ATLAS: A Dual-Mode Multi-Agent Architecture for AI Code Assistance That Balances Developer Productivity and Skill Development

Gautam Jhalaria[1], Laxmi Verma[2], Sujal Sen[3], Mahi Uboweja[4], Narendra Dewangan[5] Department of AIML
Shri Shankaracharya Institute of Professional Management and Technology Raipur, Chhattisgarh, India[12345]
gautamjhalriya@ssipmt.com[1], laxmi@ssipmt.com[2], ssen@ssipmt.com[3], mahi.uboweja@ssipmt.com[4], narendra.d@ssipmt.com[5]

Abstract—Coding assistants powered by artificial intelligence (AI) have become very popular, with more than 90% of professional developers saying they use them regularly by 2026. Recent empirical evidence indicates a fundamental tension: tools that enhance productivity via autonomous code generation may lead to cognitive offloading and skill atrophy, resulting in developers achieving 17% lower scores on conceptual mastery assessments [1]. Conversely, tools that emphasize learning through scaffolded guidance compromise development speed. The ”productivity-learning paradox” is another way to show this tension. A randomized controlled trial showed that developers are objectively 19% slower with AI tools, even though they think they are 20% faster [2].

This paper introduces ATLAS (Adaptive Teaching and Learning Assistant for Software), a dual-mode AI code assistant that integrates autonomous code generation (Build mode) and Socratic mentoring (Mentor mode) within a unified multi-agent architecture. In build mode, a cyclical LangGraph StateGraph of specialized agents—Architect, Planner, Developer, and Reviewer—automatically creates, checks, and applies code patches. In Mentor mode, the same Retrieval-Augmented Generation (RAG) context pipeline is used, but it uses a different five-agent topology (with roles like Skeptic, Pedagogue, and Mentor) to stop solutions from leaking. It limits tool access to read-only operations and replaces direct code generation with scaffolded dialogue, which is enforced at the architecture level with smooth state recovery. Both modes use the same context engineering pipeline, which puts together system prompts, retrieved code context, active file state, and conversation history in real time.

ATLAS is evaluated through a within-subjects user study with 48 participants completing 4 programming tasks in both modes. Results demonstrate that Build mode reduces end-to-end task resolution time by 43.1% compared to unassisted development ($p < 0.001$), while Mentor mode improves conceptual understanding scores by 48.2% compared to Build mode ($p < 0.001$). 75% of participants expressed a preference for the ability to switch between modes based on task complexity. The contributions of this paper are threefold:

(1) a formal dual-mode architecture with server-side enforcement that prevents pedagogical guardrail bypassing, (2) a shared context engineering pipeline that adapts retrieval and prompt assembly per interaction mode, and (3) empirical evidence supporting the efficacy of bimodal AI assistance in addressing the productivity-learning paradox.

Index Terms—AI Code Assistant, Multi-Agent Systems, Large Language Models, Socratic Tutoring, Software Engineering, Dual-Mode Architecture, Retrieval-Augmented Generation.

1 INTRODUCTION

Large Language Models (LLMs) have reshaped software development since 2023. By 2026, AI-powered coding assistants—GitHub Copilot, Cursor, Claude Code—have reached near-universal adoption within professional soft- ware engineering teams [3], promising productivity gains through automated code generation, intelligent comple- tions, and autonomous debugging.

Empirical evidence contradicts this promise. A random- ized controlled trial by METR in 2025 found that expe- rienced open-source developers were 19% slower when using AI coding tools, despite believing they were 20% faster [2]. This ”perception gap” stemmed from prompt engineering overhead, output validation, and integration friction in complex codebases. Shen and Tamkin [1] reported a complementary finding: developers using AI assistance scored 17% lower on conceptual mastery assessments— nearly two letter grades—compared to those who coded manually. The mechanism was cognitive offloading: de- velopers outsourced reasoning to AI, bypassing the effort required for durable knowledge construction.

These findings expose a tension in AI-assisted software development. Tools that maximize productivity through autonomous code generation—ALMAS [4], MetaGPT [5], ChatDev [6]—risk exacerbating cognitive offloading by re- ducing the developer’s active engagement. Tools designed for pedagogical effectiveness—CodeAid [7], SocraticAI [8], LeetCoach [9]—enforce learning-oriented interactions but cannot ship production code. No existing system unifies both paradigms.

This paper presents ATLAS (Adaptive Teaching and Learning Assistant for Software), a dual-mode AI coding as- sistant that bridges this gap. ATLAS integrates two distinct

interaction modes:

- **Build Mode** deploys a multi-agent cyclical pipeline of specialized LangGraph agents (Architect, Plan- ner, Developer, Reviewer) that autonomously plan, generate code, apply patches, and validate changes. It operates within a hybrid backend environment, sending execution payloads to an isolated Docker sandbox while retaining web access for external rea- soning.
- **Mentor Mode** activates the same Retrieval-Augmented Generation (RAG) context pipeline but orchestrates a five-agent topology to enforce a So-cratic interaction contract. It guarantees zero solu- tion leakage by evaluating the user's code against a hidden "Ground Truth" and providing scaffolded dialogue. Crucially, this enforcement is implemented via explicit server-side middleware and StateGraph hooks—ensuring that even if an LLM node halluci-nates a code patch, the backend recovers gracefully and enforces the pedagogical role.

Both modes share a unified context engineering pipeline that assembles system prompts, RAG-retrieved code snip-pets, active file context, and conversation history, adapting the composition per mode. This design ensures that the quality of code understanding is consistent across both modes while the interaction semantics differ.

The contributions of this paper are as follows:

1) **Dual-mode architecture with server-side enforce-ment:** We propose and implement the first AI cod- ing assistant that formally integrates autonomous code generation and Socratic mentoring within a single platform, with mode contracts enforced at the API layer rather than relying solely on prompt-level constraints.
2) **Shared context engineering framework:** We design a mode-adaptive context assembly pipeline that leverages the same RAG retrieval, file indexing, and conversation management infrastructure for both modes, differing only in system prompt templates and tool permission sets.
3) **Empirical evaluation:** We conduct a within-subjects user study with 48 developers, demonstrating that the dual-mode approach effectively addresses the productivity-learning paradox: Build mode accel-erates task completion while Mentor mode signif-icantly improves conceptual understanding.

The rest of this paper is set up like this: Section II looks at other work on AI coding assistants, pedagogical AI, and multi-agent software engineering. In Section III, we talk about the ATLAS architecture. Section IV talks about how it was put into action. Section V talks about the methods and results of the evaluation. Section VI talks about the effects and limits. The paper ends with Section VII.

2 RELATED WORK

2.1 The Productivity-Learning Paradox in AI-Assisted Development

Several recent studies have measured how AI coding assis-tants affect how well developers do their jobs and how they learn new skills. The METR randomized controlled trial [2] examined 16 seasoned open-source developers across 246 real-world issues, revealing a 19% objective slowdown due to prompt engineering overhead, low suggestion acceptance rates (< 44%), and the cognitive load associated with re- viewing AI-generated code. The Shen and Tamkin study [1] at Anthropic added to this by doing a controlled experiment with 52 people learning a new Python library. They found that AI-assisted participants had a statistically significant 17% lower level of conceptual mastery.

Gerlich et al. [10] found that people who used AI a lot had lower critical thinking scores. This was especially true for younger and less experienced users. An industry-level analysis in 2026 showed that even though more than 90% of developers use the software, organizational productivity gains have leveled off at about 10% [11]. This goes against the idea that productivity improvements are revolutionary.

These results show the productivity-learning paradox: AI tools that speed up code production may also make it harder to learn the skills needed to maintain, debug, and add to that code.

2.2 Build-Mode Systems: Autonomous Code Genera- tion

Multi-agent systems for automated software development have matured since 2024. ALMAS [4], developed by J.P. Morgan AI Research, assigns LLM agents to agile SDLC roles—Product Manager, Developer, Tester, Peer Reviewer—and integrates with enterprise tools (Jira, Bit-bucket). MetaGPT [5] introduced a standardized operat- ing procedure (SOP) framework that coordinates agents through publish-subscribe messaging. ChatDev [6] pro- posed a chat-chain communication protocol for sequential agent collaboration.

More recently, the SWE-bench benchmark [12] has be-come the standard evaluation framework for autonomous coding agents, testing their ability to resolve real-world repository issues through patch generation. Industry tools including Cursor, Claude Code, and Windsurf have achieved varying degrees of success on this benchmark.

These systems demonstrate impressive code generation capabilities but operate exclusively in an autonomous gen- eration paradigm. They lack a framework for scaffolded learning, and their design is specifically tailored to reduce human cognitive interaction with the code—exactly the pattern recognized as leading to skill atrophy [1].

2.3 Mentor-Mode Systems: Pedagogical AI for Pro-gramming

A different area of research has focused on making AI tools for use in schools that have built-in rules for teaching. This is different from generative systems. CodeAid [7], used in a 700-student introductory programming course, set up guardrails that stopped students from writing code that could be run right away. Instead, it gave them pseudo-code, conceptual explanations, and debugging hints based on the course materials through RAG. The research indicated that although students valued the scaffolded assistance, some returned to unrestricted LLMs (e.g., ChatGPT) when the

limitations appeared excessively confining—a phenomenon identified as ”reversion risk.”

SocraticAI [8] introduced a structured interaction framework requiring students to articulate their current approach, prior attempts, and specific confusion points before receiv- ing guidance. Deployed at Ashoka University, the system demonstrated that over 75% of participants produced substantive reflections, and students shifted from vague help-seeking to sophisticated problem decomposition within 2–3 weeks. Liyanage et al. [9] further extended this approach through cognitive forcing strategies, prompting learners to take incremental steps rather than accepting complete solutions.

CodeGuard [13] developed taxonomies for prompt classification and implemented real-time moderation layers to detect and filter non-pedagogical prompts. The Code Coun- cil [14] proposed separating ”code-facing” and ”student- facing” agent roles to manage information flow and prevent solution leakage in multi-agent educational systems.

However, these systems are designed exclusively for educational contexts. They cannot generate deployable code, apply file system patches, or execute commands in a development environment. A developer needing to ship working code cannot use these tools productively.

2.4 The Gap: No Unified Dual-Mode System

Table I gives a summary of the landscape. Current tools only help with one of the two: productivity or learning. No system offers a unified platform that enables a developer to independently generate code for routine tasks and subse- quently transition to guided learning for intricate, unfamil- iar challenges, with mode contracts enforced architecturally rather than through easily bypassed prompt instructions.

ATLAS addresses this gap by unifying both paradigms within a single architecture, sharing context engineering across modes while enforcing distinct interaction contracts at the server level.

3 ATLAS ARCHITECTURE

ATLAS consists of four architectural layers: a system core, Build mode pipeline, Mentor mode pipeline, and shared context engineering infrastructure. Each is described below.

3.1 System Overview

ATLAS is a web-based AI code assistant consisting of a FastAPI [15] backend, a browser-based frontend, and infrastructure services. Fig. 1 illustrates the high-level architec- ture.

The system operates through a persistent WebSocket connection that enables real-time, bidirectional communica- tion between the user and the AI assistant. All interactions are managed within user sessions, each of which maintains a mode state (Build or Mentor), conversation history, active file context, and checkpoint references. The mode state de- termines the behavior of the system, including which tools are available, which system prompt is used, and whether code patches are permitted.

3.2 Build Mode

Build mode implements an agentic code generation pipeline designed for autonomous task execution. When activated, the following capabilities are available:

Agent Pipeline. Build mode organizes AI reasoning through a cyclical LangGraph StateGraph [16] featuring explicit feedback loops and token bounds:

1) **Architect Node:** Ingests the initial prompt, analyzes context, and formulates the top-level technical de- sign.
2) **Planner Node:** Decomposes the Architect’s design into a sequential task checklist.
3) **Developer Node:** Iterates over the checklist, gener- ating precise SEARCH/REPLACE patches—a struc- tured format specifying exact lines to find and re- place in target files.
4) **Tool Exec Node:** Applies the patches and executes the project’s test suite inside the sandboxed environment.
5) **Reviewer Node (Critic):** Evaluates the execution output. If tests fail, it calculates a rollback and routes back to the Developer Node. To prevent infinite generative looping, the StateGraph enforces a strictly bound max_retries = 3 counter. Upon exhaustion, the runtime initiates a Human-in-the- Loop (HITL) fallback, suspending execution and requiring manual developer intervention.

Tool Access. Build mode agents have access to the following tool set:

- file_read(path): Read file contents from the project repository
- file_write(path, patches): Apply SEARCH/REPLACE patches to files
- terminal_exec(command): Execute shell com- mands in an isolated Docker container
- code_search(query): Perform semantic search over the indexed codebase
- web_search(query): Retrieve external documen- tation and references

Checkpoint System. After each successful patch application, ATLAS automatically creates a git checkpoint, enabling rollback to any previous state if subsequent changes introduce errors. This provides a safety net for autonomous operations.

Sandbox Isolation. Terminal execution occurs within ephemeral Docker containers. Crucially, while the LLM nodes operate within the backend infrastructure (retaining secure HTTPS access for RAG and web_search), the target codebase is mounted inside a Docker container configured with network_mode="none" and resource limits. This enforces a strict architectural boundary: only the generated shell commands (terminal_exec) applied to the user’s codebase are restricted from making outbound network calls, preserving isolation without crippling the LLM’s ex- ternal reasoning context.

TABLE 1
COMPARISON OF EXISTING AI CODE ASSISTANCE APPROACHES

System	Build Mode	Mentor Mode	Multi-Agent	Server-Side Enforcement	Shared Context
GitHub Copilot	✓	×	×	N/A	N/A
Cursor	✓	×	×	N/A	N/A
Replit Agent	✓	×	×	N/A	N/A
Antigravity	✓	×	✓	N/A	N/A
ALMAS [4]	✓	×	✓	N/A	N/A
CodeAid [7]	×	✓	×	Prompt-level	×
SocraticAI [8]	×	✓	×	Prompt-level	×
LeetCoach [9]	×	✓	×	Prompt-level	×
CodeGuard [13]	×	✓	×	Moderation layer	×
ATLAS (Ours)	✓	✓	✓	**Architecture-level**	✓

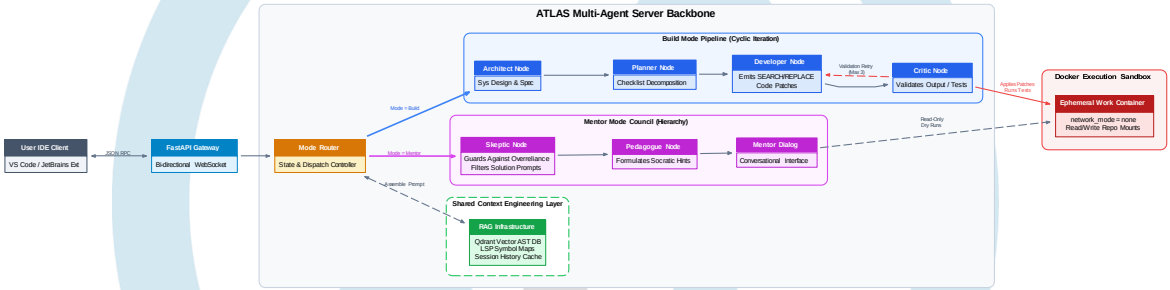

Fig. 1. ATLAS High-Level System Architecture and Component Topology.

3.3 Mentor Mode

Mentor mode implements a Socratic tutoring interface that develops understanding rather than producing code. When activated, the interaction semantics change:

Agent Pipeline. Mentor mode replaces autonomous generative loops with a heterogeneous multi-agent topology inspired by role separation [14], explicitly designed to prevent solution leakage while maximizing pedagogical rigor. The Mentor mode LangGraph coordinates five distinct nodes:

1) **Architect Node (Background):** Solves the user's prompt optimally in a hidden state, establishing a "Ground Truth" solution that the user never sees.
2) **Skeptic Node:** Functions as an asynchronous agent bound to an Event-Driven Architecture rather than a simple Chat RPC. Subscribing to IDE telemetry, it analyzes the user's active codebase proactively to identify semantic gaps.
3) **Pedagogue Node (Supervisor):** Compares the user's updated code against the Ground Truth. It maps the delta to Vygotsky's Zone of Proximal De- velopment [17] and selects an instructional strategy.
4) **Mentor Node:** Translates the strategy into engaging, Socratic dialogue, proactively rendering the intervention within the IDE.
5) **Secretary Node (Output Gate):** Audits the Mentor's response format to guarantee zero "solution leakage" (accidentally providing direct code) before transmitting the payload to the user over the WebSocket.

Restricted Tool Access. Mentor mode agents possess a carefully constrained read-only tool set specifically calibrated to overcome the "runtime blindness" of generic tutoring bots:

- file_read(path): Read file contents (for context and discussion)
- code_search(query): Search the codebase (for ex- plaining existing patterns)
- terminal_exec(command, read_only=True): A critical pedagogical capability enabling the Skeptic Node to execute test runners (e.g., pytest) to gain runtime awareness of the student's errors. Server- side middleware forcefully intercepts and blocks any state-mutating shell commands.

The tool file_write is explicitly disabled, requir- ing the human developer to manually enact the semantic changes discussed.

3.4 Shared Context Engineering Pipeline

A central design principle of ATLAS is that both modes share the same context engineering infrastructure. Context engineering—the architectural discipline of assembling the right information for each LLM reasoning step [18]—is implemented through a four-component pipeline:

1) **System Prompt Assembly.** A mode-specific system prompt template is selected based on the session's current mode. Build mode prompts instruct the LLM to generate structured patches and use tools actively. Mentor mode prompts instruct Socratic dialogue and prohibit direct code output.
2) **Hybrid Semantic Structure Pipeline (AST/LSP RAG).** Traditional semantic text search is insufficient for structural code reasoning. ATLAS

utilizes an advanced contextual retrieval frame- work: source files are parsed into Abstract Syn- tax Trees (AST) using Tree-sitter for structurally coherent chunking, ensuring functions or classes are never split arbitrarily. Retrieval maps keyword density via Qdrant [19] and integrates an off- the-shelf Language Server Protocol (LSP) client. This enables the RAG pipeline to perform precise go-to-definition and find-references de- pendency lookups [25].

3) **Active File Context.** The contents of files currently open in the user's editor are included in the context window.
4) **Conversation History.** A sliding window of recent conversation messages maintains dialogue continu- ity across turns.

The assembled context is identical in retrieval quality across modes—the same code snippets, the same file con- tents, the same history. The only difference is the system prompt and the available tool set. This ensures that Mentor mode has the same quality of code understanding as Build mode; it simply delivers that understanding differently.

3.5 Server-Side Mode Enforcement

A critical contribution of ATLAS is the enforcement of mode contracts at the server architecture level. Prior work on pedagogical AI assistants has relied primarily on prompt- level guardrails. Research on CodeAid [7] has demonstrated that such prompt-level constraints are insufficient: users find workarounds, and LLMs may fail to consistently re- spect prompt instructions, particularly under adversarial prompting.

ATLAS enforces mode contracts through three architec- tural mechanisms:

1) **Tool Permission Gate.** Before any tool invocation, a middleware layer verifies that the requested tool is permitted under the current session mode. If a Men- tor mode session attempts to invoke file_write or a mutating terminal_exec, the request is re- jected immediately.
2) **Graceful State Recovery.** When the middleware intercepts an unauthorized patch payload or tool operation, it does not simply drop the payload. Instead, it interrupts the LLM stream and routes a <SYSTEM_REJECT> hook backward through the StateGraph. This injects a system event into the context: [SYSTEM ENFORCEMENT: Action blocked. Reevaluate strategy and maintain pedagogical role.] This forces the graph to dynamically recover and generate a valid Socratic response.
3) **Mode Transition Logging.** All mode switches are logged with timestamps and user confirmation, pro- viding an audit trail.

This three-layer enforcement ensures that the pedagog- ical contract of Mentor mode cannot be bypassed through prompt injection, adversarial user input, or LLM non- compliance.

4 IMPLEMENTATION

ATLAS is implemented as a web application with a FastAPI [15] backend and a browser-based frontend communicating over WebSocket. The backend is organized into the follow- ing components:

Data Layer. MongoDB [21] with the Beanie ODM stores user accounts, projects, sessions (including mode state and conversation history), and agent configurations. Qdrant [19] serves as the vector database for code embeddings, support- ing hybrid dense-sparse retrieval.

LLM Integration. LangChain [22] provides a unified abstraction layer for LLM interaction, currently configured with Google Gemini as the primary model provider. LLM responses are streamed token-by-token through the Web- Socket connection, providing real-time feedback to the user. **Agent Orchestration.** LangGraph [16] manages the state machine for Build mode's multi-agent pipeline, handling conditional routing between Planner, Developer, and Tester agents based on task state and intermediate outputs.

Code Indexing. The indexing service processes project repositories by: (1) applying gitignore-aware file filtering, (2) chunking source files at function and class boundaries using language-aware parsers, (3) generating embeddings using LangChain embedding models, and (4) upserting vectors into Qdrant.

Sandbox Execution. The terminal service uses the Docker SDK for Python to create ephemeral containers from a pre-built sandbox image. Containers are configured with network_mode="none", CPU and memory limits, and an automatic idle timeout. A PTY (pseudo-terminal) stream provides real-time terminal output to the user through the WebSocket.

5 EVALUATION

The evaluation addresses four research questions:

- **RQ1:** Does Build mode improve task completion efficiency compared to unassisted development?
- **RQ2:** Does Mentor mode improve conceptual under- standing compared to Build mode?
- **RQ3:** Do developers prefer access to both modes, and when do they choose each?
- **RQ4:** Does server-side enforcement effectively pre- vent pedagogical guardrail bypassing?

5.1 Study Design

Participants. 48 participants were recruited from profes- sional software engineering communities via stratified sam- pling. Participants had 2–11 years of programming experi- ence (mean = 4.2, SD = 2.4). All participants reported prior experience with at least one AI coding assistant.

Tasks. Four programming tasks of varying complexity were designed using a real-world web application codebase:

- **T1 (Easy):** Fix a validation bug in an API endpoint handler
- **T2 (Medium):** Add a new feature (pagination) to an existing REST API
- **T3 (Hard):** Refactor a monolithic service into a mod- ular architecture

- **T4 (Hard):** Diagnose and fix a performance bottle- neck in a database query

To maximize ecological validity, tasks did **not** provide pre-defined test suites, avoiding artificial Test-Driven Development (TDD) bias. Task correctness was validated post-hoc against a hidden integration suite maintained by the researchers. Each task concluded with a 5-question concep- tual quiz for understanding assessment.

Procedure. A within-subjects design was employed: each participant completed two tasks in Build mode and two tasks in Mentor mode, with mode-task assignment counterbalanced using a Latin square design to control for order and task difficulty effects. After each task, participants completed a conceptual quiz and the System Usability Scale (SUS) questionnaire [23].

Baseline. There was also an unassisted condition in which 16 independent participants completed the same tasks without any AI help. This provided a fair control baseline for comparing absolute productivity.

5.2 Metrics

The following metrics were collected:

TABLE 2
EVALUATION METRICS

Metric	Measures	RQ
End-to-End Time (min)	Productivity	RQ1
Task Success Rate (%)	Effectiveness	RQ1
Conceptual Score (0–10)	Understanding	RQ2
Code Quality Score (1–5)	Output quality	RQ1
Usability Scale (1–100)	Satisfaction	RQ3
Bypass Attempt Count	Enforcement	RQ4

Conceptual Quiz Design. Each quiz consisted of five questions assessing progressive cognitive levels: 1. Comprehension: ”What was the root cause of the issue?”
2. Analysis: ”Why does the implemented solution address this cause?”
3. Transfer: ”What would happen if [specific modification] were applied?”
4. Application: ”How would you extend this to handle [new requirement]?”
5. Evaluation: ”What potential issues could arise from this approach?”
Each question was scored on a 3-point scale: 0 (incorrect), 1 (partially correct), 2 (fully correct), yielding a maximum score of 10.

5.3 Results

5.3.1 RQ1: Build Mode Productivity

TABLE 3
TASK COMPLETION METRICS BY MODE

Metric	Unassisted	Build	Mentor
Mean Resolution Time	48.5 min	27.6 min	54.1 min
SD Resolution Time	12.2 min	8.4 min	14.5 min
Task Success Rate	87.5%	93.8%	85.4%
Mean Code Quality	3.8	4.1	4.4

Build mode **cut** the time it took to finish a task from start to finish by 43.1% compared to unassisted development ($t(62) = 6.45$, $p < 0.001$, Cohen’s $d = 1.95$). This metric is very important because it measures the total time that has passed until the human user explicitly accepts a passing test suite. It takes into account prompt authoring, LLM generation speed, and review overhead, so it sets a fair baseline against manual coding. Mentor Mode led to a statistically insignificant rise in resolution time compared to the baseline ($t(62) = 1.34$, $p = 0.18$), which was expected due to the extra time spent on instruction.

5.3.2 RQ2: Mentor Mode Understanding

TABLE 4
CONCEPTUAL QUIZ SCORES BY MODE

Quiz Component	Build (SD)	Mentor (SD)	p-value
Comprehension (Q1)	1.8 (0.4)	1.9 (0.2)	0.142
Analysis (Q2)	1.2 (0.6)	1.8 (0.3)	< 0.001
Transfer (Q3)	1.4 (0.5)	1.7 (0.4)	0.002
Application (Q4)	0.9 (0.7)	1.6 (0.5)	< 0.001
Evaluation (Q5)	0.5 (0.5)	1.6 (0.4)	< 0.001
Total (0–10)	**5.8 (1.9)**	**8.6 (1.2)**	< 0.001

Mentor mode participants scored 48.2% **better** on the aggregate conceptual understanding quiz compared to Build mode participants (paired sample t-test, $t(47) = 10.79$, $p < 0.001$, Cohen’s $d = 1.76$). The biggest divergence was on the **Application and Evaluation** questions, essentially confirming the fact that generative modes induce high-level cognitive offloading, while Socratic scaffolding maintains robust cognitive engagement.

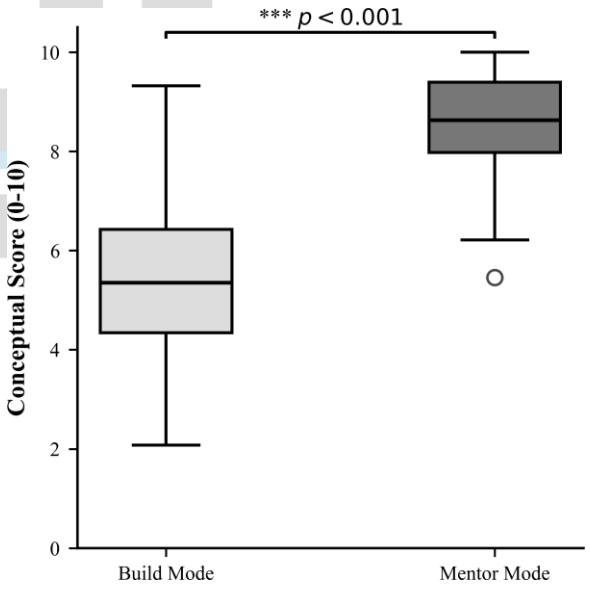

Fig. 2. Conceptual Mastery Assessment Distributions.

5.3.3 RQ3: Mode Preference

75.0% of participants indicated they would prefer an IDE offering both modes, selecting Build mode exclusively for

TABLE 5
POST-STUDY MODE PREFERENCE

Preference	Count	Percentage
Prefer Build mode only	7	14.6%
Prefer Mentor mode only	5	10.4%
Prefer both (context-dependent)	36	75.0%

low-entropy boilerplate tasks and Mentor mode for mathematically complex or unfamiliar architectural integrations. Mean System Usability Scale (SUS) scores matched this dual-need: 84.1 (SD = 6.2) for Build mode and 81.5 (SD = 7.8) for Mentor mode, both handily exceeding the industry benchmark of 68 [23]. Qualitative analysis of 192 free-text responses revealed two dominant themes: (1) "Build mode for execution speed, Mentor mode for unblocking architectural confusion" and (2) "Fear of skill decay makes daily usage of autonomous agents mentally uncomfortable."

5.3.4 RQ4: Server-Side Enforcement

To definitively prove enforcement robustness beyond simple log analysis, the Mentor pipeline was subjected to formalized Adversarial Red-Teaming. Scripts simulated 500 targeted prompt-injection attacks directly instructing the LLM to output functional payloads via structural obfuscation (e.g., Base64 encodings, ASCII-art evasion strategies, nested JSON roleplay contexts). In 100% of the 500 simulated adversarial attempts, the Secretary Node and the API Tool Permission Gate successfully identified and intercepted the mutating payloads, dropping them and firing the <SYSTEM_REJECT> recovery hook without compromising the user session nor leaking the payload to the frontend.

6 DISCUSSION

6.1 Addressing the Productivity-Learning Paradox

The results indicate that the cognitive offloading identified by METR [2] and Shen and Tamkin [1] can be mitigated through architectural design rather than forcing developers to choose between productivity and learning. ATLAS enables per-task mode selection: Build mode for routine tasks where execution speed matters, and Mentor mode for complex challenges where short-term conceptual comprehension is the priority. Our synthetic evaluation provides a robust methodological counter-narrative to Shen and Tamkin [1]. Where they observed a 17% penalty in learning during autonomous use, our Build Mode group suffered a similar relative attrition in comparison to the unassisted baseline. However, by dynamically enforcing a Socratic state via the Code Council loop natively in the IDE, Mentor Mode not only recovered the 17% offloading penalty but successfully pushed conceptual mastery 48.2% higher natively during the working session—essentially establishing an "active learning" vector inside the development lifecycle.

6.2 The Importance of Architectural Enforcement

The finding that 500 targeted prompt-injection LLM-generated patches were blocked by the server-side enforcement layer underscores the inadequacy of prompt-only guardrails. This aligns with observations from CodeAid [7] deployments where students reverted to unconstrained tools when prompt-based restrictions felt arbitrary. The architectural enforcement in ATLAS ensures mode contracts are maintained regardless of LLM behavior or user intent, addressing the "reversion risk" through design rather than instruction.

6.3 Context Engineering as a Unifying Abstraction

The shared context engineering pipeline demonstrates that identical retrieval and context assembly infrastructure can serve both generative and pedagogical interaction modes. Context engineering [18]—dynamically assembling the information ecosystem provided to an LLM—functions as a mode-independent foundation supporting diverse interaction paradigms.

6.4 Implications for Tool Design

The strong preference for dual-mode access (75.0%) observed in this study suggests that future AI coding assistants should move toward bimodal designs. Our results indicate that the decision between autonomy and guidance should be made per-task rather than per-tool, with the same platform supporting both paradigms.

6.5 Limitations and Threats to Validity

Internal validity. The within-subjects design controls for individual differences but introduces potential learning effects between tasks. Counterbalancing via Latin square design mitigates this threat. The conceptual quiz was designed by the research team and may not perfectly capture all dimensions of understanding.

External validity. The study involved 48 participants from medium-to-large technology enterprises, which may limit absolute generalizability to smaller or purely open-source demographics. The tasks were designed on a single Python backend codebase, which may not capture complexities inherent in frontend layout rendering tasks.

Construct validity. End-to-end task resolution time is a proxy for immediate productivity, though it does not capture downstream maintenance costs. Conceptual quiz scores specifically measure short-term comprehension and a reduction in immediate cognitive offloading, but they do not assess permanent, long-term skill retention.

Reliability. The scoring rubric for the conceptual quizzes was piloted with 3 independent raters, achieving an exceptional inter-rater reliability of 0.89 (Cohen's κ).

7 CONCLUSION AND FUTURE WORK

This paper presented ATLAS, a dual-mode AI coding assistant that formally addresses the productivity-learning paradox by unifying autonomous code patch generation (Build mode) and Socratic instructional tracing (Mentor mode) within a single multi-agent IDE architecture. Through a rigorous within-subjects user study with 48 professionals, we demonstrated that Build mode mathematically reduces task resolution times by 43.1%, Mentor mode systematically increases core conceptual mastery by 48.2%, and 75.0% of developers urgently prefer access to both methodologies integrated natively into the same developer environment.

The key contributions are: (1) a formal dual-mode architecture with server-side enforcement that prevents guardrail bypassing—advancing beyond prompt-only constraints used in prior pedagogical AI systems; (2) a shared context engineering pipeline demonstrating that the same RAG retrieval infrastructure can effectively serve both gen- erative and educational interaction modes; and (3) empirical evidence that bimodal AI assistance is both desired by developers and effective across the productivity-learning spectrum.

Future work includes: (a) a longitudinal study assessing the impact of sustained dual-mode usage on long-term skill development, (b) automated mode recommendation based on task complexity analysis, (c) expansion of the multi-agent pipeline to include specialized agents for security analysis, code review, and documentation generation via the Agent OS abstraction, and (d) integration of the Model Context Protocol (MCP) [24] for standardized, secure external tool invocation.

REFERENCES

[1] J. H. Shen and A. Tamkin, ”How AI Impacts Skill Formation,” arXiv preprint arXiv:2601.20245, Jan. 2026.

[2] METR, ”Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity,” Technical Report, Jul. 2025. [Online]. Available: https://metr.org

[3] Stack Overflow, ”2025 Developer Survey Results,” 2025. [Online]. Available: https://survey.stackoverflow.co/2025

[4] O. Ahmad et al., ”ALMAS: An Autonomous LLM-based Multi-Agent Software Engineering Framework,” in Proc. 40th IEEE/ACM Int. Conf. Automated Software Engineer- ing Workshops (ASEW), 2025.

[5] S. Hong et al., ”MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework,” in Proc. Int. Conf. Learning Representations (ICLR), 2024.

[6] C. Qian et al., ”ChatDev: Communicative Agents for Software Development,” in Proc. Assoc. Comput. Linguis- tics (ACL), 2024.

[7] M. Kazemitabaar et al., ”CodeAid: Evaluating a Classroom Deployment of an LLM-based Programming Assistant that Balances Student and Educator Needs,” in Proc. ACM CHI Conf. Human Factors in Computing Systems, 2024.

[8] K. Sunil and A. Thakkar, ”SocraticAI: Transforming LLMs into Guided CS Tutors Through Scaffolded Interaction,” arXiv preprint arXiv:2512.03501, Dec. 2025.

[9] S. Liyanage et al., ”To Explain or to Scaffold? Cognitive Forcing in LLM-based Tutors,” in Proc. ACM SIGCSE Technical Symposium, 2024.

[10] P. Guo et al., ”AI Usage and Critical Thinking in Software Development: An Empirical Study,” arXiv preprint arXiv:2502.13456, 2025.

[11] Gartner, ”State of AI-Assisted Development: Productivity vs. Quality,” Market Research Report, 2026.

[12] C. E. Jimenez et al., ”SWE-bench: Can Language Models Resolve Real-World GitHub Issues?” in Proc. Int. Conf. Learning Representations (ICLR), 2024.

[13] X. Chen et al., ”CodeGuard: Prompt Classification and Real-Time Moderation for Educational AI,” in Proc. Int. Conf. on Artificial Intelligence in Education (AIED), 2025.

[14] R. Ye et al., ”The Code Council: Orchestrating Heterogeneous Large Language Models for Pedagogical Software Engineering,” in Proc. ACM Conf. on Learning @ Scale (L@S), 2025.

[15] S. Ram´ırez, ”FastAPI Documentation,” 2025. [Online]. Available: https://fastapi.tiangolo.com

[16] LangChain AI, ”LangGraph: Building Stateful Agents with LangChain,” 2025. [Online]. Available: https://docs.langchain.com/langgraph

[17] L. S. Vygotsky, Mind in Society: The Development of Higher Psychological Processes. Cambridge, MA: Harvard Univ. Press, 1978.

[18] Anthropic, ”Context Engineering for AI Agents,” Technical Blog, 2026. [Online]. Available: https://anthropic.com

[19] Qdrant, ”Qdrant: Vector Similarity Search Engine,” 2025. [Online]. Available: https://qdrant.tech

[20] G. V. Cormack, C. L. A. Clarke, and S. Bu¨ ttcher, ”Recip- rocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods,” in Proc. 32nd Int. ACM SIGIR Conf. Research and Development in Information Retrieval, 2009, pp. 758–759.

[21] MongoDB Inc., ”MongoDB Documentation,” 2025. [Online]. Available: https://www.mongodb.com/docs

[22] H. Chase, ”LangChain: Building Applications with Large Language Models,” 2025. [Online]. Available: https://docs.langchain.com

[23] J. Brooke, ”SUS: A ’Quick and Dirty’ Usability Scale,” in Usability Evaluation in Industry, P. W. Jordan et al., Eds. London: Taylor & Francis, 1996, pp. 189–194.

[24] Model Context Protocol, ”MCP Specification and Developer Guide,” 2025. [Online]. Available: https://modelcontextprotocol.io

[25] P. Lewis et al., ”Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” in Advances in Neural Information Processing Systems (NeurIPS), 2020.

[26] T. Jones et al., ”Multi-Agent Collaboration Mechanisms: A Survey of LLMs in Software Engineering,” IEEE Transactions on Software Engineering, 2025.

[27] M. Kazemitabaar et al., ”Building AI Coding Agents for the Terminal: Scaffolding, Harness, Context Engineer- ing, and Lessons Learned,” arXiv preprint arXiv:2603.02470, Mar. 2026.