From Vibes to Veracity: An Adversarial AI Agentic Role-Play Framework for Software Engineering and Security*

Yongge Wang
UNC Charlotte, USA
yonwang@charlotte.edu

May 26, 2026

Abstract

The rapid proliferation of Large Language Models (LLMs) has introduced "vibe coding"—a paradigm where software development is guided by intuitive, conversational prompts. While this approach enables rapid prototyping, it often lacks the engineering rigor required for production-grade security and performance, leading to the "Vague Prompt Trap." This paper introduces a formalized **Chain-of-Thought (CoT) framework** designed to transform intuitive "vibe-based" development into a systematic engineering discipline. The primary focus of this work is the development of an AI agent based AETHERPROOF IDE[1] and the end-to-end development and security auditing of **Queen City AI (QCAI)**[2] using AETHERPROOF IDE, a heterogeneous digital ecosystem comprising native mobile applications and scalable backends. We demonstrate how tiered CoT planning, utilizing multi-agent debate modeled after the formal Fagan inspection process, enables the maintenance of architectural coherence and semantic parity across diverse technology stacks. To validate the robustness of this methodology, we provide further empirical evaluations in two specialized domains: high-performance cryptographic optimization (AES and RLCE) and adversarial security testing using the OWASP Juice Shop benchmark. Our results show that CoT-driven AI orchestration allows the human "Director" to achieve expert-level results, effectively bridging the gap between intuitive AI assistance and formal software assurance.

1 Introduction

The landscape of software engineering is undergoing a fundamental shift from manual code construction to high-level AI orchestration. This evolution represents the latest paradigm in the long history of **Automated Software Engineering**, which traces its roots back to the 1950s with the first software testing teams and progressed through the record-and-playback tools of the 1980s and the framework-driven automation (e.g., JUnit, Selenium) of the 2000s [1]. Today, the field has transitioned from deterministic, script-based automation to intelligent, generative workflows powered by Large Language Models (LLMs). This shift has given rise to the term "vibe coding," popularized by AI researcher Andrej Karpathy in early 2025 [6]. Vibe coding describes a workflow where the primary role of the developer shifts from writing code line-by-line to guiding an AI assistant through a conversational process, prioritizing functional intent or the "vibe" over low-level implementation details.

Recent industry surveys indicate that by 2025, over 84% of software developers have integrated AI-driven tools into their Development Life Cycle (SDLC) [1]. However, as developers "give in to the vibes," they often encounter the **Vague Prompt Trap**: the phenomenon where underspecified or high-level instructions lead the AI to generate hallucinated, unoptimized, or insecure code. While tools like GitHub Copilot enhance local productivity, they often optimize for short-term completion rather than global architectural coherence. This creates a "silent debt" where AI-generated code lacks clear rationale or long-term maintainability.

*Clinical trial number: not applicable.

[1] https://github.com/amalithlab/aetherproof

[2] https://queencityai.net/ and https://apps.apple.com/app/id6757207914

To address this, the research community has proposed several prompting paradigms to enforce reasoning. Notably, **Structured Chain-of-Thought (SCoT)** prompting guides LLM reasoning using programming-specific constructs (sequential, branch, loop), significantly improving code accuracy [10]. Further refinements like **Semantic Chain-of-Thought (SeCoT)** incorporate code-semantic features such as data flow into the reasoning process [9].

We argue that the most successful way to bridge the gap between intuitive "vibe-based" development and formal software engineering is through a formalized **CoT-Vibe framework**. Specifically, during the AI brainstorming process, we set up multiple agents, each powered by a different underlying LLM. These agents take on distinct roles to argue, analyze the design, and review the generated code. To structure this multi-agent interaction, we adopt a methodology inspired by the Fagan inspection process [3, 4]. Originally introduced by Michael E. Fagan as a formal software review technique, the Fagan inspection defines specific roles—such as Moderator, Author, Reader, and Tester—who systematically evaluate the software artifact to identify defects before proceeding to the next development phase. By applying this rigorous, role-based review style to our AI agents, they iteratively debate and refine the architecture until an agreement is reached, enforcing architectural coherence, applying systematic security guidelines (such as OWASP), and achieving performance levels that match or exceed those of human experts.

The current state-of-the-art (SOTA) in AI-driven coding is defined by high-reasoning models such as Anthropic's Claude Sonnet 4.6 (Thinking), which leads in autonomous agentic workflows through its native Agent SDK, and Google's Gemini 3.1 Pro, which demonstrates superior performance in long-context codebase reasoning and real-world issue resolution (scoring at the top of SWE-bench ++ with its specialized "Deep Thinking" loops) [5, 1, 6]. These 2026-era models, including Open-Source's GPT-OSS 120B, have moved beyond simple auto-completion to include internal verification steps that allow them to handle complex, multi-step logic puzzles that were previously unsolvable. Simultaneously, the deployment of these models has shifted from simple chat interfaces to agent-first Integrated Development Environments (IDEs). Tools such as **Cursor** have introduced persistent codebase awareness that significantly increases development velocity [2], while open-source agents like **Roo Code** (formerly Roo Cline) enable autonomous multi-file refactoring and "auto-execution" of builds. Most recently, platforms like Google's **Antigravity** have emerged as specialized ecosystems where multiple autonomous agents, powered by Gemini 3.1 Pro and Claude Opus 4.6 (Thinking), orchestrate complex tasks across files and terminals, communicating progress via structured human-readable artifacts.

This paper presents the CoT-Vibe framework as a solution to this shift in the software engineering landscape. A key distinction of our methodology is that it is not merely theoretical; rather, it is directly derived from the practical development experience of a large-scale, real-world deployment. Our primary contribution is the development of an AI agent based AETHERPROOF IDE[3] which integrates the framework of CoT planning methodology and a detailed case study of **Queen City AI (QCAI)**[4], a multi-platform digital ecosystem comprising native mobile applications and scalable backends, which was developed autonomously using our AETHERPROOF IDE. To further illustrate the generalizability of this framework, we also provide validation experiments in cryptographic optimization (AES and RLCE) and adversarial security testing (benchmarked against OWASP Juice Shop).

2 Related Work

2.1 Vibe Coding and Agentic Engineering

As defined by Karpathy, vibe coding represents a "code first, refine later" mindset that aligns with agile principles but relies on LLMs as the engine of production. More recently, this concept has evolved into "agentic engineering," where developers orchestrate multiple AI agents with greater oversight. Our work formalizes this transition by introducing a structured planning layer between the human "director" and the AI lead developer.

[3] https://github.com/amalithlab/aetherproof

[4] https://queencityai.net/

2.2 Chain-of-Thought in Software Engineering

Chain-of-Thought (CoT) prompting enables LLMs to decompose complex problems into intermediate reasoning steps. While the general idea of "thinking out loud" has long existed in human cognition, the formalization of this technique for Large Language Models was introduced in a seminal 2022 paper by a team at Google Research led by Jason Wei and Denny Zhou [18]. They demonstrated that providing few-shot examples comprising both a question and a step-by-step reasoning chain drastically improved the performance of LLMs on complex arithmetic and symbolic reasoning tasks.

For instance, Wei and Zhou compared models using standard few-shot prompting versus CoT prompting, as illustrated in Figure 1. In standard prompting, providing an example with only the final answer (the "Roger" problem) caused the model to hallucinate a response ("27") when evaluated on a subsequent unseen problem (the "Apples" problem) because it attempted to skip directly to the solution. In contrast, incorporating intermediate reasoning into the prompt taught the model to replicate this step-by-step "scratchpad" logic, successfully arriving at the correct answer for the new problem. Kojima et al. subsequently proved that simply appending "Let's think step by step" to a prompt without prior examples turns LLMs into effective zero-shot reasoners [7].

Standard Prompting (No CoT)
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many balls does he have now?
A: The answer is 11.
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
A: **The answer is 27.** (Incorrect)

Chain-of-Thought Prompting (With CoT)
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many balls does he have now?
A: Roger started with 5 balls. 2 cans of 3 balls each is 6 balls. 5 + 6 = 11. The answer is 11.
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
A: **The cafeteria had 23 apples originally. They used 20 to make lunch. So they had** $23 - 20 = 3$**. They bought 6 more apples, so they have** $3 + 6 = 9$**. The answer is 9.** (Correct)

Figure 1: Comparison of Standard Few-Shot Prompting vs. Chain-of-Thought Prompting, as demonstrated by Wei et al. [18].

In the context of software engineering, Li et al. introduced Structured Chain-of-Thought (SCoT), showing that guiding models through programming-specific constructs (e.g., loops, branches) significantly improves the quality of generated code over standard few-shot prompting [10]. Further work on Semantic Chain-of-Thought (SeCoT) has demonstrated that incorporating data-flow and control-flow abstractions into the reasoning chain allows models to bridge the semantic gap between requirements and implementation [9]. Our framework extends these concepts by applying CoT not just to localized code blocks, but to the entire architectural planning and security auditing process of a multi-platform ecosystem.

2.3 Automated Security and Code Review

Traditional formal code reviews, such as Michael Fagan's 1976 inspection method, are highly effective but labor-intensive, often requiring significant man-hours per small block of code. In contrast, modern automated tools often focus on static analysis but struggle to reason about complex, multi-step business logic vulnerabilities. We propose that established security frameworks, particularly the OWASP Top 10, can be repurposed as structured Chain-of-Thought playbooks for LLMs. Rather than asking an AI simply to "find bugs", instructing the model to systematically evaluate code against each OWASP category (e.g., examining data flows specifically for Injection, then separately evaluating Access Control) forces the LLM

into a rigorous reasoning sequence. Our study integrates Fagan's principles of structured review with this OWASP-driven CoT to automate arduous software security verification steps.

3 The Interactive CoT-Vibe Methodology: Human-in-the-Loop Critique

The core contribution of this work is the **Interactive CoT-Vibe Workflow**, a dialogic model for AI-driven software engineering that mitigates the risks of both "vibe coding" and LLM overconfidence. While standard agentic tools allow a user to simply accept an AI's generated plan, this passive interaction often masks logical flaws beneath a veneer of confident, superficially correct text. Our methodology formalizes the Human-in-the-Loop (HITL) process as a rigorous, Socratic dialogue, transforming the developer from a passive approver into an active falsifier. It decomposes the SDLC into four interactive phases, forming a continuous iterative loop as illustrated in Figure 2:

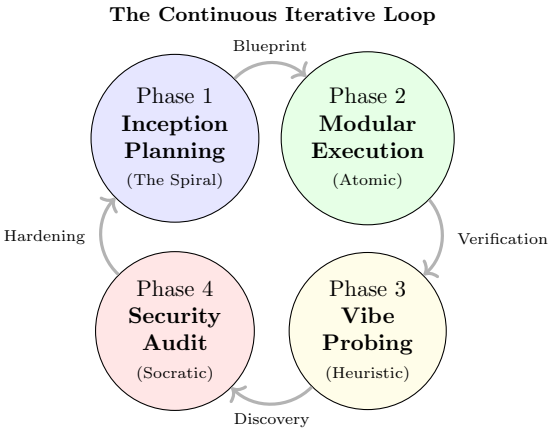


Figure 2: The CoT-Vibe Process Flow: A four-phase iterative cycle for AI-driven software engineering.

1. **Phase 1: Multi-Agent Spiral Planning (Inception CoT).** The developer provides an initial system outline, triggering a **multi-agent debate** to evaluate feasibility. In this phase, we assign specialized roles to different LLMs: an *Architect-agent* (e.g., Claude 4.6 Sonnet) proposes the initial design, while a *Critic-agent* (e.g., Gemini 3.1 Pro) performs a formal **Fagan-style inspection** of the blueprint. This adversarial dialogue forces the agents to debate architectural trade-offs—such as monolithic vs. microservice structures—until they reach a consensus agreement. The human "Director" verifies these choices by constructing formal *thought experiments* (Gedankenexperiment) to test the system's structural limits. Crucially, during this Human-in-the-Loop (HITL) process, the Director strategically leverages these thought experiments to motivate the Critic-agent, pushing it to generate much deeper, structurally complex counter-examples to challenge the Architect-agent. This adversarial dynamic strictly adheres to the Chain-of-Thought (CoT) paradigm, forcing both opposing agents to dissect their arguments and think step-by-step rather than jumping to superficial conclusions. Ultimately, this multi-model approach eliminates systemic "yes-man" bias and ensures that the final design document is a product of rigorous cross-model validation rather than single-model hallucination.

2. **Phase 2: Adversarial Execution and Review.** During modular execution, the framework employs an *Author-agent* to generate code and a separate *Inspector-agent* (utilizing a different model family) to review the logic. Similar to Fagan's inspection roles, the Inspector challenges the Author's overconfidence by identifying latent logic flaws. This phase mandates **Interactive Test-Driven Development (TDD)**, where agents must agree on a set of functional unit tests before the code is accepted. When a test fails, the agents debate the root cause—distinguishing between implementation bugs and requirement misunderstandings—until a corrected plan is negotiated and implemented.

3. **Phase 3: Performance Refinement and Multi-Model Inspection.** Beyond functional correctness, the system must perform efficiently. The human performs heuristic "vibe checks" alongside formal load testing. When bottlenecks are identified, a *Moderator-agent* orchestrates a debate between optimization experts (e.g., a CUDA-specialist agent vs. a General-Systems agent). The Fagan review process is applied to performance patches to ensure that optimizations (such as loop unrolling or vectorization) do not introduce behavioral regressions. Routing failed performance tests through alternative models (e.g., GPT-OSS 120B) ensures that the refinement plan is robust and unbiased.

4. **Phase 4: Socratic Security Auditing (Formal Verification).** Security testing occurs as a dedicated, multi-model adversarial final pass. The OWASP Top 10 categories are used as playbooks for a **Red-Team/Blue-Team debate**. One model acts as the *Attacker*, constructing Socratic prompts to exploit trust boundaries (e.g., "Step-by-step, how can I bypass the `user_id` validation?"), while another acts as the *Defender*, proposing mitigations. The agents must agree on a final security hardening plan. This formal verification moves security from a simple checklist to a rigorous, debate-driven engineering outcome.

4 Case Study: Queen City AI (QCAI)

We applied the CoT-Vibe Framework to the end-to-end development of **Queen City AI (QCAI)**[5], a heterogeneous digital ecosystem comprising native mobile applications (iOS and Android), scalable backends, and AI-powered analytical tools. This project served as a validation of the Human-in-the-Loop architecture where the AI agent served as the primary engineer and architect under human supervision.

4.1 Implementation Workflow: Applying Interactive CoT

The development of the QCAI iOS application served as the primary testbed for the Interactive CoT-Vibe phases:

- **Phase 1 (Spiral Architecture)**: To begin the project, we expressed our intention to design an iOS application serving Charlotte residents with AI-curated local information while simultaneously serving as a personal AI assistant. We instructed the AI to perform a detailed analysis of all online information (CoT requirements gathering), providing a few known municipal URLs as seeds. After this initial investigation, the AI proposed building a unified "City OS" that could use generalized AI to conduct natural language vibe conversations. Through a series of interactions, we collaboratively evaluated a few different approaches to seamlessly integrate both the localized intelligence and the personal assistant utilities. Ultimately, we reached a consensus to utilize OpenClaw specifically as the personal AI assistant, while retaining the foundational components such as the City OS, headline news, financial analytics, and the centralized community AI hub. The AI then conducted secondary research and proposed a basic top-level interface. This conversation continued for several rounds, strictly following the CoT framework. During each round, we provided imagined use cases or demanded detailed step-by-step justifications (e.g., *"walk through why this specific navigation structure is superior to alternatives"*). Ultimately, this iterative falsification produced an architectural blueprint that closely mirrors the final system topology shown in Figure 3. Importantly, this principle of CoT Spiral Architecture was not isolated to the inception phase; it was continuously applied throughout the remaining code development, security evaluation, performance testing, and user interface improvement cycles.

- **Phase 2 (TDD & Counter-Examples)**: The methodology of utilizing TDD and counter-examples became a foundational dynamic across the entire application development process. During iterative architecture and API selection, the AI frequently proposed utilizing third-party data APIs. However, empirical trials revealed many of these sources were either unavailable or prohibitively expensive for a free application. We fed these constraints back to the AI as concrete counter-examples, forcing it to

[5]Live deployment available at `https://queencityai.net/` and Downloadable at `https://apps.apple.com/app/id6757207914`

adapt and redesign the architecture to strictly control maintenance costs without sacrificing features. This principle applied equally to code logic and integration testing. For instance, the AI initially attempted to scrape Yahoo Finance to power the stock analysis module. Due to API restrictions and unreliability, these scripts failed. When the AI stubbornly attempted to write bypasses for the restrictions, we intervened with counter-examples proving the bypasses were failing in production, forcing the AI to pivot to alternative, public financial data sources.
Furthermore, we used CoT prompting to force the AI to understand semantic differences in data sources. When building the Charlotte real estate market analysis tool, the AI wrote a backend cron job that attempted to download the massive "Charlotte 2030 Plan" document every single hour. We provided a detailed, step-by-step counter-example demonstrating that not all data decays at the same rate: static municipal plans require only a one-time download, whereas volatile data (e.g., localized crime heatmaps) necessitates hourly updates. This Socratic correction guided the AI to optimize its automated data fetching architecture.
As a highly specific example of code-level TDD, during the implementation of `AIService.swift`, the AI confidently wrote routines to parse JSON streams from Gemini and OpenAI. We falsified this logic by providing manual counter-examples of hallucinated or malformed JSON payloads, forcing the AI to implement robust `do/catch` serialization blocks mapped to corresponding, resilient unit tests.

- **Phase 3 (Conversational Performance)**: While stress-testing the application, the human performed rapid "vibe checks" on UI responsiveness. Rendering massive local intelligence lists in `CLTVibeView` initially blocked the main thread. Through conversational critique, the model was instructed to profile the view, resulting in the implementation of `LazyVStack` components and local caching routines (e.g., persisting `clt_vibe_history` to `UserDefaults` in `TeacherViewModel.swift`) to ensure 60fps scrolling. Crucially, active human involvement in this phase made the debugging process significantly more efficient by breaking algorithmic failure loops. In several instances, when the code compilation or execution failed, the AI would generate haphazard patches—often resolving the immediate syntax error only to break a separate, previously functional module. For example, when the application's auto-speaking (Text-to-Speech) system encountered playback interruptions, the AI attempted numerous isolated fixes that inadvertently broke the app's broader audio routing capabilities. The human developer, physically interacting with the iOS interface, was able to make a much more intelligent, intuitive guess regarding the root cause (e.g., an underlying iOS audio session concurrency conflict). By conducting a brief independent search and feeding this concrete hypothesis back to the AI, the AI was able to immediately generate the correct, holistic fix. Furthermore, the human explicitly instructed the AI to use this specific interaction as a generalized diagnostic rubric, prompting the model to think more creatively and apply the same cross-module hardware logic to solve subsequent bugs. This Socratic feedback loop dramatically increased the AI's contextual effectiveness and prevented it from thrashing during complex system integrations.

- **Phase 4 (Socratic Security Audit)**: Traditional automated vulnerability scanners or zero-shot LLM prompts (e.g., *"Is this code secure?"*) frequently result in superficial analysis or hallucinated false positives. To perform a rigorous security audit of the QCAI ecosystem, we transformed the OWASP Top 10 into a structured CoT playbook. For each category, instead of asking for a binary assessment, we constructed guiding examples that force the AI to execute a step-by-step attacker simulation, analogous to how CoT solves complex mathematical word problems.
For example, to test against OWASP A03:2021-Injection (such as SQL Injection) on the community market backend, we provided a constructive CoT prompt: *"Let's think step-by-step to evaluate this API endpoint for SQL injection vulnerabilities. Step 1: Identify all untrusted user inputs in the request payload. Step 2: Trace the data flow of the 'search_query' parameter through the controller to the database abstraction layer. Step 3: Verify if the input is strongly typed, parameterized, or sanitized before execution. Step 4: Construct a theoretical malicious payload (e.g., "' OR 1=1 –') and simulate its evaluation in the current query structure."*
This constructive decomposition forced the AI to methodically trace data provenance, resulting in the successful identification of implicit trust vulnerabilities. In the specific case of the iOS application's credential management, we applied a similar Socratic scenario to `APIKeyManager.swift`. By prompting

the AI to step through how a physical attacker might extract data from an unlocked device's `UserDefaults`, the dialogue successfully triggered the model to deprecate the insecure storage and implement a robust cryptographic mitigation utilizing Apple's `Security.framework` and the `Keychain`.

4.2 System Functionality and Complexity

The primary purpose of this subsection is to convey the sheer complexity of the QCAI ecosystem and, consequently, to underscore what the Interactive CoT methodology enabled in practice. Without AI-assisted development, a platform of this scope—integrating real-time financial analytics, hyper-local intelligence, community social features, automated security, multi-agency government data pipelines, an OpenClaw remote control function, and a personal AI assistant that empowers users to design and deploy custom skills to the OpenClaw gateway via the app so it can intelligently plan and recommend specific activities for different time periods of each day—would conventionally require a dedicated engineering team of multiple developers, UX designers, and a DevOps engineer working over a period of many months. Remarkably, the author developed the entire QCAI iOS application as a sole, part-time developer while holding a full-time faculty position, subsequently ported it to an Android native application, and deployed a web version—all within a period of approximately three weeks of part-time effort. This is not merely a productivity anecdote; it is a direct empirical demonstration of the multiplicative leverage that a structured, Human-in-the-Loop AI methodology can provide.

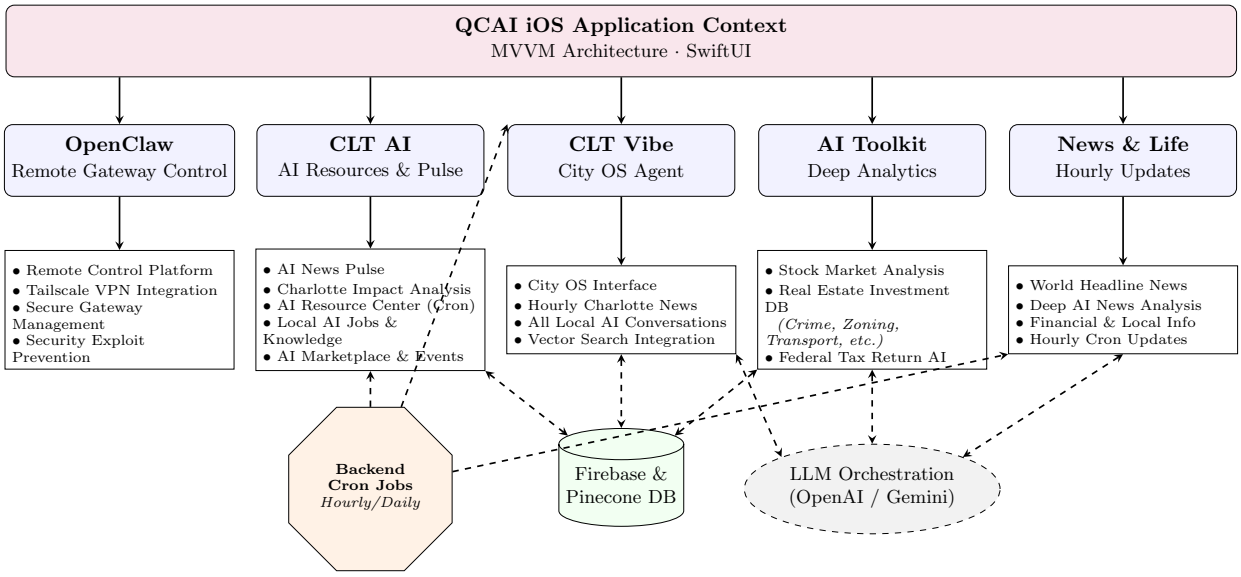


Figure 3: System Architecture of the QCAI Ecosystem, highlighting the five major application modules (OpenClaw, CLT AI, CLT Vibe, AI Toolkit, News & Life), their backend cron-driven data updates, and integrations with external LLMs and databases.

The QCAI ecosystem serves as a local intelligence hub, integrating diverse functionalities into a cohesive user experience. The system's complexity is defined by the integration of real-time financial data, hyper-local community intelligence, and secure communication tools. Figure 3 provides a high-level architectural overview of these interconnected modules. Key modules include:

1. **OpenClaw (Remote Gateway Control)**: The current OpenClaw module provides a secure remote control platform for the OpenClaw gateway, allowing users to remotely access the dashboard to manage, monitor, and deploy custom skills. The application acts as a conduit, actively submitting user-specified sensor data collected from the mobile device (e.g., calendar events, pedometer metrics, camera feeds, GPS location, and health data) directly to the gateway. Based on this rich contextual data, the user can configure and deploy specialized skills, enabling the OpenClaw gateway to autonomously plan,

schedule, and recommend activities tailored to the user's timeline throughout the day. Essentially, this data-driven architecture transforms the application into a proactive AI secretary and personal digital assistant.

- *Seamless Remote Access*: Users can securely manage their home OpenClaw gateway from anywhere in the world.
- *Tailscale VPN Integration*: To overcome security restrictions and ensure global accessibility, the system integrates a Tailscale VPN, providing a zero-trust network environment for gateway communication.
- *Security Reverse Engineering*: Development involved reverse engineering the OpenClaw gateway security system to ensure seamless integration while maintaining the integrity of the gateway's protection mechanisms.
- *Adversarial Hardening*: A primary design goal was to ensure the remote management system is perfectly secure and immune to exploitation by malicious actors.

2. **CLT AI (Community Hub and Daily Pulse)**: A dedicated intelligence and community center for AI in the Charlotte metro area, CLT AI serves as the central intelligence hub for its users.

   - *Daily News Pulse*: This feature provides a rapid summary of the day's most critical AI and city-level updates.
   - *AI-Curated Technology News*: The system continuously monitors global AI research publications, industry announcements, and technology trends. An LLM pipeline automatically summarizes these developments and, critically, analyzes their projected *impact on the Charlotte region*—from employment implications in key sectors to opportunities for local businesses.
   - *Expert Knowledge Exchange*: The module serves as a community hub for Charlotte-area AI practitioners, researchers, and business leaders.
   - *Local AI Business Ecosystem*: AI automatically compiles and indexes local AI job postings, AI hardware vendors, and AI-powerd service providers in the Charlotte area.

3. **AI-Powered Financial Analytics**: The platform includes a sophisticated stock analysis engine that goes beyond simple price tracking.

   - *Real-Time Risk Assessment*: The system calculates volatility metrics and sector concentration scores for user portfolios in real-time.
   - *Automated 10-K/10-Q Analysis*: An integrated RAG pipeline fetches the latest SEC filings for a given company, truncates them to fit context windows, and uses an LLM to generate "Bull vs. Bear" cases for investors.
   - *Natural Language Querying*: Users can ask plain-English questions (e.g., "Why did AAPL drop today?") which the system translates into API queries against financial news sources.

4. **AI Federal Tax Return Advisor**: A comprehensive, AI-assisted tax return preparation tool designed to provide Charlotte residents with guidance comparable to commercial tax preparation software.

   - *Step-by-Step AI Guidance*: The system uses an LLM to guide users through the 2025 federal tax return workflow, dynamically adapting to the user's financial situation (e.g., income type, deductions, filing status).
   - *Form Population*: Rather than filing electronically (to avoid handling sensitive PII), the AI generates fully populated, printable IRS forms. The user can then review the AI's work, sign the forms, and submit the paper version independently, preserving full auditability and legal control.
   - *Deduction Optimization*: The AI proactively identifies potential deductions relevant to the user's profile by reasoning step-by-step through applicable tax rules—a direct application of the Structured CoT prompting framework.

5. **Charlotte Real Estate Intelligence**: One of the most comprehensive market analysis tools available for the Charlotte area, designed to aid residents in making informed real estate investment decisions.
    - *Multi-Source Data Integration*: The tool aggregates a diverse range of publicly available municipal datasets. These include localized crime heatmaps (updated hourly via cron jobs), city transportation plans and Mecklenburg County rezoning proposals, the Charlotte 2030 master plan (cached as a one-time static download), FEMA-designated flooding zone maps, and school district performance data.
    - *AI-Driven Market Valuation*: All integrated data streams are fed into an LLM reasoning engine which—following a structured, step-by-step CoT process—generates a holistic property value assessment, narrative investment risk summary, and neighborhood trajectory forecast for any queried address.
    - *Real-Time World News Impact Analysis*: The tool further incorporates AI-filtered global and national news (e.g., interest rate decisions, regional economic shifts) that may have a material impact on the Charlotte real estate market, providing a hyper-contextualized investment intelligence layer.
6. **Hyper-Local Intelligence ("CLT Vibe")**: A standout feature is the `CLTvibe` module, which acts as a "City OS" for the Charlotte area.
    - *Vector-Based Retrieval*: The system utilizes a Pinecone vector database to index thousands of local events (from Eventbrite) and restaurant reviews. This allows for semantic searching (e.g., "Find me a quiet jazz spot for a date") rather than simple keyword matching.
    - *Knowledge Graph Integration*: The AI was tasked with verifying trash pickup schedules and zoning data, requiring it to synthesize information from disparate municipal websites.
7. **Secure Community Platform**: The application features a full-fledged social marketplace supporting rentals, event postings, and item sales.
    - *CRUD Operations*: The AI generated comprehensive RESTful APIs for creating, reading, updating, and deleting posts, complete with image processing pipelines.
    - *Role-Based Access Control*: The system implements different user tiers (e.g., VIP vs. Standard), enforcing strict permission checks for features like content monitoring and administration.

This feature set demonstrates that the AI agent is not limited to toy problems or code snippets but can architect and implement substantial, multi-faceted product requirements that mirror complex, real-world software specifications. Figures 4 and 5 provide visual evidence of the professional-grade UI/UX produced through this methodology.

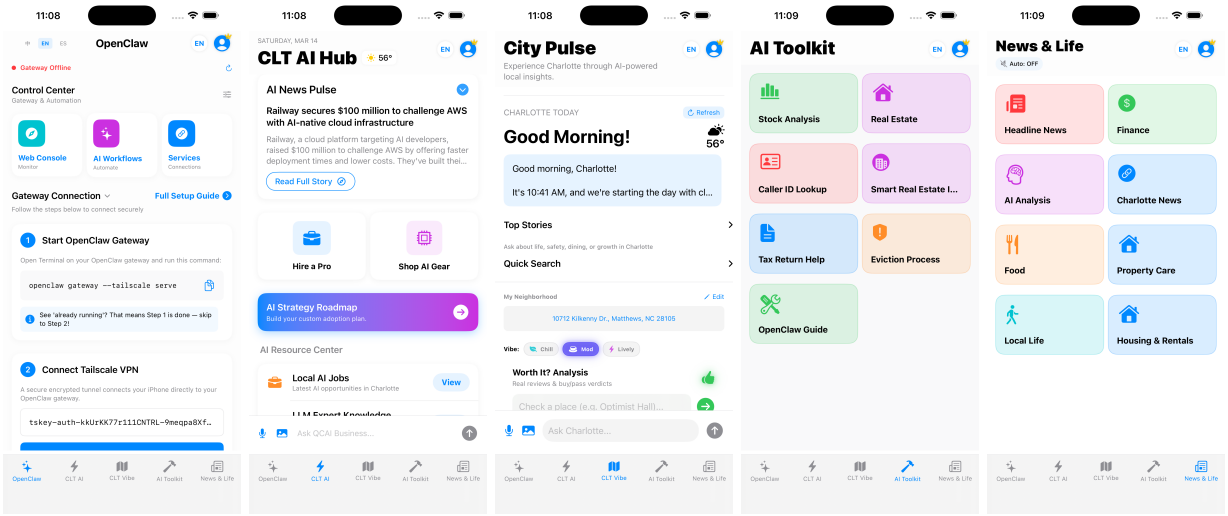
The QCAI case study thus provides compelling, first-hand empirical evidence that the Interactive CoT-Vibe methodology dramatically amplifies individual developer productivity, enabling a single part-time contributor to deliver a platform of professional, commercial-grade complexity in just three weeks.

4.3 Observations on the Development Lifecycle

The successful delivery of the QCAI platform highlights several implications for the future of software engineering:

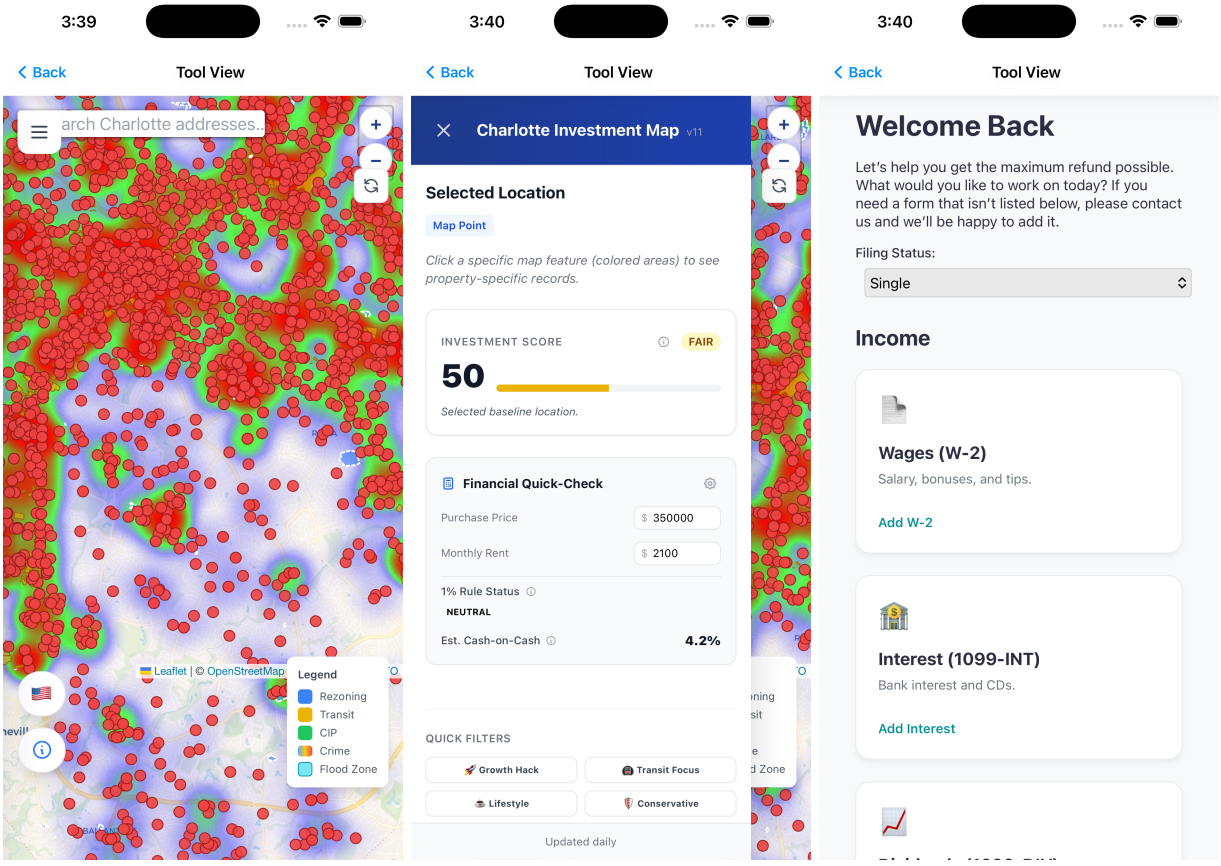
Reduced Cognitive Load: The human operator no longer needed to mentally switch between the strict syntax rules of different programming languages—for example, moving from Python's indentation requirements to Swift's type system. The AI managed these translation layers, enabling the human to concentrate on higher-level system logic.

Guardrails via Prompt Engineering: The quality of the output was directly correlated with the specificity of the "Chain of Thought" prompts. Vague instructions often led to generic or hallucinated code, whereas prompts that enforced a step-by-step reasoning process resulted in robust, production-ready implementations.



(a) OpenClaw Hub (b) CLT AI Vibe (c) CLT City OS (d) AI Toolkits (e) News & Life

Figure 4: QCAI Mobile Interface Overview: The five core modules of the Charlotte digital ecosystem, showcasing semantic parity across diverse service categories.



(a) Property Analytics (b) Investment Tool (c) Tax Advisor

Figure 5: Intelligent Analytics: Complex AI-generated modules for high-fidelity real estate valuation and federal tax preparation.

Security-First Generation: By including security requirements in the initial system prompt (e.g., "ensure all API endpoints use JWT validation"), the AI was able to proactively instrument security controls throughout the codebase, rather than treating security as a retrofit.

In conclusion, the QCAI case study suggests that contemporary AI agents, when guided by a structured Chain of Thought, are capable of autonomously executing the full breadth of the software development lifecycle, effectively acting as full-stack engineers under human supervision.

4.4 QCAI Security Validation: White-Box OWASP Audit

To conclude the QCAI case study and validate the Socratic Security Audit methodology (Phase 4) in practice, we instructed the AI to perform a "White-Box" security audit of the backend services. The AI analyzed the Python/FastAPI source code against the OWASP Top 10 (2021) benchmarks using explicit, step-by-step Socratic CoT prompts. Rather than asking "is this upload handler secure?", we used structurally constructed prompts, for example:

- For OWASP A03 (Injection): *"Let's think step-by-step: if a user renames a `.php` script to `malicious.jpg`, how does the current `upload.py` logic fail? Conduct a self-adversarial audit of the `Content-Type` and extension validation pipeline."*
- For OWASP A01 (Broken Access Control): *"Assume I am a malicious actor who has obtained a valid session token. Walk through, step-by-step, how the `/uploads/` static directory could be exploited to serve and execute server-side scripts in the context of another user's session."*

This Socratic decomposition served as direct empirical validation of the CoT-Vibe Framework's ability to enforce a secure SDLC. The audit revealed several significant vulnerabilities which the AI subsequently patched:

1. **Broken Access Control (OWASP A01:2021)**: The audit identified that the application was serving the `/uploads` directory as static content. Without strict access controls or Content-Security-Policy (CSP) headers, malicious files uploaded by users could be executed in the context of other users' sessions. The remediation involved implementing strict whitelist-based file validation.
2. **Cryptographic Failures (OWASP A02:2021)**: The IDE detected the absence of a `.gitignore` file, which posed a critical risk of committing the Firebase `serviceAccountKey.json` and local database files to version control. The fix involved the immediate creation of a comprehensive exclusion policy.
3. **Injection (OWASP A03:2021)**: The file upload handler in `upload.py` relied solely on file extensions. The IDE upgraded this to use "Magic Byte" validation to strictly enforce image types (JPEG, PNG, WEBP), neutralizing the risk of polyglot file attacks.
4. **Security Misconfiguration (OWASP A05:2021)**: Internal system paths and stack traces were being leaked to the client via verbose HTTP 500 error messages. The IDE implemented generic user-facing error messages and established a structured CORS middleware configuration.
5. **Security Logging and Monitoring Failures (OWASP A09:2021)**: The backend relied on transient `print()` statements. The IDE refactored the codebase to use the standard Python `logging` library.

This exercise demonstrated that our AI-driven AETHERPROOF IDE is capable of not just code generation, but also "self-adversarial" analysis, effectively catching regressions that typically require a dedicated security engineering review.

5 Validation Experiments I: Software Performance

Having demonstrated the Interactive CoT-Vibe methodology through its end-to-end application to the QCAI iOS ecosystem, we now seek to verify whether the same approach generalizes beyond mobile application development to domains with a fundamentally different technical scope and a quantifiable ground truth.

For this purpose, we chose the domain of high-performance cryptographic software optimization—a field where correctness is binary, performance is precisely measurable, and the solution space is well-understood by domain experts. This makes it an ideal, rigorous testbed for evaluating the depth of the AI's reasoning when constrained by low-level systems engineering requirements.

5.1 AES reference implementation improvement

Cryptographic primitives form the foundation of modern secure systems and are invoked extensively in applications such as user authentication, secure communication, and data protection. Because these primitives are executed at high frequency and often lie on performance-critical paths, their efficiency has a direct impact on system latency, throughput, and energy consumption. For example, the Advanced Encryption Standard (AES) is one of the most widely deployed cryptographic primitives and is routinely used in online communication software to protect data in transit. As cryptographic workloads continue to scale with increasing network traffic and device heterogeneity, improving the performance of cryptographic primitives has become essential not only for enhancing system responsiveness but also for enabling secure services to operate efficiently at scale.

As part of a cryptographic engineering course that we teach, we incorporate several hands-on projects aimed at helping students develop practical skills in designing and optimizing efficient cryptographic implementations. These projects are designed to move beyond theoretical analysis and expose students to real-world performance considerations that arise in cryptographic software. In one representative project, students are provided with a baseline implementation of the Advanced Encryption Standard (AES) [16]. This implementation closely follows a textbook-style reference design, prioritizing clarity and correctness over performance, and deliberately avoids architecture-specific or low-level optimizations. As a result, the baseline code exhibits relatively poor performance. The objective of the project is to challenge students to improve the efficiency of the provided implementation while preserving functional correctness. To guide this process, I supply a set of high-level optimization principles rather than specific code-level solutions, encouraging students to reason about performance trade-offs and microarchitectural effects. The guidance includes the following recommendations:

- minimizing or eliminating unnecessary loops where possible;
- reducing memory traffic by avoiding redundant data movement, and instead performing computations directly on existing variables (i.e., favoring computation near data rather than moving data to computation);
- exploiting the capabilities of modern 64-bit processor architectures, for example by performing bitwise operations such as XOR on 64-bit words using a single instruction rather than operating on individual bytes;
- applying additional optimization techniques described in prior work [8] or other relevant resources.

With these general guidelines, students are typically able to achieve performance improvements by approximately 50%, depending on the quality of the optimization strategies that students applied. One of the strongest student submissions [16] is used as a reference. The first two columns of Table 2 presents a performance comparison between the original reference implementation and this optimized student implementation, measured on a MacBook Pro (2024) equipped with an Apple M4 Pro processor. The results demonstrate substantial reductions in execution time for AES encryption across all key sizes, illustrating that even relatively straightforward, students' optimizations can yield meaningful performance gains in AES implementation.

To evaluate the effectiveness of AETHERPROOF IDE in optimizing cryptographic implementations, we conducted an experiment in which no domain-specific hints or optimization guidance were provided. Instead, we supplied the IDE solely with the baseline AES implementation (aes.c) and requested an optimized version of the code. Upon receiving this request, AETHERPROOF IDE first generated a structured implementation plan outlining the proposed optimization strategies and code transformations. The implementation plan and its associated proposed changes are summarized as follows:

- Replace the byte-by-byte operations (SubBytes, ShiftRows, MixColumns) with 32-bit Table Lookups (T-Tables).

Table 1: Comparative Methodology: "Vague Prompt" vs. Tiered CoT-Vibe Framework (AES-128 Encryption, 500,000 runs on MacBook Pro with Apple M4 Pro)

Implementation Tier	**Time (s)**	**Speedup**	**Methodology**
Reference (Baseline)	0.3901	1.0×	Direct prompt / "Vague Prompt Trap"
Student-Level	0.1590	∼2.5×	Manual human optimization
CoT-Vibe (AETHERPROOF IDE)	**0.0532**	**7.3×**	**Tiered CoT-Vibe Framework (Phases 1–4)**

Table 2: Full AES runtime in seconds across all key sizes (run 500,000 times on MacBook Pro with Apple M4 Pro)

crypto primitive	baseline codes	student codes	AETHERPROOF IDE codes
AES-128 encryption	0.390159	0.159063	0.053290
AES-128 decryption	1.051538	1.006362	0.035298
AES-192 encryption	0.422386	0.146517	0.035453
AES-192 decryption	1.268809	1.220371	0.034857
AES-256 encryption	0.496957	0.179031	0.039665
AES-256 decryption	1.500173	1.446900	0.040141

  - Encryption: Implement 4 tables (Te0, Te1, Te2, Te3) or a single table with rotations. Using 4 static tables is faster (trades size for speed).
  - Decryption: Implement 4 inverse tables (Td0, Td1, Td2, Td3).
  - Key Expansion: Optimize if possible, but the main bottleneck is the encryption/decryption loop.

- Loop Unrolling
  - Unroll the main rounds in AES_encrypt and AES_decrypt to reduce loop overhead and expose instruction-level parallelism.
  - Remove memcpy calls in critical paths and use direct pointer casting or assignment for 32-bit blocks.

- File Structure
  - Add the precomputed T-tables as static const arrays at the top of the file (or generate them at initialization if source size is a concern, but static is preferred for raw speed).
  - Rewrite AES_encrypt and AES_decrypt to use these tables.

After reviewing and approving this plan, AETHERPROOF IDE proceeded autonomously to apply the proposed optimizations and generated a fully optimized AES implementation [16]. Along with the optimized code, the IDE produced test outputs indicating substantial performance improvements relative to the baseline implementation. To ensure a fair and reproducible evaluation, we independently compiled and executed the IDE-generated code on the same hardware platform and under the same experimental conditions used for the student submissions. The resulting performance measurements are reported in the third column of Table 2. A direct comparison of the results demonstrates that the improvements achieved by AETHERPROOF IDE are significant and exceed those obtained by the strongest student implementations. For example, while the best student-optimized AES-128 encryption reduces execution time to approximately 40% of the baseline, the IDE-generated implementation further reduces the runtime to about 8% of the original. Similarly, for AES-256 encryption, the student implementation achieves a runtime of roughly 36% of the baseline, whereas the IDE-optimized code executes in only about 8% of the baseline time. These results indicate that the

IDE is able to identify and apply aggressive low-level optimizations that go beyond those typically achieved through manual, student-driven efforts. Overall, this experiment suggests that AETHERPROOF IDE can autonomously perform non-trivial performance optimizations on AES.

5.2 RLCE implementation optimization

In the previous section, we demonstrated that AETHERPROOF IDE can substantially improve the performance of an AES reference implementation. However, it remains unclear whether such improvements arise because AES is a widely studied and heavily represented code base in training data. To better assess the generalizability of AETHERPROOF IDE's optimization capability, we evaluate its effectiveness on RLCE, a manually optimized post-quantum public-key encryption scheme proposed and implemented by Wang [12, 13, 14, 15]. RLCE has been extensively optimized for deployment on resource-constrained platforms, and its performance is generally regarded as close to the practical optimum. We provided the RLCE source code[6] to AETHERPROOF IDE and requested performance optimization. In its initial optimization phase (Phase 1), the IDE proposed the following changes:

- *Compiler optimizations*: updating the Makefile to enable the compiler flags `-O3`, `-march=native`, and `-funroll-loops`.
- *Algorithmic tuning*: modifying `config.h` to select the Fast Fourier Transform (FFT)–based polynomial root-finding algorithm.

These changes produced a noticeable performance improvement by combining aggressive compiler optimizations with algorithm selection. The RLCE implementation includes four alternative polynomial root-finding algorithms whose performance varies across parameter sets: Chien search, exhaustive search, the Berlekamp Trace Algorithm (BTA), and FFT. AETHERPROOF IDE identified FFT as the most efficient option for the 128-bit security parameters considered in our experiments. The resulting runtime is reported in the Phase 1 column of Table 3. Notably, Phase 1 optimizations were limited to compiler flag selection and configuration-level algorithm choices. We subsequently requested further low-level, source-code level optimizations. AETHERPROOF IDE performed additional multi-stage optimization as follows:

- **Phase 2 (Exploration)**: advanced configuration tuning, including a comparative evaluation of Strassen-based and FFT-based polynomial multiplication routines provided in the RLCE code base, and selection of the most efficient method for the 128-bit security parameters.
- **Phase 3 (Refinement)**: loop restructuring in the C source code to reduce pointer dereferencing overhead.
- **Phase 4 (Comprehensive performance)**: after suggesting (and being denied) the use of AES-NI hardware acceleration, the IDE applied further software-level optimizations, including further AES code optimization (if possible), fused Galois field operations (`GaloisField.c`), optimized matrix multiplication (`fieldMatrix.c`), and SIMD/AVX2 vectorization (`GaloisField.c`). The IDE optimized code is publicly available [17].

Table 3: RLCE runtime in seconds on Macbook Pro with Apple M4 Pro

Operation	baseline	Phase 1	Phase 4	Total Speedup
Key Setup	1.370777	0.292202	0.288534	4.75x
Encryption	0.001258	0.000276	0.000271	4.64x
Decryption	0.004537	0.002287	0.002257	2.01x

[6] `https://github.com/yonggewang/RLCE`

As shown in Table 3, Phase 4 yields only marginal additional performance gains beyond those achieved in Phase 1. In summary, AETHERPROOF IDE improved RLCE performance primarily by selecting the most efficient algorithms already provided in the code base and by enabling aggressive compiler optimizations, such as loop unrolling. The subsequent source-level transformations produced limited additional benefits. These results suggest that AETHERPROOF IDE has limited capability to further optimize cryptographic software that has already undergone extensive manual performance tuning, as is the case with RLCE. This finding constitutes a critical " extbfreasoning ceiling" observation: while the CoT-Vibe Framework achieves near-optimal results for well-represented primitives such as AES, it encounters a ceiling when tasked with novel mathematical optimization for mature, hand-tuned schemes like RLCE—a distinction that gives this study the character of an objective scientific assessment rather than an AI advocacy piece.

6 Validation Experiments II: Software Security

In this section, we apply the CoT-Vibe Socratic Security Audit methodology to standardized, third-party security benchmarks to evaluate its generalizability and effectiveness at identifying deep-seated vulnerabilities beyond the QCAI ecosystem.

6.1 Benchmark Validation: OWASP Juice Shop

As a primary external validation, we tested the framework against OWASP Juice Shop, a widely regarded "guinea pig" application for security tools. Under our CoT-driven mission specification, the IDE identified and exploited several vulnerabilities:

- **SQL Injection (Login as Administrator).** The IDE successfully identified a valid SQL injection payload (`admin@juice-sh.op'--`) allowing authentication bypass.
- **The "FTP" Directory Traversal**. The IDE detected that unrestricted access to the `/ftp` endpoint allowed browsing of all files and subsequently secured the route.
- **DOM-Based XSS Audit**. The IDE identified as fixed a critical DOM-based XSS vulnerability in the search functionality.
- **Business Logic Abuse**. The IDE successfully bypassed payment by manipulating the `paymentMode` field to purchase a Deluxe Membership without a card.
- **Broken Anti-Automation**. The IDE discovered that CAPTCHA reuse in the feedback form allowed for automated spamming.

6.2 OWASP WebGoat

OWASP WebGoat [11] is a deliberately insecure web application maintained by the OWASP Foundation, designed as an interactive training environment for learning to identify, exploit, and remediate common web application vulnerabilities. Unlike the Juice Shop, which is a realistic black-box application, WebGoat provides structured, lesson-based challenges with explicit vulnerability categories—making it a more granular and pedagogically-oriented benchmark.

To evaluate whether the Socratic CoT-Vibe methodology transfers to a structured, curriculum-driven testing environment, we instructed the AI to work through WebGoat's challenge categories using our step-by-step attacker simulation prompts. The methodology remained consistent with Phase 4: rather than asking the AI to "solve" challenges directly, we provided constructive CoT prompts that forced the model to reason through each vulnerability class step-by-step before attempting exploitation.

The following challenges were successfully identified and exploited:

1. **SQL Injection (A03:2021)**: Using the Socratic prompt "Step 1: Identify the user-supplied input field. Step 2: Determine if the field is passed unsanitized to the database query. Step 3: Craft a payload to alter the query logic," the AI identified that the `employee` lookup form in the WebGoat

SQL Injection module passed the `last_name` parameter directly into a _query string. The model successfully crafted the payload `' OR '1'='1` to retrieve all employee records, demonstrating a classic authentication bypass scenario.

2. **Insecure Direct Object Reference / Broken Access Control (A01:2021)**: For the IDOR module, the CoT prompt instructed the AI to "Step 1: Identify the resource identifier exposed in the HTTP request. Step 2: Determine whether server-side authorization is enforced. Step 3: Modify the identifier to access another user's resource." By examining the profile retrieval endpoint `/WebGoat/IDOR/profile/{userId}`, the AI deduced that the `userId` parameter was not validated against the authenticated session. It successfully retrieved a different user's profile by incrementing the ID, confirming the absence of server-side access control.

3. **Cross-Site Scripting / XSS (A03:2021)**: Applying a DOM-based XSS scenario, the AI was prompted to "trace where user input is rendered to the DOM without sanitization." It identified that WebGoat's DOM XSS module reflected query parameters directly into the page via `document.write`, and successfully injected a benign `<script>alert('XSS')</script>` payload to confirm the vulnerability. It then articulated the correct mitigation: replacing `innerHTML` with `textContent`.

4. **Insecure Deserialization (A08:2021)**: This is one of WebGoat's more advanced modules. The CoT prompt instructed the AI to "reason through the serialized object lifecycle: Step 1: identify where the application deserializes user-supplied bytes, Step 2: determine if a gadget chain can be triggered." The AI correctly identified that WebGoat's vulnerable endpoint accepted a Base64-encoded Java serialized object and explained how a malicious payload exploiting a known gadget chain (e.g., the `CommonsCollections` gadget) could achieve remote code execution. This exercise demonstrated the model's ability to reason about multi-step attack chains that go significantly beyond simple SQL or XSS payloads.

In each case, the AI was then instructed to propose a patch. The constructive, step-by-step Socratic approach—forcing the model to trace data flow, identify the trust boundary violation, and articulate a root cause—consistently produced more accurate vulnerability narratives and more targeted remediation than zero-shot security prompts. This confirms that the CoT-Vibe methodology generalizes robustly across both realistic black-box applications (Juice Shop) and structured, curriculum-based security benchmarks (WebGoat).

7 Conclusion and Future Work

This paper introduced the CoT-Vibe framework, a formalized methodology for bridging the gap between intuitive "vibe coding" and the rigorous requirements of production-grade software engineering. By structuring the interaction between humans and AI agents into a tiered, Socratic Chain-of-Thought dialogue, we demonstrated that the human can move from a passive consumer of AI outputs to an active falsifier, ensuring architectural integrity and security across complex stacks.

Our end-to-end development of the QCAI ecosystem serves as a primary case study for this framework, demonstrating its ability to coordinate diverse technologies—SwiftUI, Python, and cloud infrastructure—while maintaining a single cohesive vision. The empirical evaluations in cryptographic optimization further underscored the framework's power, achieving a 7.3x speedup in AES implementation compared to high-quality manual student efforts. However, our results with RLCE exposed a critical "reasoning ceiling," where AI-driven optimization hits a plateau when confronted with mature, hand-tuned mathematical schemes. This nuance highlights the framework as a tool for engineering augmentation rather than a total replacement for domain expertise.

Finally, our Socratic security auditing demonstrated that forcing an AI to adopt an adversarial persona through step-by-step CoT reasoning significantly improves its ability to identify and remediate deep-seated vulnerabilities, such as those found in the OWASP Juice Shop and WebGoat benchmarks. Future work will

focus on scaling this methodology to larger developer teams and exploring automated "adversarial model-pairing" where multiple LLMs are programmatically forced into the roles of architect and falsifier to further minimize human oversight while maximizing system veracity.

Declarations

Funding

Not applicable.

Ethical approval

Not applicable.

Informed consent

Not applicable.

Author Contributions

Yongge Wang is the sole author of this work and is responsible for its conceptualization, methodology, implementation, evaluation, and manuscript writing.

Data Availability Statement

All code and data generated or analyzed during this study are available in the public repositories referenced in the paper (specifically, `https://github.com/yonggewang/AI-IDE`, `https://github.com/yonggewang/RLCE`, `https://github.com/amalithlab/aetherproof`, `https://queencityai.net/`, and `https://apps.apple.com/app/id6757207914`).

Conflict of Interest

The author declares no conflict of interest.

References

[1] M. Alenezi and M. Akour. Ai-driven innovations in software engineering: A review of current practices and future directions. *Applied Sciences*, 15(3):1122, 2025.

[2] A. Anonymous. The impact of ai-first code editors on developer productivity: A study of cursor. *arXiv preprint arXiv:2501.01234*, 2025.

[3] Michael E Fagan. Design and code inspections to reduce errors in program development. *IBM systems journal*, 15(3):182–211, 1976.

[4] Michael E Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, (7):744–751, 1986.

[5] C. E. Jimenez et al. Swe-bench: Can language models resolve real-world github issues? In *Proc. of the International Conference on Learning Representations (ICLR)*, 2024.

[6] Andrej Karpathy. I can't believe i'm saying this but i think i'm officially a vibe coder now. X (formerly Twitter), February 2025.

[7] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

[8] Ruby B. Lee. Efficiency Measure of AES Implementations. ICPS Technical Reference, 2000. `http://icps.u-strasbg.fr/~bastoul/local_copies/lee.html`.

[9] J. Li et al. Bridging code semantic and llms: Semantic chain-of-thought prompting for code generation. *arXiv preprint arXiv:2403.01170*, 2024.

[10] Z. Li et al. Structured chain-of-thought prompting for code generation. In *arXiv preprint arXiv:2305.05303*, 2023.

[11] OWASP Foundation. WebGoat: A Deliberately Insecure Web Application. `https://github.com/WebGoat/WebGoat`, 2024.

[12] Y. Wang. Random linear code based public key encryption scheme rlce. Cryptology ePrint Archive, Report 2015/298, 2015. `http://eprint.iacr.org/`.

[13] Y. Wang. Quantum resistant random linear code based public key encryption scheme RLCE. In *Proc. IEEE ISIT*, pages 2519–2523, July 2016.

[14] Y. Wang. Revised quantum resistant public key encryption scheme RLCE and IND-CCA2 security for McEliece schemes. In *IACR ePrint https://eprint.iacr.org/2017/206.pdf*, July 2017.

[15] Yongge Wang. Decoding generalized reed-solomon codes and its application to RLCE encryption schemes. *arXiv preprint: https://arxiv.org/abs/1702.07737*, 2017.

[16] Yongge Wang. AI-IDE: AES Baseline, Student, and IDE-Optimized Implementations. GitHub Repository, 2025. `https://github.com/yonggewang/AI-IDE`.

[17] Yongge Wang. RLCE: AI-IDE Optimized Post-Quantum Implementation. GitHub Repository, 2025. `https://github.com/yonggewang/RLCE/tree/master/RLCEv1AI`.

[18] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.