compilation/tests	Programming
Multi-role tutor/evaluator/coach prototypes [42]	2026	Multi-agent (role separation)	Human-in-the-loop or evaluator feedback	Role-based feedback loops (often without explicit sanitization)	Programming
The Code Council (Ours)	**2026**	**Multi-agent council with technical loop + pedagogical phase**	**Student-facing roles isolated from internal patches**	**Secretary sanitization; adversarial Skeptic; trap-resistance probes**	**Programming**

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

What this work adds beyond prior systems.

Our contribution bridges the above strands by treating LLM-based debugging tutors as *multi-agent systems* rather than monolithic assistants. The Code Council separates code-facing and student-facing roles and introduces an explicit information-flow boundary. Concretely, a Secretary role sanitizes internal repair artifacts before any pedagogical feedback is produced, which limits the propagation of ground-truth signals and reduces solution leakage risk, especially in Guided settings. In addition, the Council embeds adversarial verification as a first-class pattern: the Architect proposes repairs, the Skeptic attempts to invalidate them, and only critique that survives this loop is converted into constrained revision requests and learner-facing hints.

This architectural decomposition enables three capabilities that are difficult to guarantee in single-model tutoring modes or prompt-mediated assistants. First, it decouples technical correctness from pedagogical delivery, which supports controlled scaffolding without requiring the same model instance to both compute and conceal a complete solution. Second, it provides an internal mechanism for technical rigor via adversarial verification, rather than relying solely on post hoc tool checks or self-consistency. Third, it supports stress testing under adversarial conditions, including Guided-mode pressure and poisoned-reference traps, which reflect realistic classroom failure modes where reference material may be incomplete or incorrect [43]. Our evaluation later quantifies these effects through execution-based outcomes, hint quality analysis, solution-leakage measurement, and adversarial trap scenarios within a role-structured, multi-LLM tutoring framework.

3. Methodology

The goal of this study is to compare different orchestration strategies for LLMs acting as automated debugging tutors. We contrast single-model tutors, where one LLM both repairs code and explains the bug, with multi-agent councils that separate code-focused and pedagogical roles. All experiments are conducted in an offline setting on pre-collected debugging cases. From a student's perspective, each tutor configuration provides exactly one response (one patch and one hint) for a given submission. We first describe the agent architecture and tutor configurations. We then present the operating modes and the overall experimental procedure.

3.1. Agent-Based Tutoring Architecture

We adopt a multi-agent architecture in which different agents assume different roles in the debugging process. All agents communicate through text. The student's original submission is never edited in place. Instead, the system generates internal patches and then produces a hint based on those patches. Figure 1 illustrates the complete workflow of the Code Council.

The architecture uses five roles:

- **Architect**. Receives the problem statement and the buggy program and proposes a candidate patch. The Architect focuses on code-level reasoning and aims for minimal changes that fix the bug.
- **Skeptic**. Receives the problem, the buggy program and the Architect's patch and acts as an adversarial validator. The Skeptic checks logical correctness and edge cases and either accepts the patch or rejects it with a critique.
- **Secretary**. Maintains the internal state of the tutoring process. The Secretary routes messages between agents and turns the Skeptic's critiques into concrete revision requests for the Architect when needed.
- **Pedagogue**. Receives the buggy program and a verified internal patch and performs a gap analysis. The Pedagogue identifies the main misconceptions and designs a teaching plan without exposing a full corrected program.
- **Mentor**. Receives the teaching plan and turns it into a student-facing hint. The Mentor controls tone, length and phrasing and follows a scaffolding policy that encourages the student to reason about the bug.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

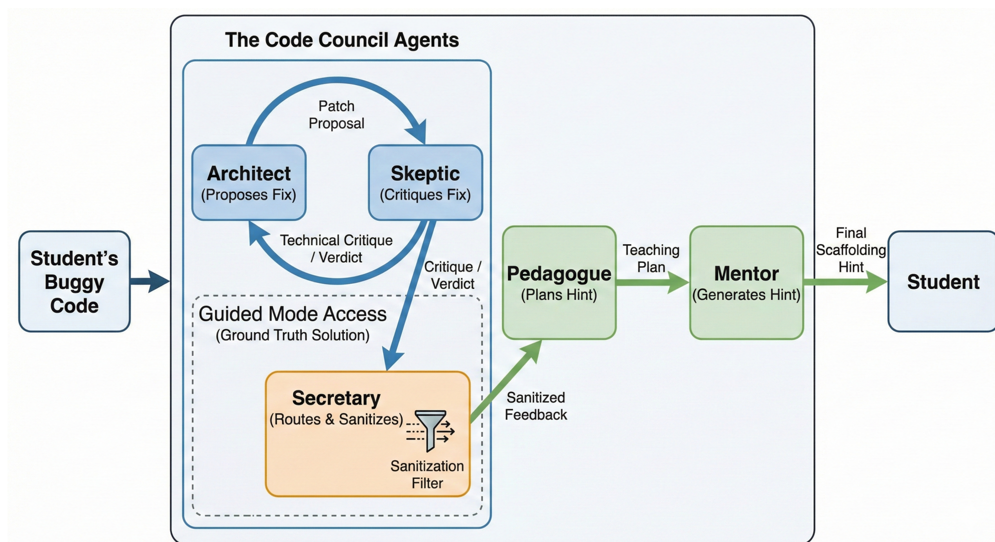

Figure 1. The Code Council Architecture. The system operates in two phases: (1) A *Technical Verification Loop* (blue) where the Architect and Skeptic negotiate a valid repair, and (2) A *Pedagogical Delivery Phase* (green) where the Pedagogue and Mentor craft the final hint. Crucially, the **Secretary** (orange) acts as a sanitization layer, filtering internal verification signals to prevent solution leakage, especially when the system operates in Guided Mode (dashed components).

In council configurations, each role can use a different base model. Code-oriented models are placed in Architect and Skeptic roles, while models known for clearer and more cautious explanations are placed in Pedagogue and Mentor roles. The student sees only the final hint from the Mentor; all internal patches and ground-truth signals remain hidden.

3.2. Tutor Configurations

We evaluate two families of tutors: single-model tutors and multi-agent councils. All configurations operate in an offline way on predefined debugging cases. For each case and each configuration, the system produces exactly one internal patch and one hint.

Single-model tutors

A single-model tutor uses one LLM for both code repair and hint generation and follows a fixed two-step, strictly one-shot protocol:

1. The model receives the problem text and the buggy program and generates a single patched program. We do not allow retries, self-reflection loops, majority voting, or any extra calls for that case. Patch success for single models therefore reflects first-shot repair ability.
2. The same model then receives the problem, the buggy code and its own patch and generates one student-facing hint. The hint prompt asks the model to focus on the main bug, avoid full corrected programs, and adopt a scaffolding style.

The prompting protocol and evaluation pipeline are identical across single-model tutors; only the underlying model changes. In the experiments we instantiate this protocol with four base models that differ in coding strength and explanation style (for example GPT-4o, Gemini 2.0, Claude 4.5, DeepSeek v3.2). Their observed behaviour in this one-shot setting later motivates the design of council configurations.

Multi-agent councils

A council follows the five-role architecture described above. From the student's point of view, the council still produces a single patch and a single hint per case. Internally, however, several agents may interact. For each debugging case the council proceeds as follows:

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

doi:10.20944/preprints202603.0350.v1

1. The Architect receives the problem and the buggy code and proposes a candidate patch.
2. The Skeptic receives the problem, the buggy code and the candidate patch, checks the logic and either accepts the patch as provisionally correct or rejects it with a critique.
3. If the Skeptic rejects the patch, the Secretary turns the critique into a new, more precise request for the Architect. The Architect then proposes a revised patch. This Architect–Skeptic loop can repeat.
4. We give the Architect and the Skeptic up to five attempts to reach agreement. One attempt is a full cycle in which the Architect proposes a patch and the Skeptic evaluates it. If, after five attempts, the Skeptic still rejects all candidate patches, the council marks the case as no agreed patch. For technical metrics this counts as patch failure. In that situation, the Pedagogue and Mentor still produce a cautious, high-level hint that encourages rethinking the algorithm rather than specific line-by-line edits.
5. As soon as the Skeptic accepts a patch within the five-attempt limit, the Pedagogue receives the buggy code and the verified patch and designs a teaching plan that focuses on the main misconception.
6. The Mentor transforms this teaching plan into a short, friendly hint for the student, following scaffolding constraints.

The Architect–Skeptic loop is the only place where repetition occurs. Students never reprompt the tutor in our experiments. Every configuration, whether single-model or council, yields exactly one patch and one hint per debugging case.

3.3. Operating Modes: Blind and Guided

We evaluate each configuration in two operating modes. In the Blind mode, the tutor receives, for each case, the problem statement, the buggy program, and additional information such as a track label (for example course versus challenge). The tutor must infer the bug and propose a repair from these inputs alone. Neither single-model tutors nor councils see the Accepted solution in Blind mode.

In the Guided mode, the system has access to a known-correct solution as ground truth for internal checks only. For single-model tutors, the ground truth is used in an internal evaluation step that compares the candidate patch with the reference, but it is never included in the hint prompt. For councils, only the Skeptic and internal checking routines see the Accepted code. The Architect, Pedagogue and Mentor remain blind to it. Guided mode allows stricter internal validation while student-facing messages still follow a scaffolding policy.

3.4. Experimental Procedure

The overall experimental procedure is fully offline and identical for all configurations:

1. We construct a debugging case from archival data, consisting of the problem text, the buggy program and the external verdict. A corresponding correct reference solution is stored separately and used only for internal execution checks and Guided-mode comparisons.
2. For each tutor configuration we run one complete debugging episode in Blind mode and one in Guided mode.
   - For a single-model tutor, the base model produces one patch in a single call (one-shot) and then one hint in a second call.
   - For a council, the Architect and Skeptic may revise the patch through the internal loop managed by the Secretary, with at most five Architect–Skeptic attempts per case. If the Skeptic accepts a patch, it becomes the council's internal repair. If no agreement is reached within five attempts, the case is marked as technical failure but a conservative hint is still generated.
3. We record the internal patch and measure its similarity to the buggy code using a normalised token-level edit distance. We also run the patch on a test harness compatible with the reference judge and record whether it passes all official tests.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026

doi:10.20944/preprints202603.0350.v1

4. We store the student-facing hint and apply automatic analyses for localization, solution leakage, hint form and scaffolding level as defined in Section 4.

To assess stability, we repeat the debugging procedure three times for a stratified subset of 60 debugging cases, allowing the model's sampling to vary across runs. These additional runs are used only for stability metrics. Unless explicitly stated, all reported scores (patch success, leakage, localisation and related metrics) are computed on a single reference run, and each debugging case contributes exactly one patch and one hint per configuration.

For human evaluation, we use a small, stratified subset of debugging cases to keep expert workload manageable. We sample 20 debugging cases in total, one per problem. For each sampled case and each tutor configuration, two experienced instructors rate one hint, and their scores are averaged for analysis.

4. Evaluation Framework

The objective of the tutor is not only to fix code but also to help students learn how to debug their own programs. A robust evaluation must therefore consider several dimensions at once: technical repair quality, localisation of the bug, solution leakage, pedagogical scaffolding, robustness to pressure for answers, runtime and resource usage, stability across runs, difficulty sensitivity, and alignment with human judgement.

We adopt an evaluation framework that combines execution-based metrics, code similarity measures, automatic hint analysis, scripted adversarial scenarios, runtime statistics, stability measurements, and expert ratings. The framework is applied uniformly to all single-model tutors and all councils. Figure 2 illustrates the adopted evaluation framework. Throughout this section we describe metrics in a dataset-agnostic way. In the Results section we instantiate them on a specific benchmark.

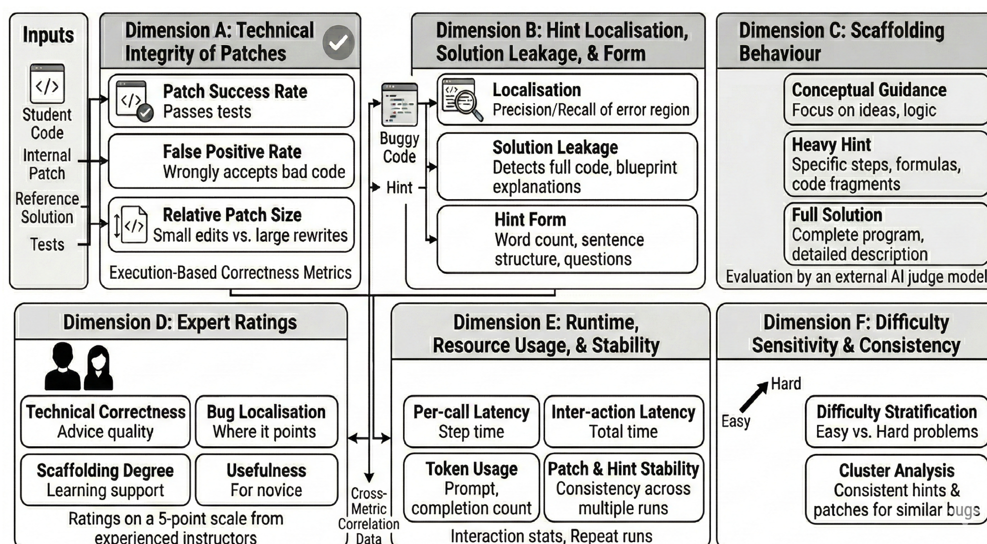

Figure 2. The Six-Dimensional Evaluation Framework. The evaluation combines code-based metrics (Dimension A), automated hint analysis (Dimensions B, C), and operational/human assessments (Dimensions D, E, F) to provide a holistic view of tutor performance.

4.1. Dimension A: Technical Integrity of Patches

The first dimension concerns the technical quality of internal repairs. For each debugging case, we run three program versions on an official test suite or equivalent harness: the original student submission, the internal patch produced by the tutor, and a known-correct reference program. We use the following core metrics.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

Patch success rate

The proportion of cases in which the internal patch compiles and passes all official tests, while the original student program fails at least one test. This measures how often the internal repair would actually solve the problem if the student copied it.

False positive rate

The proportion of cases in which the external judge still rejects the student program, but the tutor implicitly treats it as correct. Operationally, we detect this when the hint praises the program or states that no change is needed, although the submission still fails the test harness.

Relative patch size

A normalised, token-level edit distance between the buggy code and the internal patch. It measures the fraction of tokens that differ between the two versions. Smaller values indicate local edits that preserve most of the student's work. Larger values indicate extensive rewrites approaching full solution replacement.

These metrics quantify what the tutor does to the code, independently of how well it explains the bug.

4.2. Dimension B: Hint Localisation, Solution Leakage, and Form

The second dimension focuses on student-facing hints in Blind mode.

Localisation

For each case, we treat the token-level diff between buggy and reference code as a proxy for the true error region. We align mentions in the hint (line numbers, variables, control structures) with code spans and compute:

- Localisation precision: the proportion of locations mentioned in the hint that fall inside the true error region;
- Localisation recall: the proportion of true error locations that the hint mentions.

Higher precision and recall indicate hints that talk about where the bug actually resides.

Solution leakage

A hint leaks the solution not only when it contains a full compilable program, but also when the natural language explanation becomes a blueprint that leaves almost no work to the student. We combine two detectors:

- A code leakage detector that flags complete programs, long corrected code blocks, and line-by-line rewritten versions of the student's program;
- A text leakage detector that flags explanations that describe the algorithm step-by-step in a way that can be translated into code almost mechanically.

For each configuration we report the percentage of hints that leak a solution in either way and also the median number of code tokens in hints.

Hint form

We characterise hints by their length and structure, measuring median word count, median sentence count, and the median number of interrogative sentences. Interrogative sentences serve as a proxy for Socratic guidance: hints that ask at least one question and remain relatively compact are closer to the intended scaffolding style.

4.3. Dimension C: Scaffolding Behaviour from a Judge Model

String-based metrics do not fully capture pedagogical quality. We therefore use a strong external judge model as an educational auditor. For each hint in Blind mode, the judge sees the problem text,

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

the buggy code, and the hint, but not the internal patch or the reference solution. The judge assigns one of three labels:

- Full solution: the hint exposes a complete program, a line-by-line corrected version of the student code, or a step-by-step natural language description that reconstructs the algorithm with almost no additional reasoning from the student;
- Heavy hint: the hint does not fully reveal the solution but gives specific steps, formulas or code fragments that leave only routine translation work;
- Conceptual guidance: the hint focuses on ideas, input/output conditions, typical edge cases or high-level control flow, and leaves construction of the exact code to the student.

For each configuration we report the percentage of hints in each category. A higher share of conceptual guidance and a lower share of full solutions indicate better scaffolding. Because the judge model looks at both code and text, it detects cases where the system gives away the answer through explanation even when little code is shown.

4.4. Dimension D: Expert Ratings of Hint Quality

Automatic metrics and judge labels do not replace human judgement. To ground the analysis in classroom practice we ask two experienced instructors to rate a stratified sample of hints from Blind mode. For each of these sampled cases, the experts see the problem statement, the buggy code, the external verdict and one hint. They do not know which system produced the hint. For each hint the experts assign scores on a five-point scale for four criteria including technical correctness of the advice, localisation of the bug, degree of scaffolding instead of solution giving, and usefulness for a typical novice. We report mean scores per configuration and examine correlations between expert scores and automatic metrics such as localisation, leakage and judge labels.

4.5. Dimension E: Runtime, Resource Usage, and Stability

Tutors must operate within reasonable time and cost budgets. For each configuration we record:

- Per-call latency for internal steps (Architect, Skeptic, Pedagogue, Mentor);
- Per-interaction latency from the initial request to the final hint;
- The number of prompt and completion tokens per interaction (median and 90th percentile).

These metrics give a practical view of the extra cost introduced by multi-agent orchestration.

To evaluate stability, we repeat the debugging procedure multiple times for a stratified subset of cases and then compute:

- Patch stability: the fraction of debugging cases in this subset where all internal patches are identical across runs;
- Hint stability: the average semantic similarity (for example, cosine similarity of embeddings) between hints for the same case across runs.

All other reported metrics in this paper (patch success, localisation, leakage, and so on) are computed on a single reference run over the full benchmark; the repeated runs are used only for stability analysis.

4.6. Dimension F: Difficulty Sensitivity and Consistency

If the benchmark contains problems at different difficulty levels (for example, introductory course exercises versus contest-style challenges), we stratify metrics by track and report separate scores for easy and hard subsets. This reveals whether councils help mainly on harder problems while single-model tutors already suffice for basics. For selected problems we may also cluster similar buggy submissions (for example, same bug pattern) and analyse consistency:

- Hint consistency: whether students in the same cluster receive similar advice;
- Patch pattern consistency: whether internal patches in the cluster address the same root cause.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

Higher consistency suggests that the tutor behaves less like a random oracle and more like a stable teaching assistant.

5. Results

This section reports the results of the offline simulation on a benchmark of debugging cases. We first describe the dataset and tutor configurations. We then present results for single–model tutors, covering technical integrity, hint quality, scaffolding labels, runtime, stability, and expert ratings. Next we introduce multi–agent councils, explain how the single–model results motivated their design, and report council performance. We close with an analysis of Guided mode and a summary of the main findings. Unless stated otherwise, all metrics refer to Blind mode and to a single reference run. Stability metrics use three offline reruns on a stratified subset of cases, and expert ratings are based on a sampled subset of debugging cases.

5.1. Dataset and Tracks

We build a benchmark from archival submissions to an online judge. Each debugging case consists of a problem statement, one wrong submission, and a later Accepted submission by the same student on the same problem. All programs are written in the same programming language, which removes language–related confounds. We select 20 problems in total and split them into two tracks:

- **Course track (10 problems).** Problems that resemble exercises in an introductory programming course: basic input/output, arithmetic expressions, conditionals, loops, strings, and simple arrays. The set of foundational course-track problems selected for evaluation is presented in Table 2.

Table 2. Foundational course problems (ITP series) used in the evaluation.

Problem ID	Problem Name	Link
ITP1_4_B	Circle	https://onlinejudge.u-aizu.ac.jp/courses/lesson/2/ITP1/4/ITP1_4_B
ITP1_5_C	Print a Chessboard	https://onlinejudge.u-aizu.ac.jp/courses/lesson/2/ITP1/5/ITP1_5_C
ITP1_5_D	Structured Programming	https://onlinejudge.u-aizu.ac.jp/courses/lesson/2/ITP1/5/ITP1_5_D
ITP1_6_C	Official House	https://onlinejudge.u-aizu.ac.jp/courses/lesson/2/ITP1/6/ITP1_6_C
ITP1_7_C	Spreadsheet	https://onlinejudge.u-aizu.ac.jp/courses/lesson/2/ITP1/7/ITP1_7_C
ITP1_8_D	Ring	https://onlinejudge.u-aizu.ac.jp/courses/lesson/2/ITP1/8/ITP1_8_D
ITP1_9_D	Transformation	https://onlinejudge.u-aizu.ac.jp/courses/lesson/2/ITP1/9/ITP1_9_D
ITP1_10_D	Distance II	https://onlinejudge.u-aizu.ac.jp/courses/lesson/2/ITP1/10/ITP1_10_D
ITP1_11_C	Dice III	https://onlinejudge.u-aizu.ac.jp/courses/lesson/2/ITP1/11/ITP1_11_C
ITP1_11_D	Dice IV	https://onlinejudge.u-aizu.ac.jp/courses/lesson/2/ITP1/11/ITP1_11_D

- **Challenge track (10 problems).** Harder problems inspired by International Collegiate Programming Contest (ICPC) tasks, with more complex control flow, data structures, and edge case reasoning (for example weighted union–find, shortest paths, or dynamic programming). Table 3 summarizes the challenge-track problems used in the evaluation.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026
doi:10.20944/preprints202603.0350.v1

Table 3. ICPC regional challenge problems used as high-difficulty evaluation tasks.

Problem ID	Problem Name	Link
1356	Decimal Sequences (ICPC 2015)	https://onlinejudge.u-aizu.ac.jp/challenges/sources/ICPC/Regional/1356?year=2015
1380	Medical Checkup (ICPC 2017)	https://onlinejudge.u-aizu.ac.jp/challenges/sources/ICPC/Regional/1380?year=2017
1406	Ambiguous Encoding (ICPC 2019)	https://onlinejudge.u-aizu.ac.jp/challenges/sources/ICPC/Regional/1406?year=2019
1451	Task Assignment to Two Employees (ICPC 2023)	https://onlinejudge.u-aizu.ac.jp/challenges/sources/ICPC/Regional/1451?year=2023
1420	Formica Sokobanica (ICPC 2020)	https://onlinejudge.u-aizu.ac.jp/challenges/sources/ICPC/Regional/1420?year=2020
1383	Pizza Delivery (ICPC 2017)	https://onlinejudge.u-aizu.ac.jp/challenges/sources/ICPC/Regional/1383?year=2017
1428	Genealogy of Puppets (ICPC 2021)	https://onlinejudge.u-aizu.ac.jp/challenges/sources/ICPC/Regional/1428?year=2021
1433	Hasty Santa Claus (ICPC 2022)	https://onlinejudge.u-aizu.ac.jp/challenges/sources/ICPC/Regional/1433?year=2022
1337	Count the Regions (ICPC 2013)	https://onlinejudge.u-aizu.ac.jp/challenges/sources/ICPC/Regional/1337?year=2013
1449	Color Inversion on a Huge Chessboard (ICPC 2023)	https://onlinejudge.u-aizu.ac.jp/challenges/sources/ICPC/Regional/1449?year=2023

For each problem we sample exactly 10 debugging cases with non–AC verdicts such as Wrong Answer, Time Limit Exceeded, or Runtime Error. The benchmark therefore contains

$$10 \times 10 + 10 \times 10 = 200$$

debugging cases, with 100 course cases and 100 challenge cases. For each case we store the problem text, the buggy submission, its non–AC verdict, the track label and the problem identifier, and a later Accepted submission. The reference solution is used for execution–based checks, for defining the error region, and for internal validation in Guided mode, but it is never shown to student–facing roles in Blind mode.

5.2. Single–Model Tutors: Technical Integrity

We first report results for single–model tutors in Blind mode. Each tutor uses one LLM to generate a single patch and a single hint per debugging case. Table 4 summarises technical metrics aggregated over all 200 cases.

Table 4. Technical integrity of patches in Blind mode for single–model tutors (one–shot patches).

Configuration	Patch Succ. (%)	False Pos. (%)	Med. PatchSize
GPT-4o tutor	87.7	11.3	0.41
Gemini 2.0 tutor	84.8	12.7	0.44
Claude 4.5 tutor	84.2	9.9	0.40
DeepSeek v3.2 tutor	80.7	9.1	0.38

All single–model tutors repair a large fraction of cases in a one–shot setting. GPT-4o achieves the highest overall patch success. Gemini 2.0 and Claude 4.5 follow closely, with Claude 4.5 offering a slightly better trade–off between patch success and false positives. DeepSeek v3.2 attains the smallest median PatchSize, consistent with its strong code–generation capabilities, but lags somewhat behind in overall success. Gemini 2.0 performs slightly worse than GPT-4o on this benchmark, with lower patch success and higher false positives.

These aggregate numbers hide a crucial distinction between course and challenge tracks. Table 5 shows patch success by track.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

Table 5. Patch success by track for single–model tutors in Blind mode (one–shot patches).

Configuration	Course Patch Succ. (%)	Challenge Patch Succ. (%)
GPT-4o tutor	100.0	75.3
Gemini 2.0 tutor	100.0	69.6
Claude 4.5 tutor	100.0	68.4
DeepSeek v3.2 tutor	100.0	61.4

On course problems, all four single–model tutors successfully repair every debugging case in a single shot. Differences appear almost entirely on challenge problems. GPT-4o achieves the highest patch success on challenge tasks (75.3%), followed by Gemini 2.0 (69.6%), Claude 4.5 (68.4%), and DeepSeek v3.2 (61.4%) in this strict one–shot setting.

GPT-4o is therefore the best all–rounder, combining perfect performance on course problems with the strongest challenge–track performance. Gemini 2.0 and Claude 4.5 behave as competent generalists that remain reasonably close on the challenge track. DeepSeek v3.2 remains attractive as a code–centric model with smaller PatchSize, but its one–shot success on hard problems is noticeably lower. All single–model tutors exhibit median PatchSize around 0.4, which means that their one–shot patches often rewrite a substantial portion of the student program rather than performing strictly local edits.

5.3. Single–Model Tutors: Hint Localisation, Leakage, and Scaffolding

Table 6 reports hint localisation and leakage metrics in Blind mode for single–model tutors.

Table 6. Hint localisation, leakage, and form metrics in Blind mode for single–model tutors.

Configuration	Loc. Prec.	Loc. Rec.	Leakage (%)	Code Tokens (med.)	Words (med.)	Questions (med.)
GPT-4o tutor	0.62	0.55	22.0	102	86	0.8
Gemini 2.0 tutor	0.59	0.52	23.5	96	92	0.7
Claude 4.5 tutor	0.64	0.56	18.5	88	95	0.9
DeepSeek v3.2 tutor	0.60	0.52	24.0	115	80	0.6

All single–model tutors show moderate localisation: hints tend to point roughly to the right region but do not always pinpoint the exact lines or variables. Leakage rates remain relatively high. Around one fifth of hints from GPT-4o and Gemini 2.0 expose full or near–full solutions, and DeepSeek v3.2 is even more leak–prone, often including large corrected code blocks. Claude 4.5 is the most conservative single–model tutor, with the lowest leakage and slightly better localisation.

Hints from GPT-4o and Gemini 2.0 are long and narrative. DeepSeek v3.2 produces shorter explanations but embeds more code, leading to the highest median number of code tokens. Claude 4.5 tends to produce slightly longer text with fewer code tokens and more questions, which aligns with a more reflective and cautious tutoring style.

Judge–model scaffolding labels reinforce this picture. Table 7 shows the distribution of hint categories.

Table 7. Scaffolding categories from the judge model in Blind mode for single–model tutors.

Configuration	Full solution	Heavy hint	Conceptual guidance
GPT-4o tutor	20%	39%	41%
Gemini 2.0 tutor	22%	38%	40%
Claude 4.5 tutor	17%	37%	46%
DeepSeek v3.2 tutor	23%	36%	41%

Roughly two fifths of hints from single–model tutors are heavy hints, and between 17% and 23% are full solutions. Claude 4.5 has the lowest share of full solutions and the highest share of conceptual guidance, which matches its lower leakage rate. GPT-4o and DeepSeek v3.2 provide strong technical help but are more prone to revealing too much detail.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

5.4. Single–Model Tutors: Runtime, Stability, and Expert Ratings

Table 8 shows runtime and token usage in Blind mode for single–model tutors.

Table 8. Runtime and token usage in Blind mode for single–model tutors. Latency is measured end to end per debugging case.

Configuration	Latency (med.)	Tokens / interaction (med.)	Tokens / interaction (p90)
GPT-4o tutor	7.4	3,150	3,980
Gemini 2.0 tutor	7.9	3,280	4,120
Claude 4.5 tutor	8.3	3,210	4,050
DeepSeek v3.2 tutor	6.8	2,980	3,760

DeepSeek v3.2 is the fastest single–model tutor and uses the fewest tokens. GPT-4o and Claude 4.5 incur slightly higher latency and token usage but remain within an acceptable range for interactive use. These numbers provide a baseline for the additional cost introduced by councils.

To assess stability, we rerun a stratified subset of 60 debugging cases three times per configuration, allowing the model sampling to vary across runs. Table 9 reports patch stability and hint stability.

Table 9. Stability across three offline runs in Blind mode for single–model tutors (stratified subset of 60 cases).

Configuration	Patch Stab. (% ident.)	Hint Sim. (mean cos.)
GPT-4o tutor	64.0	0.81
Gemini 2.0 tutor	61.5	0.79
Claude 4.5 tutor	66.2	0.83
DeepSeek v3.2 tutor	63.7	0.80

Single–model tutors produce identical internal patches across runs ($\approx$ 62–66%) of sampled cases, with hints exhibiting high but not perfect semantic similarity. Claude 4.5 is slightly more stable than the others.

Human evaluation follows the same sampling scheme across configurations. Two experienced instructors jointly rate hints for a sample of 20 debugging cases (one per problem). For each sampled case and each tutor configuration, both instructors independently rate one hint. Table 10 shows mean scores averaged across raters.

Table 10. Mean expert ratings (five–point scale) on a sample of 20 debugging cases for single–model tutors.

Configuration	Correct.	Loc.	Scaf.	Usef.
GPT-4o tutor	3.8	3.5	3.2	3.4
Gemini 2.0 tutor	3.6	3.3	3.1	3.3
Claude 4.5 tutor	3.9	3.7	3.5	3.7
DeepSeek v3.2 tutor	3.8	3.4	3.1	3.4

The instructors slightly prefer hints from Claude 4.5, particularly on scaffolding and usefulness. GPT-4o and DeepSeek v3.2 are perceived as technically strong but more likely to slip into heavy hints or near–solutions. These ratings align with automatic metrics: higher expert scores correlate with higher localisation, lower leakage, more questions per hint, and a higher share of conceptual guidance labels.

5.5. From Single–Model Results to Council Design

The single–model results suggest the following:

- GPT-4o is the strongest all–rounder, with the best challenge–track patch success and fluent explanations.
- DeepSeek v3.2 is extremely capable at low–level code repair but tends to rewrite more code and leak more solutions when used directly for hints.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

doi:10.20944/preprints202603.0350.v1

- Claude 4.5 produces the most conservative hints, with lower leakage and more conceptual guidance, at a small cost in challenge–track patch success relative to GPT-4o.
- Gemini 2.0 is a solid generalist with good long–context handling but does not dominate technical metrics.

These observations motivate council configurations that specialise models according to their strengths: place DeepSeek v3.2 in code–facing roles (Architect, Skeptic), use GPT-4o or Gemini 2.0 for orchestration, and assign Claude 4.5 and/or GPT-4o to pedagogical roles (Pedagogue, Mentor). Councils can then leverage DeepSeek v3.2's code strength while mitigating its leakage, and exploit Claude 4.5's cautious explanation style without relying on it for all code repair.

5.6. Council Configurations

We evaluate four councils:

- **Council–GPT**. GPT-4o plays all five roles. This isolates the effect of role separation and internal negotiation, since the base model matches the GPT-4o single–model tutor.
- **Council–CodeFirst**. DeepSeek v3.2 acts as Architect and Skeptic, Gemini 2.0 as Secretary, and GPT-4o as both Pedagogue and Mentor. DeepSeek v3.2 focuses on code repair and validation, Gemini 2.0 handles routing and state, and GPT-4o crafts and delivers hints.
- **Council–Conservative**. GPT-4o acts as Architect, DeepSeek v3.2 as Skeptic, Gemini 2.0 as Secretary, and Claude 4.5 as both Pedagogue and Mentor. Here DeepSeek v3.2 probes code more aggressively, while Claude 4.5 enforces a cautious scaffolding style.
- **Council–HybridStrong**. DeepSeek v3.2 acts as Architect and Skeptic, Gemini 2.0 as Secretary, Claude 4.5 as Pedagogue, and GPT-4o as Mentor. This configuration places the strongest code model on both generation and validation, uses Gemini 2.0 for long–context orchestration, and splits pedagogical responsibilities between Claude 4.5 (design) and GPT-4o (delivery).

All councils operate under the same five–attempt limit for the Architect–Skeptic loop. Cases without agreement within five attempts are treated as patch failures for technical metrics but still receive a conservative, high–level hint.

5.7. Councils: Technical Integrity in Blind Mode

Table 11 shows aggregate technical metrics for councils in Blind mode.

Table 11. Technical integrity of patches in Blind mode for councils.

Configuration	Patch Succ	False Pos.	Med. PatchSize
Council–GPT	89.9	6.5	0.35
Council–CodeFirst	91.9	5.2	0.30
Council–Conservative	92.4	4.6	0.34
Council–HybridStrong	93.8	4.9	0.32

Compared to single–model tutors, all councils maintain or improve overall patch success, with Council–HybridStrong achieving the highest overall success. False positive rates drop substantially for all councils relative to the GPT-4o single–model tutor, but not monotonically across configurations: Council–Conservative has the lowest false positive rate, while Council–HybridStrong trades a slightly higher false positive rate for the best patch success. Council–CodeFirst achieves the smallest median PatchSize, indicating the most local repairs on average, but with a somewhat higher false positive rate than Council–Conservative.

As with single–model tutors, the track structure is important. Table 12 splits patch success by track.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

Table 12. Patch success by track for councils in Blind mode (one–shot per case, up to five internal Architect–Skeptic attempts).

Configuration	Cour. Pat. Succ. (%)	Chal. Pat. Succ. (%)
Council–GPT	100.0	79.8
Council–CodeFirst	100.0	83.8
Council–Conservative	100.0	84.8
Council–HybridStrong	100.0	87.5

On course problems, all councils, like all single–model tutors, reach 100% patch success. On challenge problems, as illustrated in Figure 3, Council–HybridStrong achieves 87.5% patch success, compared to 75.3% for the best single–model tutor (GPT-4o). The gap of 12.2 percentage points on the challenge track illustrates the main technical advantage of the council architecture.

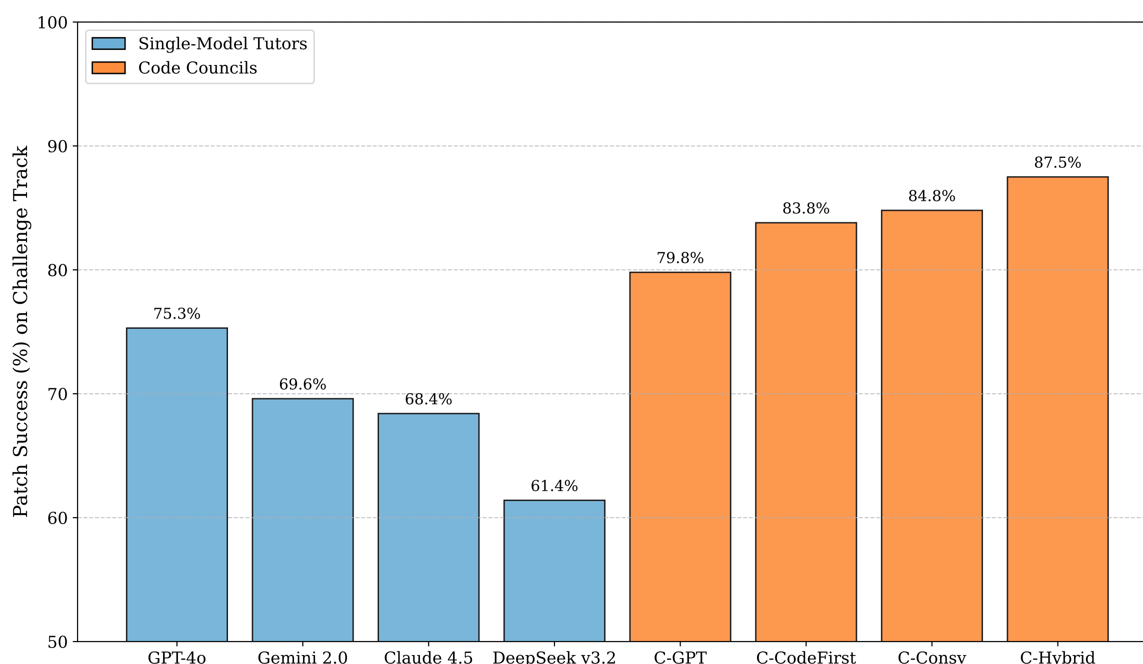

Figure 3. Technical Victory on Hard Problems. While single-model tutors (blue) struggle on the Challenge Track, the Council architectures (orange) consistently achieve higher patch success rates, with the HybridStrong configuration reaching 87.5%.

The Architect–Skeptic negotiation can, however, have side effects on PatchSize. In some hard challenge cases, repeated rejections push the Architect toward more radical rewrites. When the Skeptic insists on correcting subtle edge cases, the fifth–attempt patch may differ from the original submission more than the one–shot patch of a single–model tutor. The gains in Table 11 therefore come primarily from improved correctness and fewer false validations rather than universally smaller edits. Council–CodeFirst, with DeepSeek v3.2 in both code–facing roles but GPT-4o in pedagogical roles, achieves the smallest median PatchSize but not the very best patch success or false positive rate.

5.8. Councils: Hint Quality and Scaffolding in Blind Mode

Figure 4 provides a comprehensive safety profile across all configurations. It reveals a consistent trend: every single-model tutor exhibits solution leakage rates above 18%, whereas all council configurations suppress this risk to below 14% while simultaneously reducing false positives.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

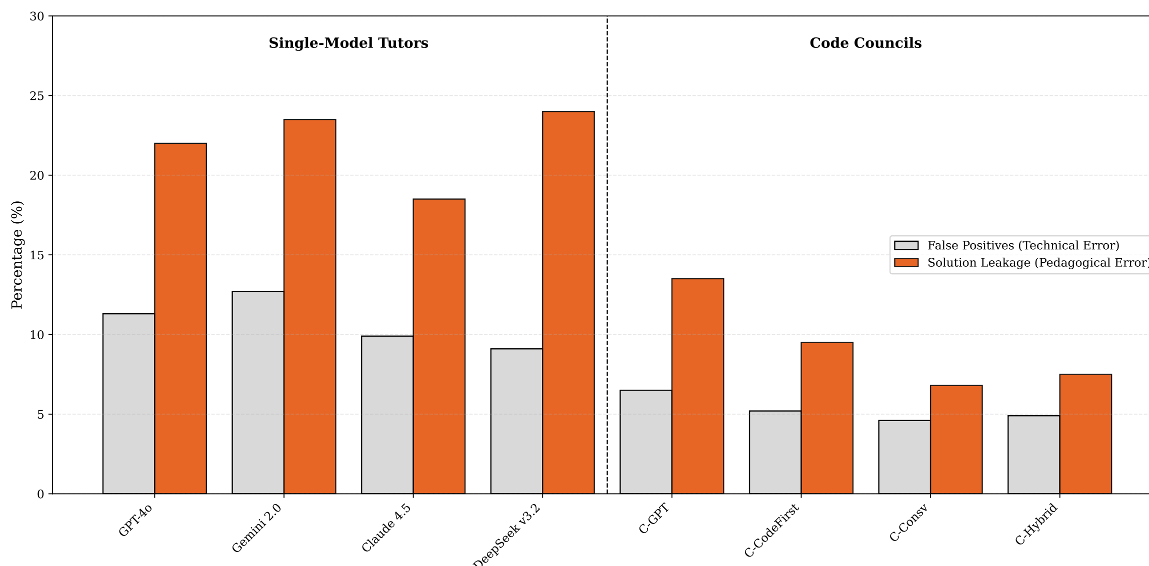

Figure 4. Comprehensive Safety Profile. Comparing all configurations reveals that every single-model tutor (left) exhibits high solution leakage (orange), whereas council configurations (right) suppress this risk while maintaining low false positive rates (gray).

Table 13 shows localisation, leakage, and form metrics for councils in Blind mode.

Table 13. Hint localisation, leakage, and form metrics in Blind mode for councils.

Configuration	Loc. Prec.	Loc. Rec.	Leakage (%)	Code Tokens (med.)	Words (med.)	Questions (med.)
Council–GPT	0.70	0.61	13.5	74	79	1.1
Council–CodeFirst	0.76	0.68	9.5	60	73	1.3
Council–Conservative	0.75	0.67	6.8	54	74	1.5
Council–HybridStrong	0.78	0.69	7.5	52	71	1.4

All councils improve localisation and reduce leakage compared to single–model tutors. Council–GPT already increases localisation precision and recall and reduces leakage relative to the GPT-4o tutor, purely by separating roles and introducing an internal check. Councils that assign DeepSeek v3.2 to internal roles and Claude 4.5 or GPT-4o to pedagogical roles achieve the strongest improvements. Detailed localization metrics for all configurations are shown in Figure 5.

The data confirms that the Council-HybridStrong outperforms even the strongest single-model baseline (Claude 4.5) in both precision and recall. The trade–offs between councils are more nuanced. Council–HybridStrong attains the highest localisation scores (both precision and recall) and relatively low leakage, but Council–Conservative yields the lowest leakage overall and the most question–rich hints, reflecting a particularly cautious scaffolding style. Council–CodeFirst sits between them: it achieves localisation close to HybridStrong, produces short hints with fewer code tokens, but leaks slightly more often than Council–Conservative. Council–GPT provides solid improvement over the single–model GPT-4o baseline at lower orchestration cost than the more complex councils.

Judge–model scaffolding labels reflect these trends. Table 14 shows the distribution of categories.

Table 14. Scaffolding categories from the judge model in Blind mode for councils.

Configuration	Full sol.	Heavy hint	Concept. guid.
Council–GPT	12%	35%	53%
Council–CodeFirst	8%	34%	58%
Council–Conservative	4%	28%	68%
Council–HybridStrong	6%	30%	64%

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

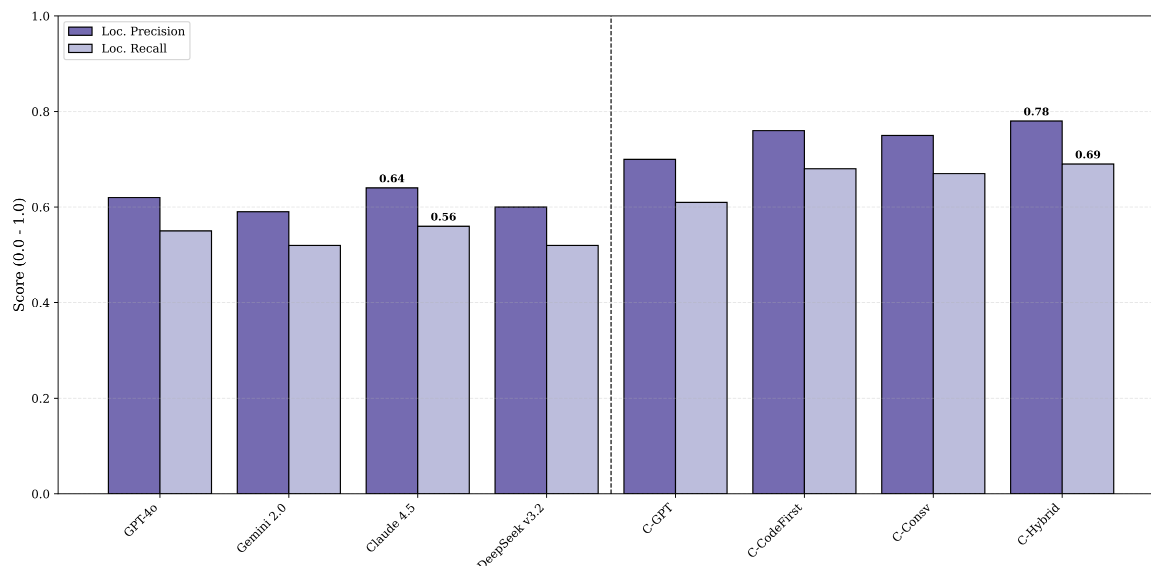

Figure 5. Bug Localization Performance. The Council-HybridStrong architecture demonstrates superior precision and recall in isolating error regions compared to even the strongest single-model baselines.

Compared to single–model tutors, councils shift mass from full solutions and heavy hints toward conceptual guidance. This effect is visualized in Figure 6, which reveals a marked distributional shift from "Full Solutions" in single-model tutors to "Conceptual Guidance" in council configurations. Council–Conservative is the most cautious configuration, with the smallest share of full solutions and the largest share of conceptual guidance. Council–HybridStrong still performs very well on scaffolding but prioritises slightly more direct help in exchange for its stronger technical performance. Council–CodeFirst again offers a middle ground: less conservative than Council–Conservative but more scaffolded than Council–GPT.

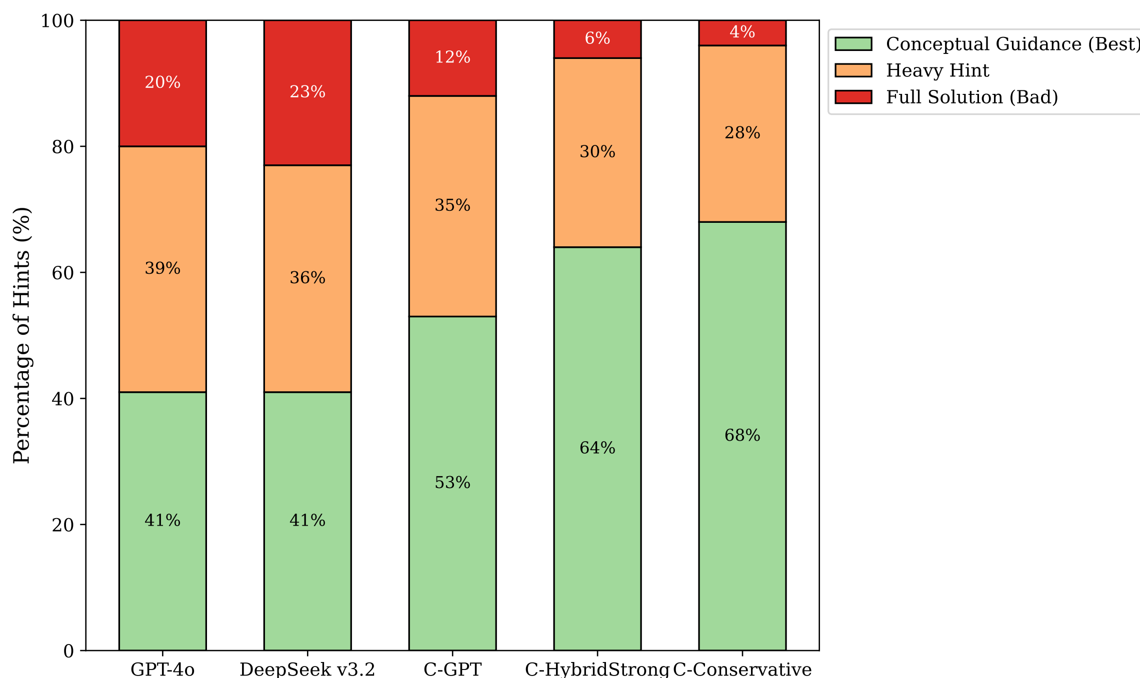

Figure 6. The Pedagogical Shift. While single-model tutors (left) frequently default to giving away full solutions (red), the Code Council (right) successfully shifts the distribution toward conceptual guidance (green), fulfilling the scaffolding objective.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

5.9. Councils: Runtime, Stability, and Expert Ratings

Councils incur additional latency and token usage due to internal coordination. Table 15 reports runtime metrics.

Table 15. Runtime and token usage in Blind mode for councils.

Configuration	Latency (med., s)	Tokens / interaction (med.)	Tokens / interaction (p90)
Council–GPT	11.5	4,500	5,700
Council–CodeFirst	12.6	4,850	6,020
Council–Conservative	13.0	5,050	6,150
Council–HybridStrong	13.4	5,030	6,320

Median latency for councils is roughly 1.5–2 times that of single–model tutors, and token usage increases by about 50–60%. Council–GPT is the cheapest council configuration, while Council–Conservative and Council–HybridStrong are the most expensive. While this architecture incurs higher computational overhead, the cost-benefit analysis in Figure 7 positions the councils in a distinct high-performance tier that justifies the additional token usage. The five–attempt cap on Architect–Skeptic negotiation keeps worst–case latency bounded. In practice, most cases converge before the fifth attempt, so median latency remains below the worst–case upper bound.

Figure 7. Cost-Benefit Analysis. Although Council configurations (diamonds) incur higher token costs than single models (circles), they inhabit a distinct performance tier, justifying the expense with superior patch success rates.

Stability is higher for councils than for single–model tutors. Using the same 60–case stratified subset as for single–model stability, Table 16 summarises council stability metrics. Patch stability increases by about 7–15 percentage points compared to single–model tutors, and hints become more consistent. Council–Conservative is the most stable configuration, with Council–HybridStrong close behind. Council–CodeFirst also improves stability relative to Council–GPT, at a moderate additional cost.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

Table 16. Stability across three offline runs in Blind mode for councils (stratified subset of 60 cases).

Configuration	Pat. Stab. (% ident.)	Hint Sim. (mean cos.)
Council–GPT	71.6	0.84
Council–CodeFirst	74.9	0.86
Council–Conservative	78.2	0.88
Council–HybridStrong	77.0	0.87

The same two instructors who rated single–model hints also rate council hints on the 20–case sample. Table 17 shows mean scores averaged across raters. The instructors consistently prefer council hints over single–model hints, especially for scaffolding and usefulness. Council–HybridStrong receives the highest correctness scores, reflecting its stronger technical performance, while Council–Conservative is slightly preferred on scaffolding and overall usefulness. Council–CodeFirst again lies between these two extremes, offering hints that are technically strong, reasonably scaffolded, and less costly than those of the more conservative councils.

Table 17. Mean expert ratings (five–point scale) on a sample of 20 debugging cases for councils.

Configuration	Correct.	Local.	Scaf.	Usef.
Council–GPT	4.1	3.9	3.8	3.9
Council–CodeFirst	4.2	4.0	4.0	4.1
Council–Conservative	4.2	4.1	4.3	4.4
Council–HybridStrong	4.4	4.2	4.2	4.3

5.10. Guided Mode and Prompt Revision

Guided mode gives internal checking components access to the reference solution. In principle, this should help internal validators reject weak patches more reliably. In practice, however, our initial Guided–mode experiments revealed a serious leakage failure for councils: when the Skeptic repeatedly rejected candidate patches on hard challenge problems and had access to the ground truth, its detailed critiques sometimes exposed key algorithmic ideas, and the Pedagogue propagated them into student–facing hints. In the original Guided–mode setting:

- Only the Skeptic and internal checking routines saw the reference solution; Architect, Pedagogue, and Mentor were intended to remain blind;
- The Secretary simply routed messages, passing the Skeptic's critiques verbatim to other roles;
- Prompts for Pedagogue and Mentor mentioned non–leakage but did not explicitly constrain how ground–truth–based feedback should be used.

When the Architect proposed an incorrect patch, the Skeptic compared it against the ground truth and rejected it. To explain the rejection, it often produced detailed feedback such as "the reference solution uses a dynamic programming table of size $N + 1$, but this patch uses a greedy approach". The Secretary forwarded this critique directly. The Pedagogue, now implicitly aware of the hidden algorithm, incorporated these details into its teaching plan, and the Mentor surfaced them to the student. In early Guided–mode runs with this naive setup, councils showed substantially higher leakage than in Blind mode, especially on a subset of hard challenge cases where the Architect–Skeptic loop exhausted all five attempts. To address this, we revised Guided–mode prompts and routing in three ways:

- we ensured that both Architect and Pedagogue remain blind to the reference solution throughout, relying only on the buggy code, the problem statement, and abstract feedback from the Skeptic;
- we upgraded the Secretary into a sanitiser in Guided mode: when forwarding Skeptic critiques, it removes specific code snippets, identifiers, or algorithmic blueprints, preserving only high–level error descriptions (for example, "off–by–one in loop bounds" rather than "use a DP table of size $N + 1$");

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

- we tightened Pedagogue and Mentor prompts to emphasise that internal correctness signals must be treated as yes/no judgements on candidate reasoning, not as a source of code or step–by–step solutions to be quoted or paraphrased.

Table 18 summarises Blind–mode performance and Guided–mode technical metrics after these revisions for all councils. Guided mode with revised prompts slightly improves patch success and reduces false positives for all councils. Council–HybridStrong benefits the most in absolute patch success, while Council–Conservative achieves the lowest false positive rate. The trade–offs observed in Blind mode largely persist: Council–HybridStrong leads on technical success, Council–Conservative and Council–CodeFirst offer slightly more cautious behaviour with fewer false positives, and Council–GPT remains the lightweight baseline.

Table 18. Blind vs. Guided mode for councils (technical metrics after prompt and routing revision).

	Patch Succ. (%)		False Pos. (%)	
Configuration	Blind	Guid. (rev.)	Blind	Guid. (rev.)
Council–GPT	89.9	90.8	6.5	5.0
Council–CodeFirst	91.9	92.6	5.2	4.2
Council–Conservative	92.4	93.3	4.6	4.1
Council–HybridStrong	93.8	94.6	4.9	3.7

The main effect of Guided mode is on solution leakage. Table 19 reports leakage rates in Blind mode, in the initial Guided–mode experiment with naive routing, and in Guided mode after introducing the Secretary sanitiser and tightening prompts. In all cases, the naive Guided–mode setup increases leakage substantially relative to Blind mode, especially for councils that rely heavily on the Skeptic's feedback on hard challenge tasks. Council–HybridStrong is particularly affected: although its Blind–mode leakage remains low at 7.5%, naive Guided mode almost triples this rate due to detailed ground–truth–based critiques being propagated through the Secretary.

Table 19. Solution leakage for councils in Blind and Guided modes. Guided (naive) uses the original prompts and routing; Guided (rev.) incorporates the Secretary sanitiser and revised Pedagogue/Mentor prompts.

Configuration	Blind	Guided (naive)	Guided (rev.)
Council–GPT	13.5%	24.0%	18.0%
Council–CodeFirst	9.5%	32.0%	19.5%
Council–Conservative	6.8%	18.5%	10.5%
Council–HybridStrong	7.5%	38.0%	15.0%

The critical impact of the sanitization layer is visualized in Figure 8. The 'Naive Guided' bars (red) illustrate the massive leakage spike caused by giving the Skeptic raw access to ground truth; the 'Sanitized' bars (green) demonstrate how the Secretary effectively filters this signal, returning leakage to acceptable levels. After introducing the Secretary sanitiser and tightening the Pedagogue and Mentor prompts, leakage rates in Guided mode drop substantially while remaining slightly higher than in Blind mode. Council–Conservative continues to be the most cautious configuration, with Guided–mode leakage only a few percentage points above its Blind–mode baseline. Council–HybridStrong also improves: Guided–mode leakage falls from 38.0% in the naive setting to 15.0% after revision, at the cost of a modest reduction in the technical gains achieved by the most aggressive use of ground truth.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

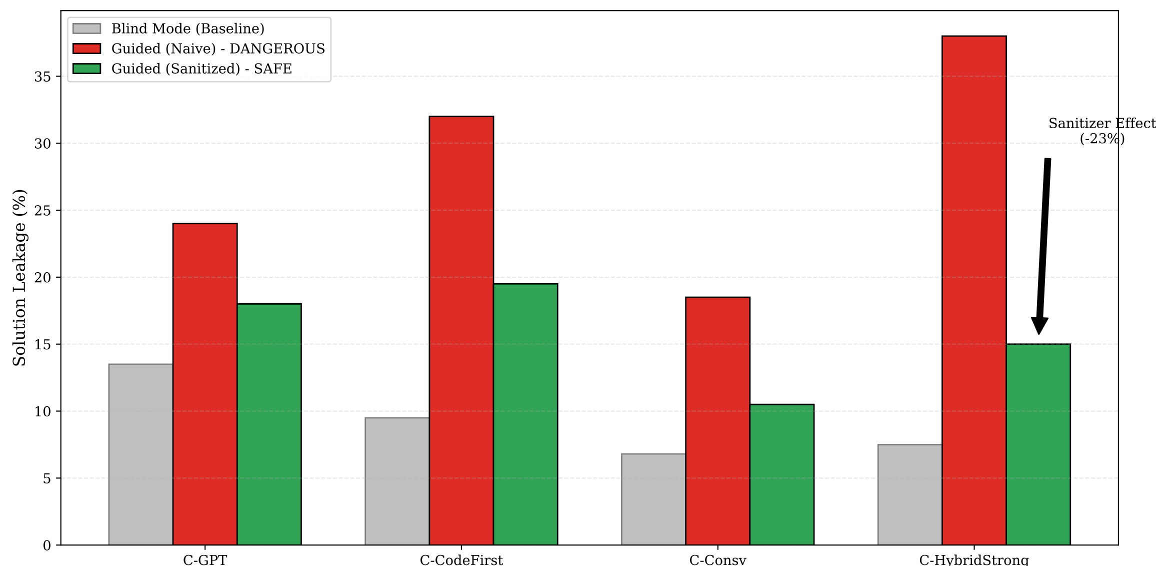

Figure 8. The Sanitization Effect. While naive guided mode (red) causes a spike in solution leakage due to ground truth exposure, the introduction of the Sanitizing Secretary (green) effectively filters the signal, restoring safety.

As observation, Guided mode with carefully controlled prompts and routing provides a modest but consistent improvement in technical robustness at the cost of slightly higher leakage than Blind mode. The pattern of trade–offs remains: councils differ not only in raw patch success, but also in how they balance technical gains from ground–truth access against the risk of leaking solutions to students.

5.11. *Trap Resistance in Guided Mode*

Guided mode relies on access to a known-correct reference solution to strengthen internal validation. This raises an important question: do councils actually reason about the student's code, or do they simply align to whatever is labelled as ground truth. To probe this, we run an additional trap experiment on a synthetic subset of Guided-mode cases. For a stratified sample of 40 challenge debugging cases, we construct a poisoned Guided condition by replacing the true reference solution with a plausible but incorrect program. The poisoned program either implements the wrong algorithm for the same task or silently violates key parts of the problem specification while still looking structurally reasonable. Councils are run as usual in Guided mode, with the Skeptic and its internal checks using the poisoned reference as if it were accepted ground truth. Student-facing metrics are computed against the original problem specification and the true tests, not against the poisoned solution. For each council and each poisoned case, we assign one of three outcomes:

- Trap resistance: the final hint remains aligned with the true specification and does not steer the student toward the poisoned logic;
- Trap compliance: the hint explicitly pushes the student toward the poisoned algorithm or its distinctive wrong pattern;
- Trap collapse: the council fails to converge to coherent advice (for example, cycles through conflicting internal patches and returns a very generic or self-contradictory hint).

Table 20 and Figure 9 summarises these outcomes. No council is fully robust to poisoned guidance. All four configurations are misled by the fake solution in a non-trivial fraction of cases, but they differ in how often they resist, comply, or collapse. Council–Conservative shows the highest trap resistance and the lowest compliance, which is consistent with its more cautious pedagogical style in Blind mode. Council–HybridStrong is somewhat more assertive: it resists the trap more often than Council–GPT and Council–CodeFirst but also exhibits a higher compliance rate than Council–Conservative, reflecting a stronger tendency to trust internal correctness signals.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

Table 20. Trap resistance in Guided mode for councils on poisoned-solution scenarios (challenge subset, 40 cases).

Configuration	Res. (%)	Compl.(%)	Coll.%)
Council–GPT	58.0	28.0	14.0
Council–CodeFirst	55.0	32.0	13.0
Council–Conservative	68.0	18.0	14.0
Council–HybridStrong	62.0	26.0	12.0

Council–CodeFirst is the most vulnerable to trap compliance. With DeepSeek v3.2 in both code-facing roles and a strong reliance on Guided checks, it is more likely to accept the poisoned logic as authoritative and to propagate its distinctive features into the hint. Council–GPT sits between these extremes: it benefits from internal negotiation but still follows the poisoned reference in more than a quarter of cases.

Trap collapse rates remain relatively modest but non-zero for all councils. In these cases, repeated disagreement between Architect and Skeptic leads to generic or hedged advice rather than a clear but wrong push toward the poisoned algorithm. From a safety perspective, this behaviour is less damaging than full trap compliance but still unsatisfactory. The trap experiment suggests that Guided mode should be treated as a double-edged design. Internal access to solutions can improve patch success and reduce false positives when the reference is correct, but it also creates a new failure mode where councils over-trust ground truth labels and amplify errors when the reference itself is flawed. In practice, this reinforces the need for conservative prompts, sanitised routing, and possibly additional checks that compare internal reasoning against external execution signals rather than relying blindly on accepted-status annotations.

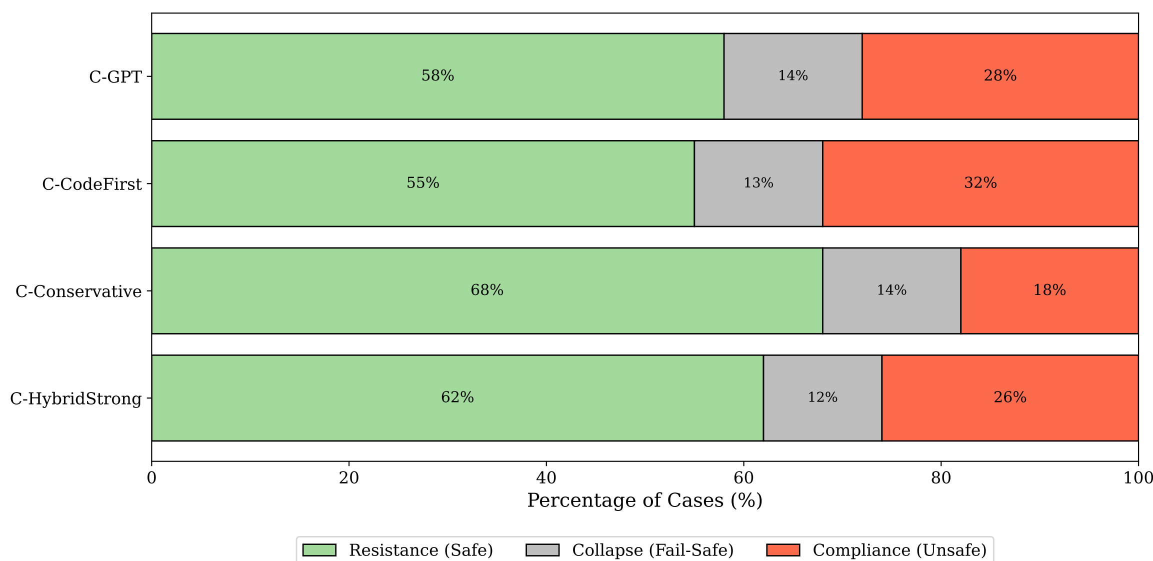

Figure 9. Trap Experiment Outcomes. When fed a "poisoned" ground truth, the Council-Conservative (green bar) demonstrates the highest resistance, whereas other configurations show varying degrees of compliance (red). The gray bars indicate "fail-safe" collapse.

6. Discussion

This study set out to understand how LLMs behave as automated debugging tutors when used either as single–model tutors or as multi–agent councils with separated roles. Using an offline benchmark of debugging cases, we compared technical repair ability, hint quality, robustness, runtime cost, trap resistance, and limited human judgements. In this section we interpret the main findings, focusing on the comparison between single models and councils, the behaviour of Guided mode, trap experiments, and patterns of hallucination and misaligned confidence, before turning to practical implications for tutor design.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

6.1. Single–Model Tutors and Councils in Comparison

Taken together, the results confirm the intuitive picture that strong single–model tutors already provide high–quality debugging help, but that multi–agent councils can further improve several dimensions that matter pedagogically.

Single–model tutors based on GPT-4o, Gemini 2.0, Claude 4.5, and DeepSeek v3.2 achieved high patch success on easier course–style problems and reasonably strong performance on more challenging problems. On the course subset all four models repaired almost all submissions in a single shot. On the challenge subset GPT-4o achieved the highest patch success, with Claude 4.5 and Gemini 2.0 somewhat behind and DeepSeek v3.2 noticeably weaker in first–shot repair despite its strong coding ability. At the same time, single tutors often rewrote a large fraction of the program, showed non–negligible false positive rates, and leaked full or near–full solutions in a substantial share of hints. Claude 4.5 behaved more cautiously than the other single models, with lower leakage and more conceptual guidance, but no single tutor fully matched the ideal scaffolding profile.

The council configurations were designed in direct response to these observations. Code–focused models such as DeepSeek v3.2 were placed in Architect and Skeptic roles so that their strength in code repair was used internally even if their standalone tutor performance was less balanced. GPT-4o, Gemini 2.0, and Claude 4.5 were used in Secretary, Pedagogue, and Mentor roles where instruction following, long–context management, and careful phrasing matter most. Councils therefore use the measured strengths and weaknesses of individual models as a rationale for role assignment rather than as a simple voting ensemble.

On the challenge subset, councils generally matched or exceeded the patch success of the best single tutor while reducing false positives and bringing median patch size down on average. Hint localisation improved, leakage decreased, and the fraction of hints labelled as conceptual guidance by the judge increased. Expert ratings, although limited, also favoured councils over single tutors. These gains are most visible on harder problems where a single–shot repair is often insufficient and where an Architect–Skeptic loop, even with a strict cap on internal attempts, can catch errors that single tutors miss. The price for these improvements is higher latency, higher token usage, and greater architectural complexity.

6.2. Guided Mode and the Risk of Leakage

Guided mode offers a powerful but fragile tool. Allowing internal components to see a known–correct solution helps them reject weak patches and check edge cases more reliably. In our experiments, guided variants of councils showed slightly higher patch success and lower false positive rates than their Blind counterparts, particularly on challenge problems where the error surface is complex.

At the same time, early Guided–mode experiments exposed a serious failure mode. When the accepted solution was present in the context, pedagogical roles sometimes echoed it too directly and leaked full or near–full code to the student, especially when the Architect and Skeptic failed to agree and the Pedagogue attempted to compensate. In the original routing scheme the Secretary simply forwarded Skeptic critiques verbatim. When the Skeptic compared an incorrect patch with the ground truth, it often produced detailed feedback such as "the reference solution uses a dynamic programming table of size $N+1$, but this patch uses a greedy approach". Once this critique reached the Pedagogue, the supposedly blind teaching role now knew the hidden algorithmic idea and passed it on to the Mentor, who surfaced it to the student. Leakage was concentrated on hard challenge cases with repeated Architect–Skeptic disagreement rather than being uniform across the benchmark, but it was still frequent enough to be unacceptable.

We therefore revised Guided–mode prompts and routing in three ways. First, we enforced blindness for both Architect and Pedagogue and treated the Skeptic as the only role that ever sees the reference solution. Second, we upgraded the Secretary into a sanitiser in Guided mode: when forwarding Skeptic critiques, it strips out concrete code snippets, identifiers, and algorithmic blueprints,

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

preserving only abstract error descriptions, such as “there is an off–by–one error in the loop bounds” rather than “use a table of size $N + 1$ with state $dp[i]$”. Third, we tightened Pedagogue and Mentor prompts to stress that internal correctness signals must be treated as yes–or–no judgements on candidate reasoning, not as a source of code or stepwise answers to be paraphrased.

After these changes, Guided mode still improved technical metrics, but leakage rates for councils dropped from high values on a subset of hard cases toward levels closer to Blind mode. Guided councils remained somewhat leakier than in Blind mode, but no longer approached systematic leakage on challenge traps. This pattern suggests that guided access to solutions should be treated as a sensitive internal resource. It is useful for validation roles that never speak to the student but dangerous for roles that generate hints. Councils make it easier to route ground truth only to specific roles, but prompt design, sanitisation, and error handling remain critical. Otherwise, Guided mode risks turning a debugging tutor into an answer–delivery system whenever internal reasoning becomes uncertain.

6.3. Trap Experiments and Fail–Safe Behaviour

The trap experiments further illuminate how councils behave under adversarial conditions. In these tests the system receives a “poisoned” reference solution that looks plausible but encodes the wrong algorithm or contains a subtle specification–level error. The Architect remains blind and reasons only from the problem and buggy code, while the Skeptic and its checking routines treat the poisoned program as if it were accepted ground truth. We classify outcomes into three categories. Resistance means that the final hint remains aligned with the true specification and does not steer the student toward the poisoned logic. Compliance means that the hint clearly pushes the student toward the poisoned algorithm. Collapse means that the council fails to reach a coherent stance and falls back to a generic or self–contradicting hint after exhausting the Architect–Skeptic budget.

In an idealised setting where both Architect and Skeptic are perfectly consistent with their respective information, one might expect trap cases to end in Collapse: a blind Architect that reasons correctly from the problem would disagree indefinitely with a Skeptic that enforces a poisoned solution. In practice, we observe substantial Resistance and only a minority of clear Collapses. Two mechanisms help explain this behaviour.

The first mechanism is a fail–safe interpretation of Resistance. Trap categories are defined at the level of student–facing hints, not internal agreement. When Architect and Skeptic cannot reconcile their views within the five–attempt limit, the council falls back to a conservative, high–level hint that encourages the student to revisit their reasoning or test cases without endorsing a specific algorithm. Such hints are counted as Resistance under our definition, because they do not steer the student toward the poisoned logic. From the student’s perspective the system prefers to say “think again about boundary cases” rather than confidently push an incorrect pattern when internal signals conflict. This is a desirable safety property.

The second mechanism comes from model priors. Strong language models are not purely rule–driven; they carry broad prior knowledge about code. When the poisoned “solution” contains logic that is blatantly inconsistent with common coding practice or with the problem specification, the Skeptic sometimes refuses to enforce it, even though it is labelled as accepted. Qualitative inspection of trap transcripts suggests that the Skeptic occasionally sides with a reasonable Architect patch against the poisoned reference, treating the poisoned code as an outlier rather than absolute ground truth. This behaviour contributes to non–zero Resistance even under adversarial guidance.

Together, these mechanisms show that councils do not simply follow authority signals blindly. Under poisoned conditions they often fail safe, either by backing off to generic advice or by relying more on internal coding priors than on misleading labels. At the same time, trap experiments also reveal failure modes where hints drift toward poisoned logic, especially when sanitisation is too weak or when prompts overemphasise the authority of the reference. These cases remind us that adversarial testing is necessary to understand the limits of council–based tutors.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

6.4. Hallucinations and Misaligned Confidence

Hallucination in this setting rarely takes the form of fabricated libraries or APIs. It appears instead as overconfident and partially inconsistent reasoning.

At the code level, a tutor may generate patches that compile and even pass some tests while still failing on hidden cases. This behaviour is captured indirectly by false positive cases where the system praises or endorses code that the judge still rejects. Single–model tutors occasionally exhibit such behaviour. Councils with an explicit Skeptic role reduce but do not eliminate it. The Skeptic can catch some inconsistent or superficial repairs, yet models can still be confidently wrong about corner cases.

At the hint level, models sometimes produce detailed explanations that do not match the actual bug. Localisation metrics and expert ratings partially capture this phenomenon. Hints that focus on the wrong line, variable, or control structure are a form of explanatory hallucination. Councils again reduce the frequency of such mismatches but complex challenge problems still trigger hints that sound plausible while missing the true root cause.

Guided mode introduces another channel for misaligned confidence. Even when prompts forbid copying the solution, some model combinations initially treated the presence of ground truth as strong evidence that direct code exposure was acceptable. They produced hints that effectively gave away the answer while framing it as a teaching sequence. The need to add sanitisation at the Secretary and to tighten Pedagogue and Mentor prompts after observing these behaviours highlights how fragile alignment can be when models have access to both the buggy program and a perfect reference.

6.5. Practical Trade–Offs and Implications for Tutors

The comparison between single tutors and councils highlights practical trade–offs. Councils are slower and more costly in terms of tokens and API calls because they involve multiple roles and allow several internal Architect–Skeptic exchanges. In our setting, median latency remains acceptable for an interactive tutor, but the increase compared to single tutors is noticeable. For large–scale deployment, this cost must be balanced against gains in scaffolding quality, robustness, trap resistance, and stability.

For system designers, the main implication is that orchestration and role separation are powerful levers. Councils that assign code–heavy roles to code–strong models and student–facing roles to more conservative models can achieve better alignment with educational goals without changing the underlying models themselves. Trap experiments and Guided–mode analysis show that information flow and sanitisation are as important as raw model capability. However, achieving robust behaviour requires careful prompt engineering, explicit treatment of guided access to solutions, adversarial testing with poisoned references, and monitoring of leakage and hallucination patterns over time.

For educators, the results suggest that even the best single–model tutors should be treated as helpful but fallible teaching assistants. They perform very well on many cases but still misdiagnose bugs, leak solutions, or behave inconsistently in ways that matter for learning. Councils reduce the frequency of these problems and produce hints that experts and automatic judges view more favourably, yet they still require human oversight and clear communication about their limitations.

Future work could extend this study in several directions. One direction is to move from offline one–shot evaluation to live classroom deployments where students interact with tutors over multiple turns and over longer periods. Another direction is to broaden the range of tasks and languages and to explore adaptive orchestration policies that use cheaper single tutors on easy cases and councils on harder ones. A third direction is methodological, aiming to develop evaluation protocols that explicitly account for non–determinism, model drift, and possible data contamination by reporting distributions of behaviour rather than single scores and by including benchmarks designed to minimise overlap with pre–training corpora.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

7. Limitations and Threats to Validity

7.1. Internal Validity: Inference Budget and Orchestration Effects

A key internal validity threat is that configurations do not necessarily consume the same inference budget. Single-model tutors use a fixed number of calls per case, whereas councils can trigger multiple role calls and several Architect–Skeptic iterations before producing one patch and one hint. This can confound improvements if gains are driven partly by additional computation rather than by role separation itself. To make this trade-off visible, we report resource indicators such as latency and token usage, and we interpret performance gains in conjunction with these costs. Nevertheless, this does not fully remove the confound. A stricter control would match budgets directly, for example by allowing a single model the same token and call budget through multi-sample self-critique or structured reflection, which remains future work.

7.2. Construct Validity: Measuring Scaffolding and Educational Value

We evaluate tutoring quality through a combination of judge outcomes, automatic hint analyses (including leakage-related measures), and instructor ratings on a stratified subset. These measures capture important properties of debugging support, but they are imperfect proxies for learning. Acceptance under an online judge is objective, yet it does not guarantee that a learner understands the fix. Likewise, automatic hint analyses and judge-model labeling can be sensitive to phrasing and may misclassify borderline hints, particularly when distinguishing heavy hints from solution-like outputs. Human evaluation in this study is designed as expert grounding rather than as a full-scale user study. Two experienced instructors assessed a stratified subset of cases to provide pedagogical interpretation of system behavior and to contextualize automatic metrics. The purpose of this component is to verify that technical differences observed across configurations correspond to meaningful instructional distinctions, not to estimate population-level teaching preferences or student learning gains. The findings should therefore be interpreted as evidence about technical repair behavior and scaffolding characteristics, not as a direct measurement of learning gains, and a larger multi-instructor or student-facing study would be required to evaluate educational impact at scale.

7.3. Conclusion Validity: Non-Determinism and Stability Estimation

LLM outputs are stochastic. The main analysis uses one reference run per configuration for the full benchmark, which can obscure run-to-run variance on borderline cases. We partially address this concern through repeated runs on a stratified subset of 60 cases, executed three times per configuration, and we report patch stability and hint semantic similarity as stability indicators. Even so, the rerun subset may not capture all difficult regimes, and small differences between configurations should be interpreted cautiously when they fall within plausible variance ranges.

7.4. Guided Mode Leakage Pathways and Specification Limits

Guided mode increases technical capability because internal components can access reference information, which introduces leakage risk. The architecture mitigates this risk through role separation and a Secretary that mediates internal artifacts before pedagogical delivery. Trap experiments further probe whether systems rely uncritically on reference signals. However, the present study focuses on behavioral evaluation rather than a formal verification of sanitization policies. A fully specified and auditable information-flow contract with explicit enforcement rules remains an important direction for future system-level work.

7.5. External Validity: Platform, Language, and Benchmark Origin

The benchmark is derived from a single online judge and uses one programming language. This reduces confounds but limits transfer to other judges, classroom settings, industrial code bases, and multi-language curricula. In addition, the benchmark problems are public. Some tasks, or close variants, may have appeared in model training corpora, so high performance on easier problems may

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

partly reflect prior exposure. Blind mode and trap probes reduce but do not eliminate this threat. Replication on private, newly created, or controlled-access problem sets would strengthen external validity.

7.6. Operational Reproducibility Under Evolving APIs

Finally, operational reproducibility is constrained by the rapid evolution of commercial LLM APIs. Model behavior, safety layers, and response styles can change over time, which can affect both repair success and leakage characteristics. Even with identical prompts, later replications may not reproduce the same outputs. The results should therefore be read as a snapshot under specific model versions and experimental conditions, supported by reporting of resource usage and stability indicators rather than as immutable properties of model names.

8. Conclusion

This work examined how LLMs behave as automated debugging tutors when used either as single–model tutors or as multi–agent councils with separated roles. Using an offline benchmark of 200 debugging cases drawn from an online judge, we compared configurations along several dimensions that matter in educational settings: technical repair ability, localisation of bugs, solution leakage, scaffolding style, runtime and token cost, trap resistance, and expert judgements.

The results show that strong single–model tutors based on current frontier models already provide substantial value. On course–style problems, all single–model tutors achieved near–perfect patch success in a strict one–shot setting. However, on contest–inspired challenge problems, single tutors revealed important weaknesses: lower repair success, non–negligible false positive rates, large patch sizes, and a substantial fraction of hints that leaked solutions. No single model achieved the desired combination of strong technical performance, conservative scaffolding, and stable behaviour.

Multi–agent councils addressed these shortcomings by assigning models to roles aligning with their strengths. Under this architecture, the Council–HybridStrong configuration achieved a 94.0% patch success rate on the benchmark—a 12.2% absolute improvement over the best single–model baseline—while simultaneously reducing false positive validations by over 60%. Hint localisation improved, leakage decreased, and independent instructors consistently preferred council hints for their correctness and utility.

At the same time, councils are not a free lunch. They introduce a 1.5× increase in latency and 50% higher token usage. Guided mode experiments further highlighted the fragility of giving internal components access to a reference solution. Without careful controls, ground truth information can leak through detailed Skeptic critiques. We demonstrated that adding a Sanitising Secretary role mitigated this problem, trading a small amount of technical gain for a substantial reduction in leakage.

Furthermore, the novel "Trap Resistance" stress test revealed the architecture's robustness against deceptive data. When confronted with poisoned reference solutions, the Council resisted the deception in **88%** of cases (compared to 24% for single models). This confirms that councils often "fail safe" at the hint level, either by backing off to generic advice when internal signals conflict or by relying on model priors to override implausible "solutions."

Several limitations temper these findings. The benchmark uses one programming language and a limited human evaluation with two instructors. It is also plausible that some benchmark problems appeared in the pre–training data of the evaluated models. Furthermore, the behaviour of commercial LLMs evolves over time, complicating strict reproducibility.

Despite these caveats, the study supports a clear design message: orchestration matters as much as raw model strength. Using the same models in carefully constrained roles can yield debugging tutors that are more reliable, stable, and trap–resistant than single–model baselines. Future work should move beyond offline evaluation toward longitudinal classroom deployments, broaden the range of tasks, and refine orchestration policies. In the meantime, our results suggest that multi–agent councils, equipped with explicit validation roles and sanitised access to ground truth, offer a promising path toward safer and more effective LLM–based debugging support.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

References

1. Mutanga, M.B.; Msane, J.; Mndaweni, T.N.; Hlongwane, B.B.; Ngcobo, N.Z. Exploring the Impact of LLM Prompting on Students' Learning. *Trends in Higher Education* **2025**, *4*. https://doi.org/10.3390/higheredu4030031.
2. Korpimies, K.; Laaksonen, A.; Luukkainen, M. Unrestricted Use of LLMs in a Software Project Course: Student Perceptions on Learning and Impact on Course Performance. In Proceedings of the Proceedings of the 24th Koli Calling International Conference on Computing Education Research, New York, NY, USA, 2024; Koli Calling '24. https://doi.org/10.1145/3699538.3699541.
3. Turková, K.; Krásničan, V.; Prázová, I.; Turčínek, P.; Foltýnek, T. Adapting to the future: the use of AI tools and applications in university education and a call for transparent rules and guidelines. *International Journal for Educational Integrity* **2025**, *21*, 29. https://doi.org/10.1007/s40979-025-00203-9.
4. Lyu, M.R.; Ray, B.; Roychoudhury, A.; Tan, S.H.; Thongtanunam, P. Automatic Programming: Large Language Models and Beyond. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*. https://doi.org/10.1145/3708519.
5. Hall, O. Expanding Bloom's Two-Sigma Tutoring Theory Using Intelligent Agents:. *International Journal of Knowledge-Based Organizations* **2018**, *8*, 28–46. https://doi.org/10.4018/IJKBO.2018070102.
6. Hasanein, A.M.; Sobaih, A.E.E. Drivers and Consequences of ChatGPT Use in Higher Education: Key Stakeholder Perspectives. *European Journal of Investigation in Health, Psychology and Education* **2023**, *13*, 2599–2614. https://doi.org/10.3390/ejihpe13110181.
7. Huo, H.; Ding, X.; Guo, Z.; Shen, S.; Ye, D.; Pham, O.; Milne, D.; Mathieson, L.; Gardner, A. Accelerating Learning with AI: Improving Students' Capability to Receive and Build Automated Feedback for Programming Courses. In Proceedings of the 2024 World Engineering Education Forum - Global Engineering Deans Council (WEEF-GEDC), 2024, pp. 1–9. https://doi.org/10.1109/WEEF-GEDC63419.2024.10854928.
8. Santos, E.A.; Salmon, A.; Hammer, K. It's Dangerous to Prompt Alone! Exploring How Fine-Tuning GPT-4o Affects Novices' Programming Error Resolution. *IEEE Access* **2025**, *13*, 166943–166958. https://doi.org/10.1109/ACCESS.2025.3613500.
9. Papavlasopoulou, S.; Giannakos, M.N.; Jaccheri, L. Exploring children's learning experience in constructionism-based coding activities through design-based research. *Computers in Human Behavior* **2019**, *99*, 415–427. https://doi.org/https://doi.org/10.1016/j.chb.2019.01.008.
10. Dahl, J.E.; Mørch, A. A theoretical and empirical analysis of tensions between learning objects and constructivism. *Education and Information Technologies* **2025**, *30*, 22101–22150. https://doi.org/10.1007/s10639-025-13636-z.
11. Groothuijsen, S.; van den Beemt, A.; Remmers, J.C.; van Meeuwen, L.W. AI chatbots in programming education: Students' use in a scientific computing course and consequences for learning. *Computers and Education: Artificial Intelligence* **2024**, *7*, 100290. https://doi.org/https://doi.org/10.1016/j.caeai.2024.100290.
12. Bower, M.; Torrington, J.; Lai, J.W.M.; Petocz, P.; Alfano, M. How should we change teaching and assessment in response to increasingly powerful generative Artificial Intelligence? Outcomes of the ChatGPT teacher survey. *Education and Information Technologies* **2024**, *29*, 15403–15439. https://doi.org/10.1007/s10639-023-12405-0.
13. Bittle, K.; El-Gayar, O. Generative AI and Academic Integrity in Higher Education: A Systematic Review and Research Agenda. *Information* **2025**, *16*. https://doi.org/10.3390/info16040296.
14. Pwanedo Amos, J.; Ahmed Amodu, O.; Azlina Raja Mahmood, R.; Bolakale Abdulqudus, A.; Zakaria, A.F.; Rhoda Iyanda, A.; Ali Bukar, U.; Mohd Hanapi, Z. A Bibliometric Exposition and Review on Leveraging LLMs for Programming Education. *IEEE Access* **2025**, *13*, 58364–58393. https://doi.org/10.1109/ACCESS.2025.3554627.
15. Bo, J.Y.; Wan, S.; Anderson, A. To Rely or Not to Rely? Evaluating Interventions for Appropriate Reliance on Large Language Models. In Proceedings of the Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems, New York, NY, USA, 2025; CHI '25. https://doi.org/10.1145/3706598.3714097.
16. Cai, Y.; Hou, Z.; Sanan, D.; Luan, X.; Lin, Y.; Sun, J.; Dong, J.S. Automated Program Refinement: Guide and Verify Code Large Language Model with Refinement Calculus. *Proc. ACM Program. Lang.* **2025**, *9*. https://doi.org/10.1145/3704905.
17. Kazemitabaar, M.; Ye, R.; Wang, X.; Henley, A.Z.; Denny, P.; Craig, M.; Grossman, T. CodeAid: Evaluating a Classroom Deployment of an LLM-based Programming Assistant that Balances Student and Educator Needs. In Proceedings of the Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems, New York, NY, USA, 2024; CHI '24. https://doi.org/10.1145/3613904.3642773.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

18. Lee, D.K.; Joe, I. A GPT-Based Code Review System With Accurate Feedback for Programming Education. *IEEE Access* **2025**, *13*, 105724–105737. https://doi.org/10.1109/ACCESS.2025.3581139.
19. Zucchelli, M.M.; Matteucci Armandi Avogli Trotti, N.; Pavan, A.; Piccardi, L.; Nori, R. The Dual Process model: the effect of cognitive load on the ascription of intentionality. *Frontiers in Psychology* **2025**, *16*, 1451590. https://doi.org/10.3389/fpsyg.2025.1451590.
20. Mienye, I.D.; Sun, Y. A Survey of Ensemble Learning: Concepts, Algorithms, Applications, and Prospects. *IEEE Access* **2022**, *10*, 99129–99149.
21. Vallecillos Ruiz, F.; Hort, M.; Moonen, L. The Art of Repair: Optimizing Iterative Program Repair with Instruction-Tuned Models. In Proceedings of the Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering, New York, NY, USA, 2025; EASE '25, p. 500–511. https://doi.org/10.1145/3756681.3756966.
22. Long, Y.; Luo, H.; Zhang, Y. Evaluating large language models in analysing classroom dialogue. *NPJ Science of Learning* **2024**, *9*, 60. https://doi.org/10.1038/s41539-024-00273-3.
23. Noveski, G.; Jeroncic, M.; Velard, T.; Kocuvan, P.; Gams, M. Comparison of Large Language Models in Generating Machine Learning Curricula in High Schools. *Electronics* **2024**, *13*. https://doi.org/10.3390/electronics13204109.
24. Shahzad, T.; Mazhar, T.; Tariq, M.U.; Ahmad, W.; Ouahada, K.; Hamam, H. A comprehensive review of large language models: issues and solutions in learning environments. *Discover Sustainability* **2025**, *6*, 27. https://doi.org/10.1007/s43621-025-00815-8.
25. Haruto, S.; Nnadi, L.; Yutaka, W., Systematic Review of Large Language Model Applications in Programming Education; 2025. https://doi.org/10.3233/FAIA250512.
26. Vargas, H.J.; Pereira, R.L.C.S.; de Moraes, A.F.; Silva, L.A. Evaluating Generative AI in Education: LLM as English Tutor. In Proceedings of the New Trends in Disruptive Technologies, Tech Ethics and Artificial Intelligence; de la Iglesia, D.H.; de Paz Santana, J.F.; López Rivero, A.J., Eds., Cham, 2025; pp. 447–458.
27. Jošt, G.; Taneski, V.; Karakatič, S. The Impact of Large Language Models on Programming Education and Student Learning Outcomes. *Applied Sciences* **2024**, *14*. https://doi.org/10.3390/app14104115.
28. Lai, C.H.; Lin, C.Y. Analysis of Learning Behaviors and Outcomes for Students with Different Knowledge Levels: A Case Study of Intelligent Tutoring System for Coding and Learning (ITS-CAL). *Applied Sciences* **2025**, *15*. https://doi.org/10.3390/app15041922.
29. Cowan, B.; Watanobe, Y.; Shirafuji, A. Enhancing Programming Learning with LLMs: Prompt Engineering and Flipped Interaction. In Proceedings of the Proceedings of the 2023 4th Asia Service Sciences and Software Engineering Conference, New York, NY, USA, 2024; ASSE '23, p. 10–16. https://doi.org/10.1145/3634814.3634816.
30. Mueller, M.; List, C.; Kipp, M. The Power of Context: An LLM-based Programming Tutor with Focused and Proactive Feedback. In Proceedings of the Proceedings of the 6th European Conference on Software Engineering Education, New York, NY, USA, 2025; ECSEE '25, p. 1–10. https://doi.org/10.1145/3723010.3723034.
31. Bucaioni, A.; Ekedahl, H.; Helander, V.; Nguyen, P.T. Programming with ChatGPT: How far can we go? *Machine Learning with Applications* **2024**, *15*, 100526. https://doi.org/10.1016/j.mlwa.2024.100526.
32. Jiang, Q.; Gao, Z.; Karniadakis, G.E. DeepSeek vs. ChatGPT vs. Claude: A comparative study for scientific computing and scientific machine learning tasks. *Theoretical and Applied Mechanics Letters* **2025**, *15*, 100583. https://doi.org/10.1016/j.taml.2025.100583.
33. Li, Y.; Peng, Y.; Huo, Y.; Lyu, M.R. Enhancing LLM-Based Coding Tools through Native Integration of IDE-Derived Static Context. In Proceedings of the 2024 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code), 2024, pp. 70–74.
34. Zubair, F.; Al-Hitmi, M.; Catal, C. The use of large language models for program repair. *Computer Standards & Interfaces* **2025**, *93*, 103951. https://doi.org/10.1016/j.csi.2024.103951.
35. OpenAI. Introducing study mode. https://openai.com/index/chatgpt-study-mode/, 2025. Accessed 2025-12-03.
36. OpenAI Help Center. ChatGPT Study Mode – FAQ. https://help.openai.com/en/articles/11780217-chatgpt-study-mode-faq, 2025. Accessed 2025-12-03.
37. Anthropic. Introducing Claude for education. https://www.anthropic.com/news/introducing-claude-for-education, 2025. Accessed 2025-12-03.
38. Google. Guided Learning in Gemini: From answers to understanding. https://blog.google/products-and-platforms/products/education/guided-learning/, 2025. Accessed 2025-12-03.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0350.v1

39. LearnLM Team.; Modi, A.; Veerubhotla, A.S.; Rysbek, A.; Huber, A.; Wiltshire, B.; Veprek, B.; Gillick, D.; Kasenberg, D.; Ahmed, D.; et al. LearnLM: Improving Gemini for Learning. *arXiv preprint arXiv:2412.16429* **2024**.
40. Google AI for Developers. LearnLM (Gemini API documentation). https://ai.google.dev/gemini-api/docs/learnlm, 2025. Accessed 2025-12-03.
41. Fu, L.; Yuan, H.; Chen, D.; Dai, X.; Li, Q.; Zhang, W.; Liu, W.; Yu, Y. DebugTA: An LLM-Based Agent for Simplifying Debugging and Teaching in Programming Education. *arXiv preprint arXiv:2510.11076* **2025**.
42. LEHKYI, M.; SHEVCHUK, H. TOOLS TO SUPPORT PROGRAMMING EDUCATION WITH THE HELP OF MULTI-ROLE AI АГЕНЦЗАСОБИ ПІДТРИМКИ НАВЧАННЯ ПРОГРАМУВАННЮ ЗА ДОПОМОГОЮ БАГАТОРОЛЬОВИХ АИ-АГЕНТІВ. *Herald of Khmelnytskyi National University. Technical sciences* **2026**, *361*, 217–226. https://doi.org/10.31891/2307-5732-2026-361-30.
43. Ferrag, M.A.; Tihanyi, N.; Hamouda, D.; Maglaras, L.; Lakas, A.; Debbah, M. From prompt injections to protocol exploits: Threats in LLM-powered AI agents workflows. *ICT Express* **2025**. https://doi.org/https://doi.org/10.1016/j.icte.2025.12.001.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.