Professor + AI Team: A Rapid-Iteration, Low-code Development Paradigm for Educational Computing Innovation

Dr. Xiaoguang Ma, University of Wisconsin - Platteville

Dr. Xiaoguang Ma is an Associate Professor in the Department of Electrical and Computer Engineering at the University of Wisconsin-Platteville, where he has taught undergraduate courses since joining in 2018. He earned his Ph.D. from Florida State University, an M.S. from the Chinese Academy of Sciences, and a B.S. from Tsinghua University. Prior to academia, Dr. Ma worked as a Communication Architect at ABB Inc., specializing in industrial communication protocols and cybersecurity, and holds a 2018 patent for ”Parallel Redundancy Protocol over Wide Area Networks.” His current research explores smart microgrid cybersecurity and blockchain applications for data security in agriculture, collaborating on projects like a cybersecurity testbed and dairy farm data protection at UW-Platteville’s Pioneer Farm. Dr. Ma’s work bridges education and innovation, advancing solutions for critical infrastructure and technology security.

Jing Wang

Jing Wang is an Instructional Designer with 19 years of experience developing scalable, data-driven learning programs across higher education and Fortune 500 companies. Currently an Instructional Designer at the Universities of Wisconsin System, she specializes in blending AI, analytics, and instructional systems design to create learner-centric solutions that deliver measurable impact. With a background in economics and a Master’s Degree in Instructional Design from Florida State University, Jing bridges the gap between analytical strategy and innovative pedagogy. Her recent work emphasizes using AI automation to elevate digital accessibility and streamline the development of complex STEM courses, ultimately building future-ready learning ecosystems.

Professor + AI Team: A Rapid-Iteration, Low-code Development Paradigm for Educational Computing Innovation

Abstract

The integration of generative AI into engineering curricula is often constrained by a trade-off between operational efficiency and pedagogical governance: faculty want faster ways to build and update instructional tools, but also need clear oversight of accuracy, structure, and instructional intent. The emergence of "vibe coding," using natural language to generate executable tools and workflows with AI assistance, can reduce implementation friction and expand what educators can prototype on their own. However, using current AI models as the sole implementation engine for large systems can introduce practical risks around reliability, maintainability, cost, and data governance. As a result, many academic implementations confine AI to post-hoc assistance for discrete coding tasks, limiting educator control over system architecture.

To reduce this efficiency–governance trade-off, this research operationalizes the convergence of three developments: code-centric low-code workflow platforms, modern LLMs that generate reliable code when tasks are narrowly scoped, and agentic AI methodologies that decompose complex objectives across specialized assistants.

Building on these developments, we present the "Professor + AI Team" model, a faculty-led, Agile-inspired protocol in which the educator acts as product owner and technical lead, coordinating a Development-Layer AI team for architecture reasoning, targeted implementation, debugging, and example search. We report an auto-ethnographic engineering case study in which a single faculty member, without prior experience in JavaScript, n8n, or vibe coding, successfully architected a 200+ node agentic workflow for curriculum material generation. The case shows that a domain expert can learn and operationalize these tools during development when the workflow is decomposed into inspectable stages, bounded code tasks, and mandatory human-in-the-loop checkpoints.

The protocol emphasizes deterministic scaffolding around probabilistic AI outputs, structured task sequencing, artifact-first state management, and mandatory human checkpoints. Persisted artifacts, including planning documents, manifests, and reusable professor-persona files, function as workflow memory and support cumulative refinement across runs. The protocol also supports selective re-entry and scope adaptation: because intermediate artifacts are persisted as editable files, the educator can modify specific elements, such as adding a learning outcome to the planning artifact or inserting a slide into the slide and notes files, and then rerun only the relevant downstream stages instead of regenerating the entire workflow from the beginning.

Using this approach, we observed a substantial reduction in time to a minimum viable product, from an estimated 8-13 months for a standard two-person team to approximately six weeks, and we demonstrated replication in a second tool with smaller scope. We also documented an emergent outcome we call "reciprocal competence": iterative debugging improved the educator's ability to specify constraints, verify outputs, and integrate components. By empowering educators to act as AI orchestrators rather than routine code authors, this work contributes a reproducible blueprint

for expert-led educational computing innovation.

1. Introduction

Engineering educators routinely translate emerging technologies into teachable experiences, designing new labs, demonstrations, simulations, and software tools that help students connect theory to practice. However, transforming a pedagogical idea into a reliable, reusable software artifact often requires full-stack development (data storage, secure APIs, user interfaces, testing, and deployment) that competes directly with teaching, research, and service. When implementation is shifted to centralized IT or outsourced teams, progress is frequently constrained by long delivery cycles, high costs, and repeated translation between educational intent and technical specifications. Even when a custom tool is delivered, maintaining and upgrading it can be challenging, leading to stagnation or "abandonware" once funding exhausts or key student developers graduate.

1.1 The Emergence of "Vibe Coding"

The rapid evolution of Large Language Models (LLMs) has given rise to a new development paradigm often described as "vibe coding." Unlike traditional programming, which requires manual authoring of precise syntax and logic, vibe coding allows a user to describe desired intent or functionalities in natural language, relying on AI to generate the underlying implementation[1]. Recent reports characterize this shift as moving from code authoring toward code validation, enabling domain experts to prototype solutions substantially faster than traditional methods[2,3,4].

However, for academic domain experts (such as Engineering faculty), relying solely on unstructured vibe coding introduces engineering risks that often stall development beyond the prototype phase. The primary challenges are determinism and maintainability: while LLMs can generate isolated code snippets effectively, they often struggle to preserve structural coherence across large systems, producing brittle "spaghetti code" that is difficult for non-professional developers to debug[5,6]. In addition, common vibe coding workflows lack built-in guardrails for data security, error handling, and scalability, creating a "black box" where educators cannot readily verify how student data is processed or how edge cases are handled. A non-coder may successfully prompt an AI to "build a curriculum generator," but still lack the technical vocabulary to specify the architectural constraints (e.g., recursion limits, input validation, API rate limiting, and recovery behaviors) required for reliability[7]. Consequently, without a deterministic structural framework, the "vibe" can collapse under real-world complexity.

1.2 The Landscape of Low-code Platforms

Parallel to the rise of generative AI is the maturation of Low-code/No-code (LCNC) platforms, which have evolved from simple automation tools into sophisticated development environments capable of powering mission-critical applications. When applied to the specific requirements of AI-augmented educational software, this landscape offers distinct categories of tools, each with specific trade-offs:

No-Code Tools (e.g., Make, Zapier): These platforms excel at simple linear integrations between SaaS applications, such as sending a form submission to a spreadsheet. Yet, they

often lack the granular control required for complex data processing or algorithmic decision-making. Their "black box" nature can make debugging complex pedagogical flows, such as recursive feedback loops, difficult or impossible.

AI-Native Builders (e.g., Langflow, Flowise): These newer platforms are specialized for "chaining" LLM prompts and are excellent for rapid prototyping of conversational agents and Retrieval-Augmented Generation (RAG) pipelines. However, industry reports indicate that they can be brittle when handling deterministic business logic or maintaining complex state across long workflows, making them less suitable for robust enterprise-grade applications[8]. They are optimized for probabilistic outputs rather than the deterministic reliability required for curriculum logic.

Code-Centric Low-code Platforms (e.g., n8n): This represents a distinct category that combines visual orchestration with developer-grade control. Best exemplified by n8n, these platforms utilize a node-based architecture where distinct functions (nodes) are connected via wires to pass data strictly as JSON objects. In particular, n8n explicitly supports embedding custom JavaScript within workflows for complex data manipulation and integration tasks. This architecture blends visual orchestration with deterministic programmability, preserving low-code transparency without sacrificing engineering rigor[9].

To illustrate this capability, Figure 1 demonstrates a typical hybrid workflow in n8n. The process initiates with a Trigger Node (e.g., a form submission), passing data to a probabilistic AI Agent Node for reasoning. The output is then routed through a deterministic Logic "IF" Node before reaching specific downstream steps that execute a "Code" node for the final task. These Code nodes serve as the deterministic targets for vibe coding, where the educator generates the precise JavaScript required for execution.

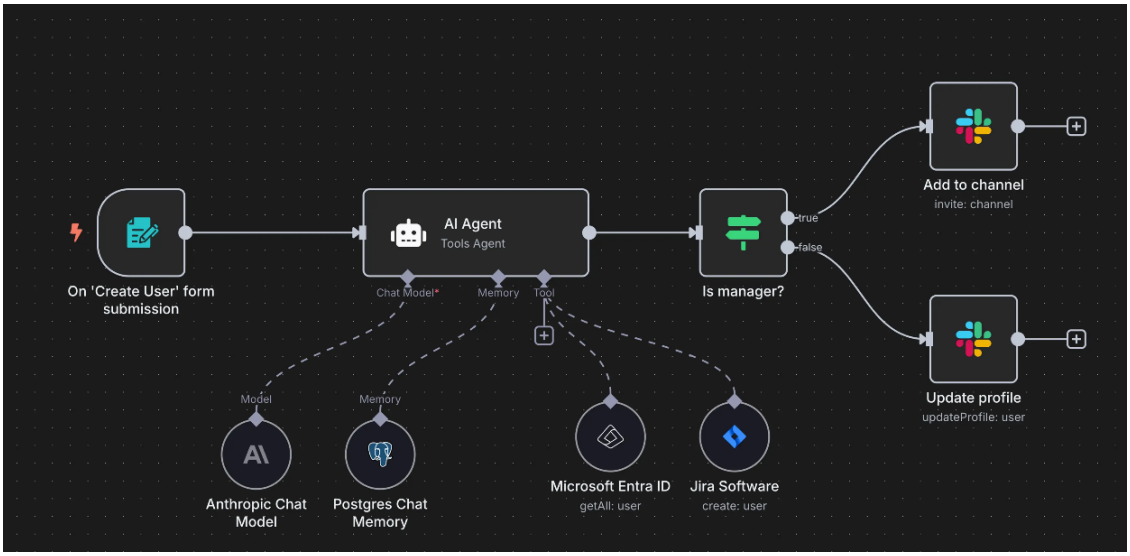

Figure 1: An example n8n workflow showing the hybrid architecture.(Source: n8n.io)

1.3 The Rise of Agentic AI

While code-centric low-code platforms provide the necessary deterministic structure, traditional automation workflows remain insufficient for the complex, adaptive reasoning required in engineering active learning systems. To elevate these platforms from static data pipelines to intelligent cognitive architectures, we must integrate agentic AI. Unlike standard generative AI, which is typically reactive (responding to a single prompt with a single output), agentic AI is defined by its autonomy and goal-oriented behavior[10]. This shift allows the system to move beyond simple linear automation to complex, non-linear problem solving.

An agentic system does not simply generate text; it perceives its environment, reasons through complex multi-step problems, uses external tools (such as web search or code execution), and iteratively reflects on its own output to improve quality[11]. Leading frameworks characterize this approach through four key design patterns: Reflection, Tool Use, Planning, and Multi-Agent Collaboration[12]. Existing literature has applied these concepts to active learning simulations, where prompt engineering was used to create isolated agentic scenarios for computer networks education[13]. However, when applied to complex pedagogical tasks, such as the automated curriculum generation system developed in this study, this approach allows for the decomposition of a monolithic task (e.g., "create a dynamic coding lab") into discrete sub-problems handled by specialized agents. For instance, a "Planner Agent" may outline the learning objectives, a "Researcher Agent" synthesizes technical content from verified sources, and a "Reviewer Agent" audits the generated material against strict pedagogical constraints to ensure alignment with learning goals[14]. This division of labor mimics a human development team, significantly reducing hallucination rates by narrowing the context window for each specific agent[15].

1.4 Challenges in AI-Assisted Development

While the individual technologies of vibe coding, agentic AI, and low-code platforms offer promise, their integration into academic engineering presents four distinct challenges:

Conflicting Pedagogical Requirements (The Model Problem): Effective engineering education often requires balancing competing demands: materials should support engagement (e.g., active learning) while remaining technically precise and free of ungrounded claims. In practice, a single model configuration may struggle to optimize both creativity and factual rigor at the same time; prompts that encourage novelty can increase the risk of inaccuracies, whereas stricter constraints may reduce pedagogical richness. These tensions motivate a role-specialized, multi-agent workflow in which different agents (e.g., a "Creative Director" and a "Technical Auditor") operate with distinct objectives and parameters[16].

The "Customization Ceiling" (The Platform Problem): In our implementation, the main bottleneck was usually not whether the workflow could represent the intended pedagogical logic, but whether the platform's embedded runtimes and user-interface surfaces could support it. For example, we encountered cases where LLM-generated JavaScript relied on libraries or module-loading behaviors that were not available in the n8n execution environment, requiring refactoring to platform-supported alternatives. We also found that some built-in human-in-the-loop (HITL) interfaces (e.g., form-based checkpoints) offered limited

control over layout (such as fixed-width display), which constrained how much review context could be shown to the instructor at once. These boundaries created a practical "ceiling" where progress depended on engineering around platform runtime and UI constraints rather than adding new workflow logic. Because the platform is actively evolving, we expect some of these constraints to relax over time as supported runtimes and UI capabilities expand.

Contextual Collapse in Large Systems (The Complexity Problem): "Vibe coding" can work well for small, self-contained scripts, but we observed recurring failure modes as system complexity increased. As constraints accumulated (multiple routes, branching logic, data contracts, formatting rules, and error-handling behaviors), the model tended to optimize locally for the most recent prompt rather than maintaining global coherence across the system. In practice, this appeared as a cyclical debugging pattern: a patch that fixes "Route A" introduces a regression in "Route B," and the next patch drifts back toward the earlier failure. This behavior is consistent with the fact that LLMs primarily perform high-dimensional pattern matching on text: they often produce syntactically polished code while exhibiting weaker engineering resilience under integration, edge cases, and evolving requirements.

The problem can be amplified in low-code environments due to partial observability: significant logic may reside in visual wiring, node configuration, and platform-specific settings that are not fully represented in the text given to the model. We found that progress required deliberate human intervention to increase observability and constrain the search space, for example, providing screenshots of workflow logic, supplying concrete input/output JSON examples and reference variables, restating constraints and invariants explicitly, proposing debugging hypotheses, and reusing successful patterns from community examples or prior projects. In several cases, comparing suggestions across multiple LLMs and incorporating "lessons learned" artifacts helped break a "Sisyphean" loop in which the model confidently proposed changes that repeatedly failed in the full workflow context.

Vendor Lock-In, Data Privacy, and Operational Reliability (The Privacy Problem): Many commercial educational AI platforms operate as closed SaaS ecosystems. For universities, this can introduce recurring costs and raise compliance concerns (e.g., FERPA) when student-related content is processed outside institutional control. In addition, reliance on a vendor-managed roadmap creates a risk that critical instructional infrastructure may be disrupted by price changes, feature deprecations or policy shifts.

Self-hosting and local processing can mitigate some privacy and lock-in concerns, but our experience showed that it also introduced a distinct operational burden: hardware failures, stack updates (e.g., Docker and container dependencies), platform upgrades (e.g., n8n version changes), and the need for disciplined backups. In one incident, an unexpected shutdown corrupted the Docker-hosted n8n data on a development laptop, highlighting the fragility of relying on a single development environment without explicit recovery planning.

To address these challenges, we used a faculty-led workflow approach: n8n keeps the system structure inspectable, while role-based AI assistance fills in only narrowly scoped, platform-specific code steps. We evaluated the approach by building two tools with the same framework: a curriculum-

development workflow and a paper-to-presentation generator.

The remainder of this paper is organized as follows. Section 2 details the methodology, justifying the selection of the n8n platform and defining the twelve-step "Professor + AI Team" development protocol. Section 3 presents the results of the case study, quantifying the acceleration in development velocity and analyzing the pedagogical quality of the generated curriculum, while also discussing the emergent phenomenon of reciprocal competence. Finally, Section 4 concludes with a summary of contributions and outlines future directions.

2. Methodology

This work presents a faculty-led, auto-ethnographic engineering case study of building low-code agentic AI workflows for instructional artifact generation. We report two implementations developed in the same setting: POSED (Plan, Outline, Sources, Evaluation, Draft), a curriculum-development workflow that produces lecture slides, instructor notes, and related materials; and a paper-to-presentation workflow that converts recent publications into teachable slide decks for classroom introduction and study. The second implementation functions as an internal replication to test whether the workflow patterns and development protocol transfer beyond a single project. The methodology centers on two strategic choices: selecting a platform architecture that supports structured task sequencing, inspectable orchestration, and explicit session-state management, and applying a "Professor + AI Team" protocol that uses AI assistance to accelerate implementation while maintaining instructor control.

In both implementations, the workflow is organized as repeatable subflows with artifact-based state management, where intermediate outputs are written to files (e.g., Markdown artifacts plus a session manifest) rather than carried as large in-memory payloads between steps. This design maintains a persistent working context across stages while reducing state-coupling between subroutines. Dynamic prompts are constructed from the current workflow state, approved artifacts, and task-specific requirements, allowing each subflow to operate with bounded context rather than relying on long conversational histories. Mandatory pause/approval checkpoints are used to enforce human oversight and to create practical debugging points. Together, these design choices provide structured planning and decomposition, explicit state management, curated context assembly, and restartable execution at the subflow level, supporting rapid iteration during development.

Data Captured. Evidence for this case study includes (i) n8n workflow JSON exports for both implementations, used as a structural proxy for workflow complexity and modularity, (ii) execution logs and iteration notes collected during debugging, and (iii) reusable engineering artifacts produced during development, such as prompt-engineering templates, loop-handling patterns, orchestrator planning notes, and recovery/restart procedures.

For the POSED workflow, the broader intended scope was the development of full course materials as specified in the planning outline. In principle, this includes planning artifacts, lecture materials, student handouts, and assessments. However, due to time limitation, this project delivered and analyzed only the planning artifacts and lecture materials, including the plan, outline, professor persona, reference-material collection, lecture slides, and lecture notes. Student activities,

student handouts, and assessments were intentionally left for future development. For the lecture-generation workflow, content volume and student level were designed to be customizable through planning inputs such as course context and lecture/activity time allocation, rather than being fixed in the workflow architecture.

The paper-to-presentation workflow was deployed to generate a conference presentation and to support broader teaching and research use cases, including translating recent publications into teachable slide decks for classroom introduction, self-study, and sharing with research assistants and colleagues. This field-use deployment also motivated the addition of a pacing timer to support time-aware delivery. We also incorporated formative design feedback from an instructional designer on selected outputs. The feedback emphasized cognitive load management, contextual anchoring to the target audience, facilitator support (e.g., instructor notes), and opportunities for active learning.

Validation Approach. In this study, validation refers to how generated artifacts were checked before being accepted for downstream use or reported as usable outputs. Generated content was not accepted on the basis of model output alone. Workflow-wide validation relied on deterministic structure checks, mandatory professor review-and-approve checkpoints, and formative review of selected outputs by an instructional designer. Deterministic checks enforced expected structures, file contracts, and stage-level completeness before outputs could advance downstream. Professor checkpoints were used throughout the workflow to verify technical accuracy, instructional fit, and readiness for further generation. Instructional-designer review focused on cognitive load, pacing, audience fit, facilitator support, and opportunities for active learning.

A fourth check, an AI reviewer agent, was used selectively only during the draft stage for long-form instructional outputs such as slides and speaker notes. Its purpose was to pre-screen content for likely reliability issues, unsupported claims, and pedagogical weaknesses before professor review. This step was not applied to earlier workflow stages because those stages produced shorter artifacts whose quality depended primarily on direct user feedback and iterative professor input.

Analysis Approach. In this study, analysis refers to how the collected development evidence was interpreted to derive workflow-level findings. We analyzed development artifacts by (i) coding recurring failure modes, such as malformed structured outputs, divergent revision loops, state-coupling between subroutines, and operational faults in self-hosted deployment, and (ii) mapping each failure mode to workflow controls intended to mitigate it, such as "AI for text, code for JSON," file/Markdown-based state contracts, manifest-driven session control, and mandatory HITL gates.

Scope and Transferability. This paper reports a faculty-led implementation and distills practical workflow patterns that were effective in our setting. The contribution is therefore methodological and engineering-oriented: it concerns how the workflows were developed, governed, debugged, and made usable, rather than a formal classroom-outcomes evaluation. Because low-code platforms and LLM capabilities evolve quickly, we present these findings as reusable design practices and lessons learned, intended to help other educators adapt the approach to their own constraints, toolchains, and institutional environments, rather than as fixed performance guarantees.

2.1 Strategic Selection of the Low-Code Platform

We selected n8n as the architectural foundation for this system after a comparative analysis against other platforms like Langflow and Make. This decision was driven by five critical academic and engineering requirements:

Deterministic Logic via Code Nodes: Unlike pure no-code platforms that abstract away all logic into pre-set menus, n8n treats JavaScript as a native component. This feature is critical for our architecture. We utilize the platform's code nodes to handle decisive logic, loops, if/else switches, array manipulation, and strict data transformation, while reserving the AI agents for creative work. This separation of concerns prevents the "black box" unpredictability common in pure AI applications.

Rigorous HITL Capabilities: Academic materials require absolute professor oversight. The POSED system integrates non-negotiable faculty checkpoints using n8n's Form Nodes. These nodes pause the workflow to render the intermediate content as rich, readable HTML directly in the browser. They simultaneously collect structured feedback via text fields and multiple-choice selectors, ensuring that the workflow resumes only after the domain expert has explicitly reviewed and approved the material.

Open Architecture and Data Sovereignty: n8n supports a self-hosted implementation. We deployed the system locally using Docker, ensuring that sensitive course data and intellectual property remained within the institution's control.

Built-in Agentic Orchestration and Model Agnosticism: The platform's modular design functions as a "Main Orchestrator," capable of delegating tasks to specialized sub-workflows. Crucially, n8n permits the granular assignment of different LLM backends to specific nodes, including local models via Ollama. This flexibility allowed us to select the optimal model for each agentic role based on specific performance needs and privacy restrictions, blending high-intelligence commercial APIs with cost-effective local inference within a single workflow.

Vibrant Community and Ecosystem: The platform is supported by a rapidly expanding global user base that actively contributes to its development. This ecosystem provides a rich library of community-built nodes, shared workflow templates, and extensive troubleshooting resources on public forums. This collective knowledge base significantly accelerated our development process, allowing us to adapt existing, proven workflow patterns rather than engineering every component from scratch.

2.2 The "Professor + AI Team" Agile-Inspired Development Protocol

To overcome the "implementation gap," this project validated a collaborative development model where the domain expert leads a fast-iteration, Agile-inspired workflow. Figure 3 summarizes the four phases and twelve steps of the protocol, including the escalation loop used when debugging does not converge.

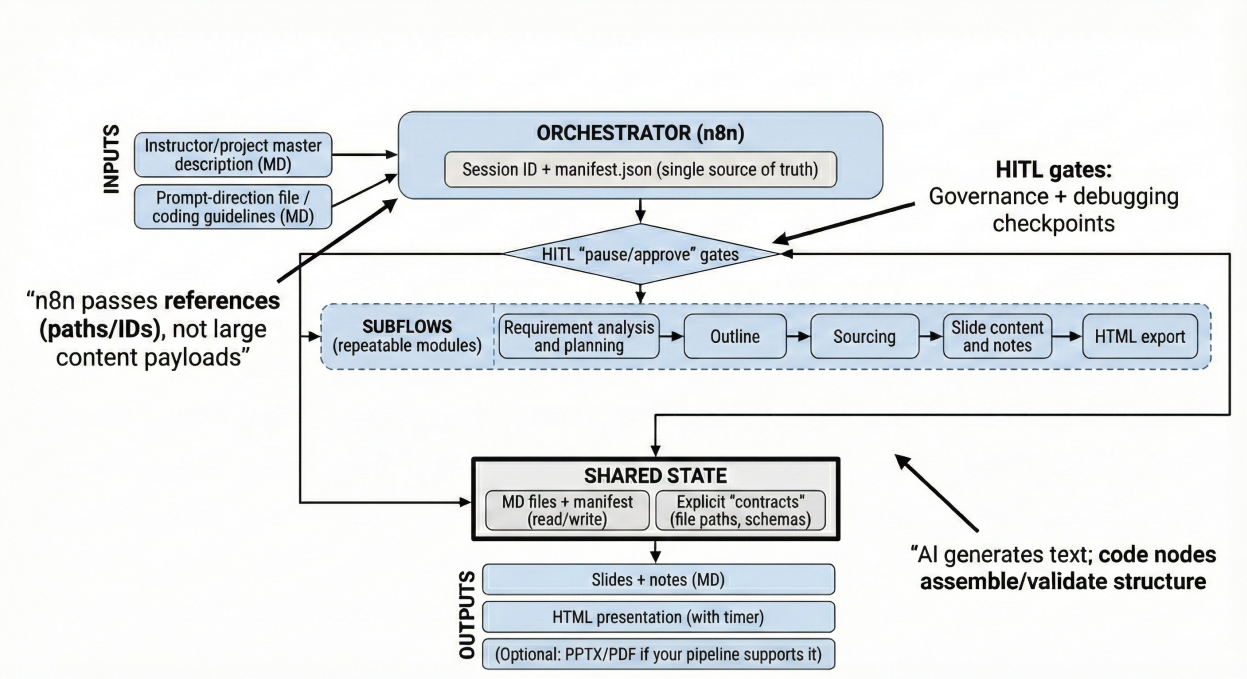

Figure 2: Artifact-first workflow architecture with file-based contracts and HITL governance.

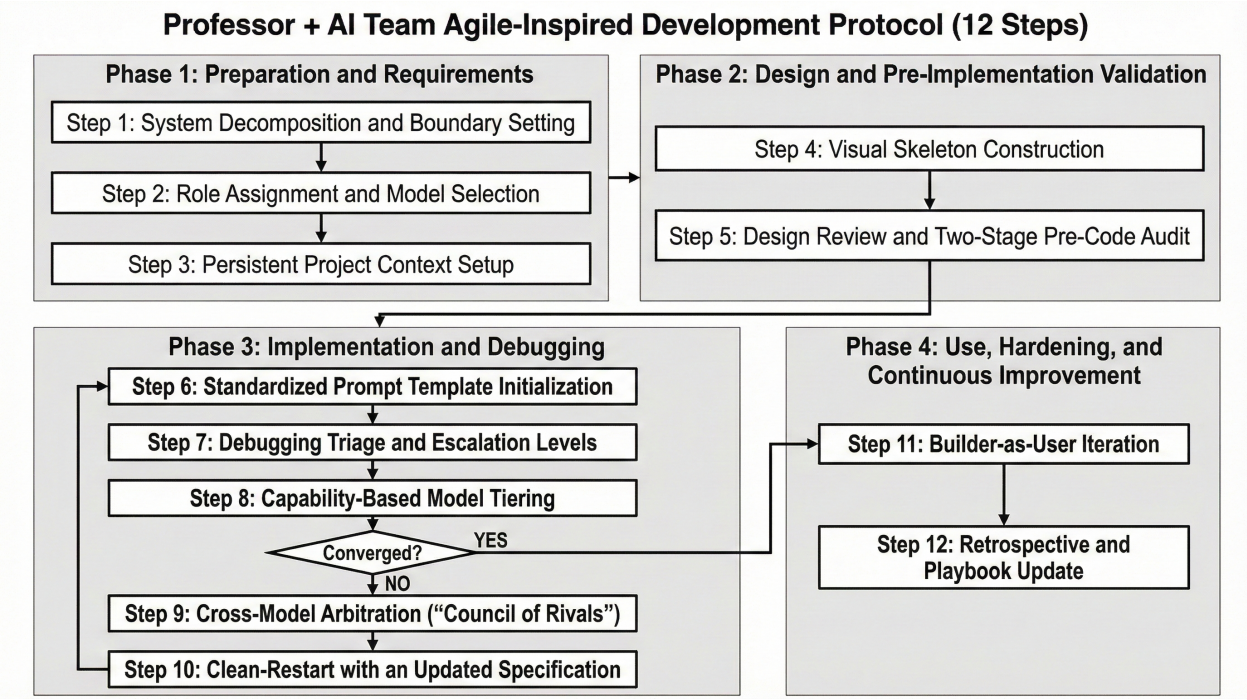

Figure 3: Overview of the four-phase, twelve-step "Professor + AI Team" protocol. The diagram highlights the escalation path (cross-model arbitration and clean restart) used when debugging fails to converge, and the transition into use-driven hardening and retrospective updates.

2.2.1 Phase 1: Preparation and Requirements

Step 1: System Decomposition and Boundary Setting. To prevent the AI from losing coherence or hallucinating due to context overload, we separated the overall framework from specific implementation details. The system was decomposed into discrete functional stages, such as Blueprint Generation and Persona Synthesis (see Figure 2).

Step 2: Role Assignment and Model Selection. We utilized a Development-Layer AI Team for system architecture, implementation, debugging support, and rapid example-finding, treating the models as distinct employees with specialized job descriptions. Role assignments were pragmatic and capability-driven during the development period: we selected models based on which most consistently performed each task well at that time, recognizing that these relative strengths may shift as models and platforms evolve.

- **Strategic Partner & Architect (Gemini):** This agent served as the primary design partner. Rather than writing code, we used Socratic dialogue to define the architecture step by step. Discussions focused on validating workflow topology, defining data governance strategies (inputs/outputs for each node), selecting appropriate n8n nodes, and drafting functional specifications for the AI agents.

- **Implementation Specialist (Claude):** Once design decisions and data flow were established, implementation details were handed off to this agent. We used it to generate the JavaScript required for n8n Code Nodes and to draft precise JSON configurations when needed.

- **Debugging Consultant (ChatGPT):** When failures occurred, we used this agent to propose debugging hypotheses, suggest targeted tests, and provide a second opinion on error interpretation. It also assisted with drafting user-facing documentation (e.g., operating procedures and recovery steps) to improve maintainability.

- **Research Assistant (Grok):** We used this agent to quickly surface candidate examples (e.g., community patterns) and potential platform constraints (e.g., runtime or UI limitations) for subsequent verification and incorporation into the workflow design.

Step 3: Persistent Project Context Setup. To reduce repetitive prompting and keep implementation aligned with the actual execution environment, we leveraged the *Project* feature in the Claude interface used by our Implementation Specialist (Figure 4). Unlike a standard chat thread, this feature allowed us to maintain a persistent project context. The uploaded context package included the system's `docker-compose` configuration, global environment variables (`.env`), and development artifacts we created to stabilize implementation (e.g., a best-practices guide for complex nodes, loop-handling patterns, orchestrator planning notes, and documented backup/recovery procedures). Providing this persistent context improved consistency across iterations by grounding generated code in the local deployment constraints and established conventions, while still requiring human verification and targeted debugging when platform-specific edge cases emerged.

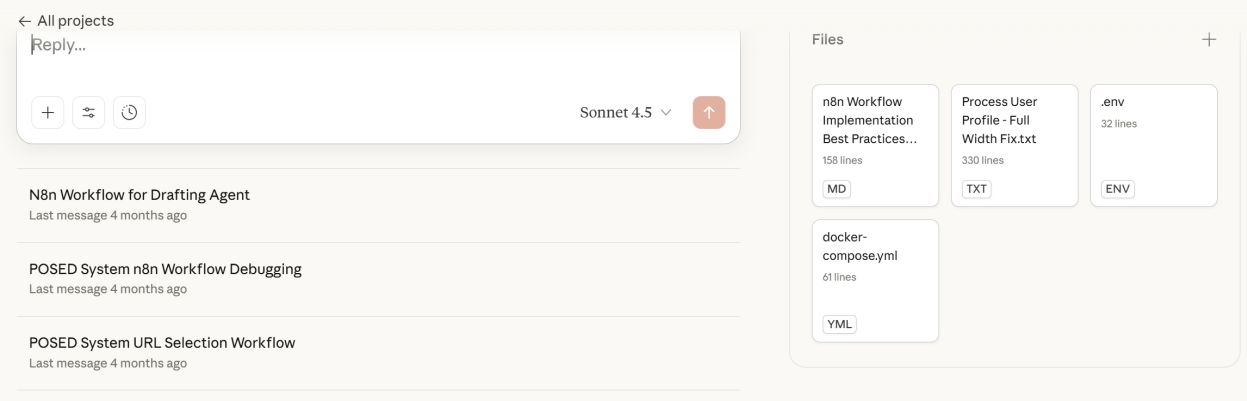

Figure 4: Persistent Project Context Setup. A screenshot of Claude interface. Critical documentation, e.g. workflow best practices file is pinned to the project context (right pane), supporting architectural consistency across separate coding sessions/chats.

2.2.2 Phase 2: Design and Pre-Implementation Validation

Step 4: Visual Skeleton Construction. A critical success factor was that the domain expert drew the workflow topology in n8n manually before requesting any generated code (see Figure 5 for a representative topology). Empty nodes were placed as placeholders with brief comments describing their intended behavior (pseudocode injection). We found this manual skeleton step was necessary because requiring an LLM to generate the n8n subflow JSON directly often failed in practice: the model frequently produced invalid node configurations or malformed JSON that could not be imported as legitimate n8n nodes. Starting from a human-constructed skeleton, subsequent AI assistance could be constrained to filling in narrow implementation details (e.g., Code Node JavaScript) within a valid workflow structure.

Step 5: Design Review and Two-Stage Pre-Code Audit. Before writing any JavaScript, we first clarified the subflow's function, inputs, outputs, and control logic with the Strategic Partner (Gemini), often using a flowchart or brief pseudocode description. After the skeleton was drawn, we exported the subflow JSON and captured a screenshot of the complete topology. These artifacts were then used for a strict pre-implementation review to reduce rework and limit expensive inference usage:

- **Stage A (Logic and Dataflow Audit, Gemini):** The exported skeleton JSON and screenshot were provided to Gemini along with the written logic description. We asked Gemini to trace the data flow, verify inputs/outputs, and identify missing steps, logic errors, or data-shape mismatches.

- **Stage B (Implementation Readiness Check, Claude):** Only after Gemini verified the logic did we pass the same artifact bundle, along with the verified logic and I/O description, to Claude. We used a brief "clear the air" discussion to confirm assumptions before requesting Code Node implementations and any required configuration details.

Implementation Spotlight: The Persona Extraction Sub-Workflow

To illustrate the application of this protocol, we examine the "Persona Extraction" agent (see Figure 5). This sub-workflow addresses a common failure mode in AI-generated content: the lack of authentic voice.

Instead of relying on generic style prompts, this agent is engineered to clone the domain expert's specific pedagogical tone through a five-stage process:

1. **Ingestion:** The user uploads raw PDF samples of previous lectures via a Form Node.

2. **Decomposition:** A loop node iterates through the documents, extracting raw text while preserving structural hierarchy.

3. **Synthesis (AI Agent):** The extracted text is passed to an AI agent with a strict instruction to analyze rhetorical patterns, vocabulary complexity, and humor.

4. **Human Verification (HITL):** Crucially, the AI does not save this profile automatically. A Form Node pauses the execution, presenting the generated profile to the professor for editing. After approval, the persona file is saved to the local folder.

5. **Manifest Update:** Finally, the workflow writes the persona's path and filename to the central `session_manifest.json`, making it available to downstream drafting agents.

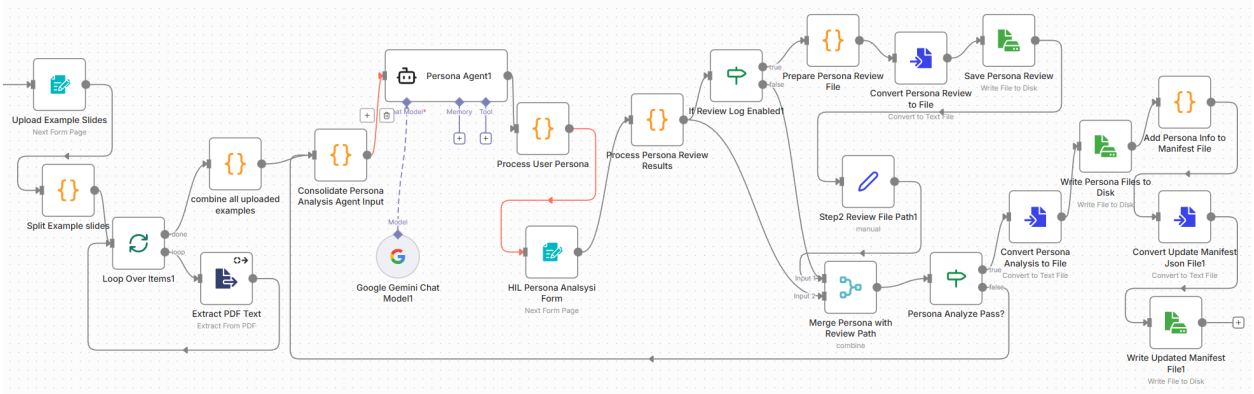

Figure 5: **The Persona Extraction Agent.** A visual representation of the n8n sub-workflow that analyzes instructor materials.

2.2.3 Phase 3: Implementation and Debugging

Step 6: Standardized Prompt Template Initialization. With the skeleton validated (Step 5) and the context established (Step 3), we used a consistent "Initial Prompt Template" to trigger the coding phase for each sub-workflow. A condensed example used for the "Persona Extraction" subflow is shown below:

> **Role:** You are an expert n8n JavaScript developer.

Goal: Implement the requested node or subflow reliably and debuggably.

Inputs: (i) the approved implementation plan, (ii) file-path references to intermediate Markdown artifacts, and (iii) the current manifest.json/session metadata.

Core workflow directions: Ask clarifying questions if needed; prefer deterministic parsing/validation in code nodes; generate text in Markdown; treat JSON as a code-assembled structure ("AI for text, code for JSON"); and update the manifest after each successful stage.

Output format: Return only the requested n8n node code (and any required schema/config), plus a short checklist of assumptions.

(Template excerpted for brevity; the full prompt template and node patterns are available as a development artifact.)

Step 7: Debugging Triage and Escalation Levels. Errors were categorized into three levels and addressed through the n8n interface (see Figure 6):

- **Level 1 (Syntax):** Syntax errors were handled by returning the execution trace to Claude and requesting a corrected implementation.
- **Level 2 (Logic):** Logic errors were identified by inspecting node inputs and outputs, and were typically resolved by refining the task specification or clarifying the requirement prompt.
- **Level 3 (Design):** Design errors reflected more fundamental flaws in the workflow topology and therefore required structural changes. For example, we found that complex branching logic implemented inside AI Agent nodes through prompt instructions was unreliable. The solution was to extract that logic into a preceding deterministic Code Node, which pre-calculated the relevant context and constructed a corresponding prompt for the AI agent.

Step 8: Capability-Based Model Tiering. We prioritized the most reliable model for the components of the workflow where mistakes are expensive, especially n8n Code Nodes that implement non-trivial JavaScript (e.g., parsing, loop control, and dynamic prompt construction). In a low-code environment, much of the system logic is not expressed as plain source code; it is embedded in the visual workflow topology and node configuration. As a result, effective debugging often required the model to reason from multiple context sources at once, including a screenshot of the logic flow, exported n8n subflow JSON, and the surrounding project files (e.g., Markdown artifacts and session/manifest metadata) that define inputs and outputs. One lesson from development was that strong benchmark or "coding test" performance did not always translate to dependable real-world behavior under these platform-specific constraints and partial visibility. We also found that a "cheap draft + expensive fix" pattern was often counterproductive: having a weaker model generate the first version and then delegating a stronger model to debug it tended to create longer repair cycles and more regressions than starting with the stronger model for the critical code path. For these high-leverage

nodes, using the most capable model up front reduced rework and produced faster overall convergence, even when its per-call cost was higher.

Step 9: Cross-Model Arbitration ("Council of Rivals"). When implementation stalled (e.g., repeated regressions or conflicting assumptions), we treated disagreement between the Strategic Partner (Gemini) and the Implementation Specialist (Claude) as a signal to escalate rather than iterate blindly. We then requested independent second opinions from ChatGPT and Grok and compared the proposed fixes to surface hidden assumptions and identify the next concrete test. The Project Director synthesized these perspectives and selected the final engineering decision.

Step 10: Clean-Restart with an Updated Specification. We observed that extended troubleshooting threads can accumulate conflicting assumptions and "polluted" context, reducing the quality of subsequent fixes. When debugging exceeded a certain number of iterations without converging (typically after five attempts), we deliberately abandoned the thread and restarted in a fresh session. Rather than retrying with the same prompt, we began the new session with an updated prompt template that incorporated lessons from the failure. For example, we added explicit directives such as "avoid relative references; use absolute data retrieval," or "add conditional logic inside the loop to distinguish the initial run from feedback iterations." This refined restart helped the next session begin from a clearer, higher-quality specification and reduced the likelihood of repeating the same error patterns.

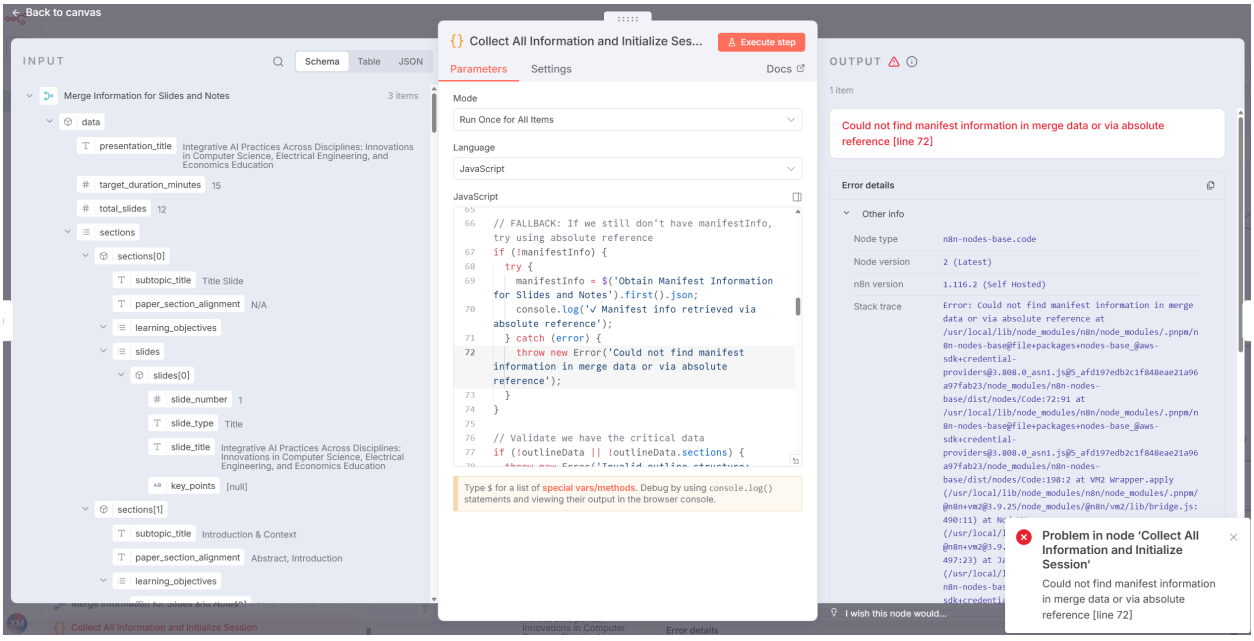

Figure 6: **The Targeted Debugging Interface.** A screenshot of the n8n Code Node during a debugging session. The interface provides a clear separation of concerns: the input JSON schema is validated on the left, the JavaScript logic is edited in the center, and specific execution errors are isolated on the right. This allows for precise, targeted troubleshooting rather than opaque trial-and-error.

2.2.4 Phase 4: Use, Hardening, and Continuous Improvement

The engineering lifecycle concluded not with a static release, but with a formal process for continuous iteration, leveraging principles from Agile development. This final phase focuses on improving usability, maintainability, and accumulated team knowledge (Steps 11–12).

Step 11: Builder-as-user Iteration. Because the development was faculty-led, the lead developer also used the system in authentic day-to-day tasks. This "builder-as-user" setup created rapid feedback during development: we could immediately identify steps that were confusing or unnecessary, add logging when a process was hard to diagnose, and revise data flows that were technically correct but awkward to operate. As a result, iteration was guided not only by whether the workflow ran, but by whether it was practical to use and maintain.

Step 12: Retrospective and Playbook Update. After completing a subfunction, we conducted a short retrospective with the AI assistants to capture what worked and what failed. We asked for concise lessons learned and for revised prompt templates or reusable patterns that could reduce repeated mistakes in the next iteration. These notes were then saved back into the persistent Project Context (e.g., the Best Practices file described in Step 3), creating a growing internal playbook that supported faster and more consistent development over time.

This twelve-step protocol outlines a disciplined approach to human–AI co-development. By assigning clear roles (e.g., Architect, Implementer, and Project Director) and using explicit workflow constraints and checkpoints, we made the development process more transparent and easier to debug. The result is a practical pipeline that supports cost-aware iteration and traceable decisions while keeping the educator in charge of key design choices. The workflow leverages LLMs for targeted reasoning and coding tasks without surrendering oversight of the system's structure, behavior, or instructional intent.

From the perspective of software engineering, the protocol functions like a short-cycled Agile sprint. The educator plays product owner and end-user, setting requirements and immediately validating usability, while AI assistants serve the roles of specialized contributors for architecture, implementation, and troubleshooting. This role shift enables rapid iteration with explicit acceptance gates and continuous refinement guided by real use.

3. Results and Discussion

This section reports results from two faculty-built tools implemented with the same low-code + LLM development paradigm: (i) the POSED curriculum workflow and (ii) a paper-to-presentation generator with a smaller, input-constrained scope. Across both builds, we observed (a) faster time-to-MVP than a typical small-team software effort, (b) deployable end-to-end artifacts suitable for real teaching and presenting contexts, and (c) practical engineering patterns that helped stabilize development (e.g., artifact-first file contracts, HITL checkpoints, and recovery procedures). We conclude by discussing why the approach worked and how the developer role shifted toward verification-oriented practice.

3.1 Build Outcomes and Replication Across Two Tools

The primary hypothesis of this study is that an AI-augmented workflow, when paired with deterministic orchestration, explicit artifact contracts, and mandatory HITL checkpoints, can reduce the cycle time required for a faculty member to implement customized educational tools.

Table 1 compares the observed timeline for the initial implementation with an estimated timeline for a traditional small two-person development team. Although the traditional estimate is necessarily approximate, it is broadly consistent with published ranges of 4–9 months for custom educational software developed by small teams[17]. We use the higher end of that range to reflect the additional specification and iteration often required for novel pedagogical tools. This comparison provides a practical baseline for interpreting the observed reduction in time-to-MVP.

Table 1: Comparison of estimated traditional development timeline versus the initial AI-augmented timeline.

Metric	**Traditional Dev (Est.)**	**Professor + AI**	**Improvement**
Team Structure	1 Full-Stack Engineer + 1 Designer	1 Faculty + AI	N/A
Requirement Gathering	3–5 Weeks	1 Week	$\sim$3–5$\times$ faster
Implementation	6–10 Months	4–5 Weeks	$\sim$6–10$\times$ faster
Total Time	8–13 Months	5–6 Weeks	$\sim$6–11$\times$ faster

A second implementation using the same low-code framework, a paper-to-presentation generator with a smaller, input-constrained scope, reached a working MVP in approximately 2.5 weeks. This reduction was enabled by reusable workflow artifacts and controls established during the first build (e.g., prompt templates, loop-handling patterns, and operational procedures), as well as infrastructure elements that transferred directly between projects (e.g., `docker-compose` setup, `.env` conventions, and the n8n hosting configuration). In contrast, exported subflow JSON was not plug-and-play: differences in project variables, folder structures, and file formats required substantial adaptation before a prior subflow could run in the new workflow. Even so, these previously working subflows remained valuable as reference implementations; they could be provided to the Implementation Specialist as concrete examples of previously validated node patterns and execution behavior in n8n, which reduced ambiguity and shortened debugging cycles in the second build.

Table 2: Replication across two projects using the same low-code + AI development paradigm.

Tool	**Scope Driver**	**Time-to-MVP**	**Workflow Size**
POSED curriculum workflow	Open-ended course material generation	6 weeks	214 nodes
Paper-to-presentation workflow	Constrained by input paper	2.5 weeks	180 nodes

Workflow exports provide a lightweight structural proxy for architectural choices made between builds. The POSED workflow contained 214 nodes, including 32 file-oriented nodes (e.g., read / write / convert / extract; $\sim$15%). The paper-to-presentation workflow contained 180 nodes with 59

file-oriented nodes ($\sim$33%), reflecting a deliberate architectural shift: the workflow passes references (paths/IDs) and subflows read/write Markdown artifacts, rather than carrying large content payloads through inter-node state.

3.2 Operational Resilience in Self-Hosted Development

During development, an unexpected shutdown corrupted the Docker-hosted n8n state and triggered repeated startup failures. To reduce "single-laptop fragility" and make recovery repeatable, we adopted lightweight workflow version control by routinely exporting and archiving the n8n workflow JSON, and we codified operational scripts and procedures.

These operational controls included start/stop scripts for clean service lifecycle management, a "safe backup" script to snapshot the working state, and an update script to reduce friction when upgrading the n8n stack. For worst-case crash loops caused by corrupted execution data (e.g., infinite-loop artifacts), we created a hard reset and recovery guide that documents a deterministic restore path while preserving external vector storage (Qdrant). Collectively, these practices treated maintainability and recovery as first-order requirements, enabling faster iteration without relying on a fragile, one-off development environment.

3.3 Output Artifacts and Usability Evidence

This work emphasizes implementability and deployability over benchmark-style comparisons. Figure 7 shows a representative slide produced by the curriculum workflow after passing HITL gates, illustrating that the pipeline can produce a classroom-ready artifact rather than intermediate drafts.

Beyond static slides, the paper-to-presentation workflow produces an interactive HTML deck intended for real teaching and presenting contexts. Figure 8 shows an example output generated from a conference paper, including a built-in pacing timer added to address a practical instructor constraint: maintaining time awareness while presenting.

To complement the author's own observations during development, the second author, an instructional designer, reviewed the speaker notes and slide decks produced by both workflows. The review asked a practical question: would these artifacts be usable in a real class session? Specifically, the reviewer examined (i) whether the materials managed cognitive load through clear segmentation and pacing, (ii) whether examples and framing were anchored to the target ECE audience, (iii) whether instructor-facing supports (e.g., speaker notes or a teacher layer) were present, and (iv) whether the materials created openings for active learning.

Across the sampled artifacts, the reviewer highlighted that enforced structure, for example, one primary idea per slide, explicit transitions, and intermediate approval checkpoints, helps reduce overloaded slides and supports pacing. The review also noted the value of domain-specific examples and analogies aligned with students' prior knowledge, and emphasized facilitator-facing supports (e.g., speaker notes and time-management cues) that align with the pacing timer added to the HTML workflow. Overall, this expert review suggests that workflow engineering decisions (artifact-first file contracts, structured templates, and mandatory human checkpoints) can translate

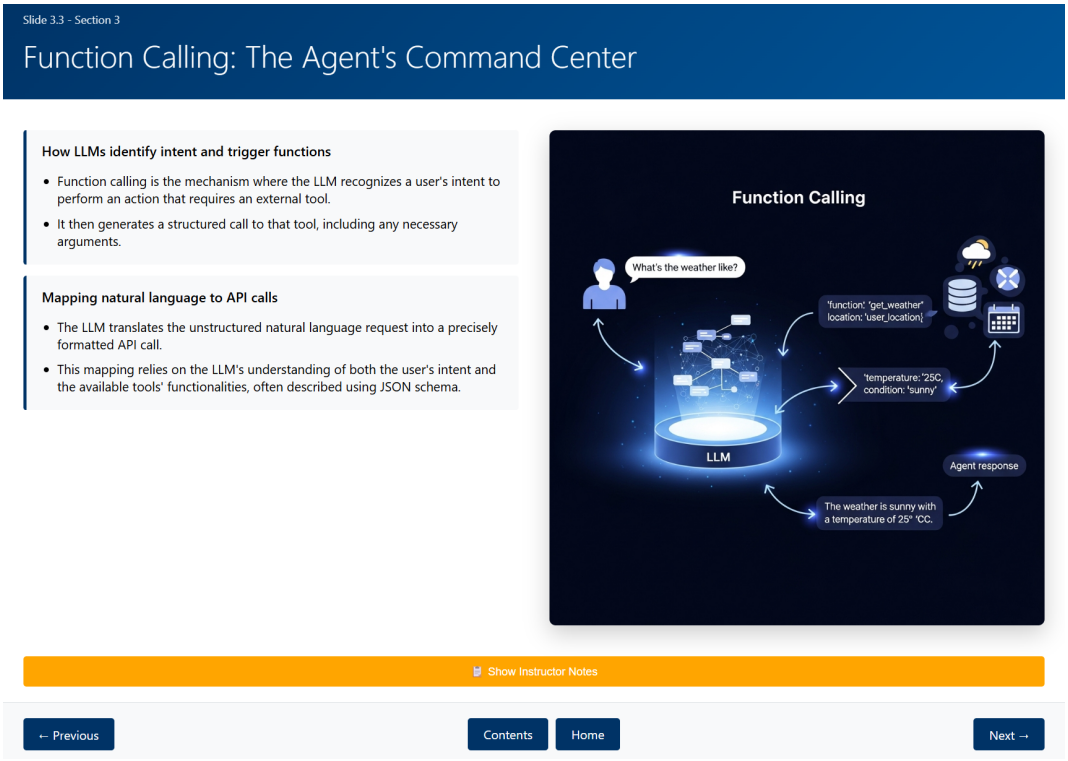

Figure 7: Representative slide produced by the curriculum workflow after passing HITL gates.

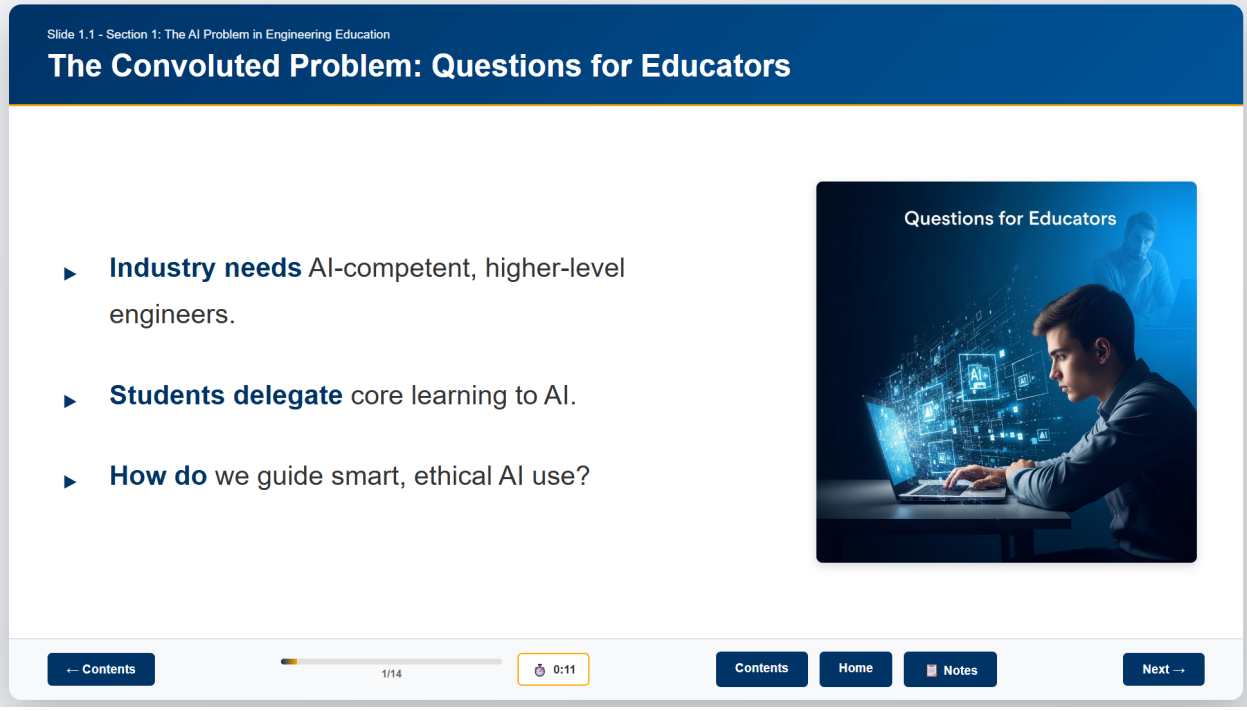

Figure 8: Paper-to-presentation tool output in HTML. The generated deck includes navigation controls and a pacing timer added to support time management during live presentations.

into pedagogically usable artifacts in realistic teaching contexts.

3.4 Discussion: Why It Worked and What the Role Shift Enabled

The effectiveness of the paradigm is best explained by the interaction of deterministic workflow controls with probabilistic AI-agent outputs, coupled with a development approach that prioritized verification, stability, transparency, and maintainability over one-shot generation.

Deterministic scaffolding around probabilistic AI-agent outputs. n8n provided explicit branching logic, reproducible execution traces, and targeted Code Nodes for deterministic parsing and validation. Beyond making the workflow easier to debug, this scaffolding functioned as a quality-control layer: it enforced required formats, caught malformed outputs early, and prevented downstream steps from running on invalid intermediate artifacts. These controls constrained LLM calls within a predictable execution frame and enabled refined restarts of only the failed subroutines rather than re-running the entire workflow.

Artifact-first file contracts (with a manifest as the index). In both projects, intermediate and final outputs were generated as human-readable files (e.g., persona, outline, drafts), enabling inspection and editability outside the automation tool. A key refinement in the second project was introducing a central `manifest.json` as an index over these artifacts. Each subflow begins by reading the manifest to locate the resources it needs and ends by updating it with newly created artifacts; therefore, a subflow can operate with minimal inter-workflow parameter passing. Beyond portability, the manifest also improves user control: because artifacts are stored as editable files, instructors can locate outputs quickly via the manifest and make offline modifications (e.g., in a text editor) before resuming the workflow, increasing flexibility without breaking the orchestration logic.

Failure modes and mitigations. Several recurring failure modes were observed (e.g., malformed structured outputs, divergent revision loops, and token growth during iterative refinement). Mitigations included the "AI for text, code for JSON" pattern, convergent loop data handling with lightweight memory strategies, and operational recovery procedures for self-hosted deployments. Mandatory HITL gates ensured that human judgment was not advisory but compulsory, blocking downstream execution until intermediate outputs were approved.

Emergent reciprocal competence and faculty-centered customization. An important qualitative outcome was the emergence of *reciprocal competence*: as the faculty developer iteratively debugged and refined the workflow, they internalized both software-engineering practices (e.g., modular decomposition, explicit state contracts, recovery procedures) and more effective ways to elicit and validate AI assistance. This competence became encoded into reusable artifacts, prompt templates, loop-handling patterns, orchestrator planning notes, and restart guides that were directly reused in the second implementation and contributed to the observed reduction in development time. The pacing-timer feature in the HTML generator is a concrete example of faculty-centered customization: such micro-features are often too specialized for a traditional development queue, yet they materially improve teaching practice and become feasible when faculty can iterate rapidly within a governed low-code environment.

Implications: a shift toward verification-oriented development and capability requirements. Across both implementations, we observed a consistent role shift: the developer spent less time writing code directly and more time specifying constraints, defining inputs/outputs, running targeted tests, and integrating changes into a working system. In this sense, the workflow resembles a compressed Agile cycle, where AI assistants accelerate implementation while the human sets acceptance criteria and remains accountable for correctness, usability, and maintainability.

Viewed through an industry lens, this role shift resembles the transition from an individual contributor to a technical lead coordinating a small team: instead of implementing every component directly, the developer invests more effort in clarifying intent, delegating to specialized contributors, integrating outputs, and reviewing quality through concrete acceptance checks and working demos. In our case, the "team" consisted of AI assistants with different strengths, so effective coordination relied on technical leadership skills such as capability-aware model selection and tiering, configuring models for specific sub-tasks, providing sufficient and well-scoped context (e.g., workflow screenshots, exported JSON, and artifact files), and constructing prompts that define clear inputs, outputs, and acceptance criteria. These practices complement specification, verification, and integration by making collaboration with multiple AI agents more predictable and easier to debug.

Harness engineering framing. More recently, the emerging harness engineering lens provides useful vocabulary for interpreting several architectural aspects of this work[18,19,20]. In current industry usage, a harness often refers to a runtime layer that manages state persistence, replay, observability, error recovery, and other control functions around long-running agent loops with tool invocation. The present framework implements analogous functions through explicit human-designed workflow structure rather than autonomous runtime control: structured task sequencing is realized through n8n subflows; artifact-based state persistence through Markdown files and `manifest.json`; working context through persisted instructional artifacts; dynamic prompt assembly through code nodes; tool integration through native n8n nodes; validation hooks through deterministic code-node checks; and human approval gates through HITL Form nodes. At the same time, this study is more accurately framed as a governed workflow framework than as a full harness. In educational settings, privacy, intellectual property, and academic integrity must take priority over runtime automation, which is why human-designed controls remain central. In this respect, the work can be viewed as an early framework-level precursor that shares design intent with harness engineering while remaining bounded by institutional governance constraints.

Positioning within prior work. This interpretation also helps situate the present study within our prior line of work. Earlier efforts established agentic AI patterns for educational design through prompt-engineered multi-agent educational simulation[13], followed by broader interdisciplinary AI-supported teaching practices and early POSED-related curriculum-development workflows[21]. The present paper extends that trajectory from AI-assisted instructional generation to the engineering of the governing workflow framework itself, making the system more inspectable, restartable, and adaptable across educational software tasks.

More broadly, the experience suggests a capability shift that mirrors earlier transitions in computing practice. As higher-level languages and compilers matured, the premium skill moved from

low-level resource micromanagement toward algorithmic thinking, system structure, and interface design. In an AI-augmented development setting, a similar rebalancing may occur: value concentrates less in writing routine syntax and more in (i) articulating intent as testable requirements, (ii) designing clear data and interface contracts, (iii) instrumenting workflows for transparency and debugging, and (iv) validating outputs for correctness, safety, and maintainability. For engineering educators, this suggests a need to examine how curricula can better emphasize verification- and integration-centered skills alongside foundational programming.

4. Conclusion and Future Work

This work contributes a repeatable, faculty-led development protocol and a set of transferable workflow patterns for building customized educational computing tools on a code-centric low-code platform. Across two implementations, deterministic orchestration, artifact-first file contracts, and mandatory HITL gates enabled rapid iteration while keeping human judgment and institutional data governance central. The reusable professor-persona file, in particular, functions as a form of long-term instructional memory that can be retained, revised, and extended across runs. Framed in harness engineering terms, the present system constitutes an early framework-level implementation: it already realizes key functions such as structured task sequencing, artifact-based state persistence, dynamic prompt assembly, tool integration, validation hooks, and human approval gates through explicit workflow design rather than autonomous runtime control. An additional practical advantage of the framework is that it supports heterogeneous LLM routing within a single governed workflow. This allows capability-intensive stages to be assigned to stronger cloud-based models when needed, while privacy-sensitive, IP-sensitive, or cost-sensitive stages can be routed to local LLMs, making model choice a stage-level governance decision rather than a single global commitment.

Future work will evaluate the protocol as a faculty-centered development method across additional tools, instructors, and course contexts. We will (i) characterize which workflow patterns transfer cleanly versus which require adaptation, (ii) quantify development effort using practical measures such as time-to-MVP (minimum viable product), iteration counts, and recovery events, and (iii) document adoption friction points (e.g., platform constraints, debugging bottlenecks, and self-hosting maintenance) to produce a more robust implementation playbook. We will also study how artifact-first contracts and HITL gates shape quality and maintainability over time by incorporating user feedback loops (e.g., instructor revision patterns and student-facing usability signals) into subsequent iterations of the toolchain. In parallel, we will explore selective adoption of harness-oriented mechanisms within the n8n framework. Two concrete directions are (i) node-level replay hooks, where a failing node emits a structured failure signal that triggers a bounded fallback routine and localized retry rather than restarting the entire workflow, and (ii) longitudinal persona memory, where instructor feedback collected at HITL gates is consolidated by the LLM into the persistent professor-persona artifact, enabling prompts to adapt to individual preferences across runs. Both extensions remain compatible with academic integrity and workflow-governance requirements because persistence remains within the institution-controlled artifact layer.

Finally, this work suggests broader instructional implications for computing education. If AI-assisted development shifts effort toward specification, verification, integration, and reliability,

then curricula may need to more explicitly teach and assess these skills alongside foundational programming. We will pilot course activities that emphasize acceptance criteria, interface and data contracts, workflow instrumentation, regression detection, and recovery documentation, and we will examine how these competencies align with emerging industry practice in AI-augmented and "vibe coding" workflows.

References

[1] V. Bamil, "Vibe coding: Toward an AI-native paradigm for semantic and intent-driven software engineering," *arXiv preprint arXiv:2510.17842*, 2025.

[2] J. Li, Y. Hou, L. Lin, R. Zhu, H. Cao, and A. E. Ali, "Vibe coding for UX design: Understanding UX professionals' perceptions of AI-assisted design and development," *arXiv preprint arXiv:2509.10652*, 2025.

[3] I. Mehta, "A quarter of startups in YC's current cohort have codebases that are almost entirely AI-generated," TechCrunch, Mar. 2025, [Online]. Available: https://techcrunch.com/2025/03/06/a-quarter-of-startups-in-ycs-current-cohort-have-codebases-that-are-almost-entirely-ai-generated/.

[4] J. Morais, "The engineer in the AI age: The orchestrator and architect," The New Stack, Sep. 2025, [Online]. Available: https://thenewstack.io/the-engineer-in-the-ai-age-the-orchestrator-and-architect/.

[5] A. Sarkar and I. Drosos, "Vibe coding: Programming through conversation with artificial intelligence," in *Proc. 36th Annual Conf. Psychology of Programming Interest Group (PPIG)*, Belgrade, Serbia, Sep. 2025, arXiv:2506.23253. [Online]. Available: https://arxiv.org/abs/2506.23253.

[6] M. A. A. Alamin, G. Uddin, S. Malakar, S. Afroz, T. Haider, and A. Iqbal, "Developer discussion topics on the adoption and barriers of low code software development platforms," *Empirical Software Engineering*, vol. 28, no. 1, p. 4, 2023.

[7] N. Rao, J. Tsay, K. Kate, V. J. Hellendoorn, and M. Hirzel, "AI for low-code for AI," in *Proc. 29th Int. Conf. Intelligent User Interfaces (IUI)*, Greenville, SC, USA, Mar. 2024.

[8] Bluebash, "How to choose between langflow and n8n for AI workflow automation?" 2025, [Online]. Available: https://www.bluebash.co/blog/langflow-vs-n8n-ai-workflow-automation/.

[9] D. Serekov, A. Bissembayev, T. Iliev, A. Mukasheva, and J. W. Kang, "Evaluating low-code development platforms: A MULTIMOORA approach," *Engineering Proceedings*, vol. 104, no. 1, p. 15, Aug. 2025.

[10] IBM, "What is agentic AI?" 2025, [Online]. Available: https://www.ibm.com/think/topics/agentic-ai.

[11] A. Ng, "Agentic design patterns part 5: Multi-agent collaboration," The Batch, DeepLearning.AI, Apr. 2024, [Online]. Available: https://www.deeplearning.ai/the-batch/agentic-design-patterns-part-5-multi-agent-collaboration/.

[12] DeepLearning.AI, "AI agentic design patterns with autogen," 2024, [Online]. Available: https://www.deeplearning.ai/short-courses/ai-agentic-design-patterns-with-autogen/.

[13] X. Ma and J. Wang, "WIP: Active learning through prompt engineering and agentic AI simulation: A pilot project in computer networks education," in *Proc. IEEE Frontiers in Education Conf. (FIE)*, Washington, DC, USA, Oct. 2024.

[14] J. Moura, "Crewai: Orchestrating role-based AI agents," GitHub repository, 2024, [Online]. Available: https://github.com/joaomdmoura/crewAI.

[15] L. Zhang *et al.*, "Multi-agent systems for IEP development: Bridging the gap in special education," CIDDL Research Brief, Nov. 2024, [Online]. Available: https://ciddl.org/multi-agent-systems-for-iep-development/.

[16] G. Kostopoulos, V. Gkamas, M. Rigou, and S. Kotsiantis, "Agentic AI in education: State of the art and future directions," *IEEE Access*, vol. 13, pp. 177 467–177 491, Oct. 2025.

[17] B. Ramsey, "Educational software development: Creating personalized learning experiences," Scopic Software, 2023, [Online]. Available: https://scopicsoftware.com/blog/education-software-development/.

[18] Anthropic, "Effective harnesses for long-running agents," Anthropic Engineering, Nov. 2025, [Online]. Available: https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents.

[19] R. Lopopolo, "Harness engineering: Leveraging codex in an agent-first world," OpenAI Engineering, Feb. 2026, [Online]. Available: https://openai.com/index/harness-engineering/.

[20] B. Böckeler, "Harness engineering for coding agent users," Martin Fowler, Apr. 2026, [Online]. Available: https://martinfowler.com/articles/harness-engineering.html.

[21] X. Ma, Y. Wu, X. Liu, and J. Wang, "Integrative AI practices across disciplines: Innovations in computer science, electrical engineering, and economics education," in *Proc. IEEE Frontiers in Education Conf. (FIE)*, Nashville, TN, USA, Nov. 2025.