Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

VibeSec: A Dual-Mode Security Architecture for AI-Generated Web Applications

Lalit Barik, Aditya Sasmal
Department of Computing Technologies
SRM Institute of Science and Technology
Kattankulathur, Tamil Nadu, India

Dr. G. Balamurugan
Assistant Professor
Department of Computing Technologies
SRM Institute of Science and Technology
Chennai, Tamil Nadu, India

***Abstract*—The emergence of AI-powered code generation platforms has democratized software development, enabling rapid application prototyping through natural language interfaces. However, this paradigm shift introduces significant security challenges, as generated code may contain vulnerabilities inherited from training data or introduce novel flaws through architectural misunderstandings. This paper presents VibeSec, a dual-mode security architecture that systematically evaluates AI-generated web applications through two complementary approaches: proactive review before code is saved and reactive analysis after code is delivered. The proactive mode employs a security reviewer agent that evaluates generated code before persistence, iteratively regenerating with feedback when vulnerabilities are detected. The reactive mode provides post-generation analysis comprising dependency auditing and LLM-based logical analysis within isolated execution environments. A fix-via-AI mechanism enables contextual remediation by sending vulnerability details back to the code generation agent for targeted, structure-preserving fixes. The architecture extends an existing open-source AI code generation platform built on Next.js, tRPC, and E2B sandboxes. We evaluate VibeSec across 20 experimental runs spanning 10 prompts under both conditions. Results show proactive mode achieves 100/100 security scores for 4 of 10 prompts but struggles with authentication and authorization patterns. The fix-via-AI mechanism improved scores in 4 of 6 cases but degraded security in 2 cases, introducing critical vulnerabilities during remediation. An extended evaluation with detailed functional prompts showed that explicit functionality requirements do not improve and may worsen security outcomes. Pattern-matching static analysis (Semgrep) produced zero findings across all test cases, establishing that AI-generated code requires semantic analysis. A significant false positive was identified: the dependency audit incorrectly attributed sandbox template infrastructure vulnerabilities to generated code. Generated applications frequently exhibited functional defects (non-functional buttons, infinite loading states) representing a distinct failure mode orthogonal to but equally important as security. VibeSec represents a significant advancement toward trustworthy AI-generated software by embedding comprehensive security validation directly into the development pipeline without disrupting user experience.**

***Index Terms*—AI Code Generation, Dual-Mode Security, Proactive Review, Reactive Analysis, LLM-Based Analysis, Sandbox Environment, Vulnerability Assessment**

I. Introduction

The proliferation of large language models (LLMs) capable of generating functional code has given rise to a new category of development platforms. Tools such as Lovable, Bolt, and similar AI-powered coding environments allow users to create full-stack applications through conversational interfaces, dramatically reducing development time and lowering barriers to entry for non-technical users. These platforms interpret natural language prompts and generate production-ready code, typically for modern web frameworks like Next.js.

Despite their transformative potential, AI code generation platforms introduce unique security challenges that demand systematic investigation. LLMs are trained on vast corpora of publicly available code, which includes vulnerable patterns, deprecated practices, and security anti-patterns. Research has demonstrated that LLMs can inadvertently reproduce these vulnerabilities, generating code susceptible to SQL injection, cross-site scripting, insecure authentication mechanisms, and other common weaknesses [1]. Furthermore, generated applications often incorporate third-party dependencies that may contain known vulnerabilities, and the logical structure of AI-generated code may introduce business logic flaws that traditional security tools fail to detect.

The gap between code generation and security validation represents a critical vulnerability in the AI development workflow. Current platforms typically prioritize functional correctness over security assurance, leaving users unaware of potential risks in their generated applications. This paper addresses this gap by introducing VibeSec, a dual-mode security architecture that systematically evaluates AI-generated applications through both proactive and reactive approaches.

The principal contributions of this research are:

- A dual-mode security architecture (proactive and reactive) for AI-generated applications, extending an existing open-source code generation platform [18]
- A proactive security reviewer that evaluates code before saving, with intelligent regeneration and repeated-issue detection
- A reactive security analysis pipeline comprising dependency auditing and LLM-based logical analysis
- A fix-via-AI mechanism that sends vulnerability context back to the code generation agent for targeted, structure-preserving fixes
- A quantitative risk scoring methodology that aggregates findings from heterogeneous analysis tools

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

This research establishes a foundation for trustworthy AI-generated software by demonstrating that systematic security validation can be embedded directly into the development pipeline while maintaining the seamless user experience that defines modern AI coding platforms.

II. Related Work

A. AI Code Generation and Security Implications

The security implications of LLM-generated code have attracted increasing research attention. Pearce et al. [2] conducted a comprehensive study examining whether LLMs produce secure code, finding that models frequently generate vulnerable code even when explicitly prompted for secure implementations. Their analysis revealed that models trained on public repositories inadvertently learn insecure patterns present in their training data. Similarly, Khoury et al. [3] evaluated code produced by GitHub Copilot and identified security vulnerabilities in approximately 40% of generated code snippets across multiple programming languages.

Research by Perry et al. [4] explored the effectiveness of security-focused prompting strategies, demonstrating that carefully crafted prompts can reduce but not eliminate vulnerabilities in generated code. These findings underscore the need for automated security validation as an integral component of AI code generation platforms rather than relying solely on improved model training or prompting techniques.

B. Static Analysis for Vulnerability Detection

Static analysis tools have long been employed to identify security vulnerabilities without executing code. Tools such as Semgrep, ESLint with security plugins, and commercial solutions provide pattern-based detection of common vulnerability classes [5]. Research by Johnson et al. [6] evaluated the effectiveness of static analysis tools for web application vulnerabilities, finding that while these tools excel at detecting known patterns, they struggle with context-dependent vulnerabilities and business logic flaws.

The integration of static analysis into CI/CD pipelines has been widely studied, with research demonstrating that early vulnerability detection significantly reduces remediation costs [7]. However, the application of static analysis to AI-generated code presents unique challenges. Our evaluation of Semgrep with community rule sets on AI-generated React applications revealed that pattern-matching static analysis is insufficient for detecting the architectural vulnerabilities prevalent in AI-generated code, which require semantic understanding to identify. This finding motivates the reliance on LLM-based analysis as the primary logical assessment layer in VibeSec's reactive pipeline.

C. Dependency Security Management

Modern web applications typically rely on numerous third-party packages, making dependency security a critical concern. The npm ecosystem, in particular, has faced challenges with vulnerable and malicious packages [8]. Tools like npm audit and Snyk provide automated vulnerability scanning for dependencies, correlating package versions against vulnerability databases including the National Vulnerability Database (NVD) and GitHub Advisory Database.

Research by Zimmermann et al. [9] analyzed the security implications of the npm dependency network, revealing that transitive dependencies create complex attack surfaces. Their findings emphasize the importance of comprehensive dependency auditing for AI-generated applications, which may incorporate packages without thorough consideration of their security posture.

D. LLM-Based Code Analysis

Recent work has explored using LLMs themselves for code analysis and vulnerability detection. Studies by Cheshkov et al. [10] demonstrated that LLMs can identify certain vulnerability classes with accuracy comparable to specialized tools, particularly for logical flaws that static analyzers miss. The ability of LLMs to understand code context and intent makes them well-suited for detecting business logic vulnerabilities and security anti-patterns that manifest at the architectural level.

However, research also indicates that LLM-based analysis has limitations, including inconsistent performance across vulnerability types and susceptibility to prompt engineering effects [11]. These findings motivate the multi-layered approach adopted in VibeSec, where LLM analysis complements rather than replaces traditional security tools.

E. Research Gap and Contribution

While existing research has examined individual security assessment techniques, limited work has addressed their integration into AI code generation workflows. Current platforms lack systematic security validation, leaving users unaware of vulnerabilities in their generated applications. Furthermore, no prior work has explored a dual-mode approach that combines proactive security review (before code is saved) with reactive post-generation analysis. VibeSec addresses this gap by proposing a dual-mode architecture that intercepts the code generation workflow at two critical points, providing comprehensive security validation without disrupting the user experience.

III. System Architecture

A. Platform Overview

VibeSec extends Vibe [18], an open-source AI coding platform where users interact with a chat interface to generate web applications through natural language prompts. The base platform provides the code generation pipeline, comprising a GPT-5.4-based code agent orchestrated through Inngest Agent Kit, E2B sandbox execution environments, file management, and the conversational user interface. The base platform handles sandbox provisioning, code generation through tool-using agents (terminal, file creation, and file reading tools), and fragment persistence.

Our contribution is the security architecture layered on top of this platform, intercepting the generation workflow at two critical points: before code is saved (proactive mode)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

and after code is delivered (reactive mode). The complete workflow encompasses user interaction through a React-based interface, type-safe API communication via tRPC, background job orchestration through Inngest, sandboxed code execution via E2B, and the dual-mode security validation described in the following sections.

B. Dual-Mode Security Architecture

The VibeSec security architecture implements two complementary modes that can be toggled per project:

1) Proactive Security Mode: In proactive mode, generated code is reviewed for security vulnerabilities *before* it is saved to the database. The proactive mode augments the code generation agent's system prompt with security-prescribing instructions that specify secure alternatives for common vulnerability patterns (e.g., using environment variables instead of hardcoded secrets, Zod validation instead of unvalidated input, DOMPurify instead of raw HTML rendering).

After code generation, a SecurityReviewerAgent powered by GPT-5.4 reviews the generated files. The reviewer is instructed to flag only exploitable vulnerabilities rather than theoretical concerns, with explicit accuracy rules to reduce false positives (e.g., not flagging Zod-validated code as missing validation, not flagging hardcoded URL parameters as user input).

If vulnerabilities are found, the system enters a regeneration loop with a maximum of three attempts. The reviewer generates structured feedback that is appended to the code agent's prompt, enabling the agent to fix the identified issues while preserving the application's functionality. A repeated-issue detection mechanism tracks previously flagged issues across regeneration attempts by matching on category, file, and line proximity. If all issues in a review are repeated from previous attempts, the code is accepted as best-effort rather than entering an infinite regeneration loop.

If all regeneration attempts fail, the system falls back to reactive mode: the code is still saved, but the user receives a notification explaining that the security reviewer could not resolve all issues and that the code may contain vulnerabilities.

2) Reactive Security Mode: In reactive mode, code is generated and saved immediately without pre-save review. Users can then trigger a security analysis on demand, which runs a multi-step analysis pipeline comprising two analysis phases:

Dependency Audit. The system executes npm audit within the sandbox environment, parsing the JSON output to extract vulnerability information including severity classification, affected packages, and remediation guidance.

LLM-Based Logical Analysis. A GPT-5.4-mini agent analyzes the codebase for logical vulnerabilities that static tools cannot identify, including business logic flaws, authorization bypasses, data exposure in API responses, missing rate limiting, and cryptographic weaknesses. The agent receives the complete codebase with prioritized ordering of server-side files.

We initially evaluated Semgrep as a static analysis layer within the reactive pipeline, testing it with the `auto` configuration and multiple community rule sets (typescript, react, owasp-top-ten, secrets, security-audit) on AI-generated React applications. Across all test cases, Semgrep produced zero vulnerability findings despite the presence of confirmed vulnerabilities detected by the LLM analyzer, including hardcoded secrets in localStorage, client-side authentication checks, and plaintext password handling in object literals. This result demonstrates that pattern-matching static analysis is insufficient for AI-generated code, which tends to produce architectural vulnerabilities requiring semantic understanding rather than syntactic pattern matching to detect. Accordingly, the reactive pipeline relies on dependency auditing and LLM-based analysis as its two primary assessment layers.

Each analysis phase runs as a separate Inngest step, enabling individual observability, retry, and error resilience. If one analyzer fails, the remaining analyzers still produce results, which are merged into the final assessment.

3) Fix-via-AI Mechanism: When the reactive analysis identifies vulnerabilities, users can invoke the fix-via-AI mechanism, which sends the vulnerability details back to the code generation agent through the chat interface. Unlike regex-based auto-fix approaches, this mechanism leverages the code agent's contextual understanding to produce complete, working fixes.

To preserve the existing file structure during fixes, the system injects a FIX_MODE_PROMPT into the agent's system prompt when modifying an existing application. This prompt includes the list of existing files from the most recent fragment, informing the agent about the project structure and instructing it to read relevant files before making changes. The agent connects to the same E2B sandbox (identified by its sandbox ID), reads the current code, and applies targeted fixes rather than consolidating the application into a single file.

After the fix is applied, a new fragment is saved with the updated files, and the security analysis automatically re-runs to verify the fixes, providing the user with an updated security score.

4) Risk Scoring and Aggregation: The final step synthesizes findings from both analysis phases into a unified security assessment. A quantitative risk scoring algorithm combines individual findings using empirically calibrated weights and category multipliers, producing an overall risk score between 0 and 100.

Findings are persistently stored for historical analysis, trend identification, and continuous improvement of the security pipeline. The aggregated results power an interactive security dashboard that presents vulnerability information in an accessible format suitable for both technical and non-technical users.

C. Technology Stack

The VibeSec architecture is built on the following technology stack:

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

TABLE I: VibeSec Technology Stack Components

Component	Purpose
Next.js 15	Provides React-based frontend with App Router for optimized page routing
tRPC	Enables end-to-end type-safe API communication between frontend and backend
Inngest Agent Kit	Orchestrates asynchronous background jobs and all OpenAI API calls for code generation, security review, and analysis
Prisma	Provides type-safe database access with automatic migrations and query building
E2B	Creates isolated sandbox environments for secure code execution
GPT-5.4	Powers the code generation agent via Inngest Agent Kit
GPT-5.4	Powers the security reviewer (proactive mode) via Inngest Agent Kit
GPT-5.4-mini	Powers the logical analysis (reactive mode) via Inngest Agent Kit
Clerk	Provides authentication and user management

The base code generation platform (Next.js, tRPC, Inngest, Prisma, E2B, Clerk) was developed by Erdeljac [18]. Our contribution extends this platform with the dual-mode security architecture, including the proactive reviewer, reactive analysis pipeline, fix-via-AI mechanism, and risk scoring system.

D. Data Model Design

The system's data model captures the relationships between users, projects, generated code, and security assessments. Core entities include:

Users represent platform accounts with authentication handled through external providers. Each user may own multiple projects, which encapsulate a complete application generation context including chat history and generated files.

Projects serve as the primary organizational unit, containing all artifacts associated with a particular application generation effort. Each project maintains its file structure, conversation history with the AI assistant, a toggle for security mode selection (proactive or reactive), and associated security reports.

Fragments capture the generated code artifacts, including the sandbox URL for live preview, file contents, and sandbox identifier for reuse during fix operations.

Security scans capture the results of reactive analysis for a specific message, including component scores from each analysis layer, detailed findings with severity classifications, the aggregated risk score, and vulnerability counts by severity.

Messages store the conversation between users and the AI assistant, with support for standard result messages, error messages, and security-rejected messages that inform users when proactive review could not resolve all issues.

IV. Implementation Methodology

A. System Integration Approach

The security architecture integrates into the existing code generation workflow through strategic interception points. When a user submits a prompt through the chat interface, the request flows through the tRPC API layer to trigger an Inngest background job. This job orchestrates the complete generation and security assessment workflow:

The process begins with sandbox creation or connection. For initial generation, an isolated E2B environment is provisioned with a pre-configured Next.js template. For fix operations, the system connects to the existing sandbox identified by its sandbox ID, retrieves the file list from the most recent fragment, and injects the FIX_MODE_PROMPT into the agent's system prompt.

The code agent receives the user prompt and generates code within the sandbox, writing files directly to the isolated environment using tool calls (terminal, createOrUpdateFiles, readFiles). The agent is orchestrated through Inngest Agent Kit with GPT-5.4, supporting up to 15 tool-use iterations.

In proactive mode, after code generation completes, the SecurityReviewerAgent reviews the generated files. If vulnerabilities are found, the regeneration loop initiates, providing structured feedback to the code agent. Repeated-issue detection prevents infinite loops by accepting code when all remaining issues have been previously flagged.

In reactive mode, code is saved immediately. Users can trigger the security analysis pipeline, which runs as a 5-step Inngest function: connect-sandbox, read-files, dependency-audit, llm-analysis, and merge-findings. Each step is individually visible and retryable in the Inngest dashboard.

Results from both analysis phases are aggregated and stored in the database before being returned to the frontend for display. This asynchronous workflow ensures that users receive comprehensive security feedback without blocking the generation process.

B. Proactive Security Review Implementation

The proactive security reviewer is implemented as a SecurityReviewerAgent class that wraps a GPT-5.4 agent via Inngest Agent Kit. The reviewer receives the generated files with line numbers and a system prompt that includes:

Explicit accuracy rules to reduce false positives, including instructions to only flag exploitable vulnerabilities, to recognize existing validation (e.g., Zod safeParse), and to consider the data source (hardcoded constants are not user input).

A previous-issues block that lists issues flagged in prior regeneration attempts. This context prevents the reviewer from re-flagging the same issues that the code agent was already instructed to fix, unless the code at the flagged location has not changed.

The reviewer returns a structured JSON response with an overall secure/insecure assessment, a list of issues (each with category, title, severity, file, line, description, and fix suggestion), and a summary. If the review indicates the code is insecure, a feedback message is generated and appended to the code agent's prompt for the next regeneration attempt.

The regeneration loop tracks all previously flagged issues. For each new review, the system checks whether each issue matches a previous issue by category, file, and line proximity (within 10 lines). If all issues in a review are repeated from previous attempts, the code is accepted as best-effort. If the maximum of three regeneration attempts is exhausted without passing, the system falls back to reactive mode.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

C. Dependency Audit Implementation

Dependency auditing executes within the sandbox environment to ensure accurate resolution of the complete dependency tree. The process begins by checking for a package.json file within the generated file structure. When one exists, the system executes npm audit within the sandbox, capturing the JSON output for analysis.

A critical implementation detail concerns npm audit's exit code behavior: when vulnerabilities are found, npm audit returns a non-zero exit code, which the E2B sandbox environment raises as a CommandExitError. The pipeline handles this by catching the error and extracting the stdout containing the JSON vulnerability report, rather than treating the non-zero exit code as a failure.

The audit results are parsed to extract vulnerability information for each affected package, including severity classification, remediation guidance, and affected versions. npm severity levels (critical, high, moderate, low, info) are mapped to the system's four-tier severity scale. Findings are categorized by severity and aggregated for risk scoring.

For applications without package manifests, the dependency layer returns an empty findings set with appropriate status indicating the absence of dependency management.

D. LLM-Based Logical Analysis Implementation

The LLM analysis layer employs a GPT-5.4-mini agent via Inngest Agent Kit with a specialized prompt that guides the model toward consistent, actionable security assessments. The analysis prompt includes:

A structured code context prepared by prioritizing server-side files (API routes, server actions, middleware, library utilities) over client-side components, while including all source files (not just server-side code) to ensure comprehensive coverage. This ensures detection of client-side vulnerabilities such as XSS via dangerouslySetInnerHTML and sensitive data stored in localStorage. The context has a 50,000-character limit to fit within the model's context window.

Six analysis dimensions: business logic flaws (IDOR, race conditions, privilege escalation), authentication and authorization logic (route bypasses, session inconsistencies, role confusion), data exposure (sensitive fields in API responses, information leaks in error messages), API security (missing rate limits, mass assignment, improper CORS), cryptographic issues (weak hashing, missing salts, predictable random values), and state management issues (client-side trust, TOCTOU vulnerabilities).

The prompt instructs the model to return findings in a structured JSON array with severity classifications, location references, exploit scenarios, and remediation recommendations. This structured output enables automated integration with the risk scoring system.

E. Fix-via-AI Implementation

The fix-via-AI mechanism addresses a key limitation of automated security remediation: regex-based approaches produce fragile, line-by-line replacements that often break code structure and fail to handle context-dependent fixes. Instead, the fix-via-AI mechanism sends vulnerability details back through the chat to the code generation agent, which understands the full application context.

When a user initiates a fix, the system creates a user message containing formatted vulnerability details (severity, title, file location, and fix suggestion for each finding) and sends a code-agent/run event with the existing sandbox ID. The code agent connects to the same sandbox, reads existing files, and applies targeted fixes.

To prevent the common failure mode where the agent consolidates a multi-file application into a single page.tsx, the system injects a FIX_MODE_PROMPT that includes the list of existing files from the most recent fragment. This prompt informs the agent about the project structure, instructs it to read relevant files before making changes, and specifies that app/page.tsx must remain the main entry point. After fixes are applied, the security analysis automatically re-runs to verify the results.

F. Risk Scoring Methodology

The risk scoring system combines findings from both analysis layers into a unified quantitative assessment. Each finding receives a severity classification based on its potential impact and exploitability:

Critical findings represent vulnerabilities that could lead to complete system compromise, data breaches, or unauthorized access to sensitive functionality. These include remote code execution, authentication bypass, and exposure of sensitive credentials.

High severity findings indicate vulnerabilities with significant impact but requiring specific conditions or user interaction to exploit. These include cross-site scripting, SQL injection with limited impact, and insecure direct object references.

Medium severity findings encompass information disclosure, missing security headers, and vulnerabilities requiring authenticated access or other preconditions.

Low severity findings include minor information leaks, outdated dependencies without known exploits, and deviations from security best practices without immediate exploitation potential.

The overall risk score is computed as:

$$\text{RiskScore} = \max\left(0, 100 - \sum_{i=1}^{n} w_{s_i} \cdot m_{c_i}\right)$$

Where w_{s_i} is the penalty weight for the severity of finding i (critical: 25, high: 15, medium: 5, low: 1), and m_{c_i} is the category multiplier for the finding's vulnerability category. Categories with higher exploitability or impact receive higher multipliers (e.g., authentication: 1.5, SQL injection: 1.5, XSS: 1.3, secrets management: 1.4, input validation: 1.2).

The final score is categorized into risk levels for intuitive user communication: low risk (90–100), medium risk (70–89), high risk (50–69), and critical risk (below 50).

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

V. Results and Evaluation

A. Experimental Setup

To evaluate the VibeSec architecture, we tested it against a corpus of AI-generated Next.js applications. The test corpus comprised 5 security-sensitive prompts designed to naturally elicit vulnerable code patterns: (P1) a user authentication system with localStorage-based sessions and an admin dashboard, (P2) a blog platform with search and a contact API, (P3) an e-commerce product catalog with shopping cart and admin inventory management, (P4) a user profile management app with messaging, and (P5) a data dashboard with API key configuration settings.

Each prompt was tested under two conditions: (1) reactive mode (post-generation analysis only) and (2) proactive mode (pre-save review with regeneration). This yielded 10 total experimental runs. All applications were generated using GPT-5.4 (temperature 0.1) within E2B sandboxes, with the reactive analysis pipeline (dependency audit + LLM analysis via GPT-5.4-mini) applied to all runs. For reactive runs, the fix-via-AI mechanism was invoked once after the initial analysis. Each prompt was intentionally brief to simulate real-world usage patterns on AI code generation platforms.

B. Security Scores by Condition

Table II presents the security scores across all 10 experimental runs, excluding template-level dependency findings (see Finding 6). The reactive condition reports pre-fix and post-fix scores (after one fix-via-AI cycle), while the proactive condition reports the score after regeneration completes.

TABLE II: Security Scores by Prompt and Condition (Excluding Template Dependencies)

Prompt	Condition	Score	Crit.	High
P1: Auth	Reactive (pre-fix)	0	3	1
	Reactive (post-fix)	20	1	2
	Proactive	5	2	2
P2: Blog	Reactive (pre-fix)	51	0	2
	Reactive (post-fix)	70	0	1
	Proactive	100	0	0
P3: E-commerce	Reactive (pre-fix)	51	0	2
	Reactive (post-fix)	5	2	2
	Proactive	45	0	2
P4: Profile	Reactive (pre-fix)	100	0	0
	Proactive	100	0	0
P5: Dashboard	Reactive (pre-fix)	100	0	0
	Proactive	100	0	0

C. Proactive Security Review Effectiveness

The proactive reviewer's effectiveness varied substantially across prompt types. Table III summarizes the regeneration behavior and outcomes.

TABLE III: Proactive Security Review Outcomes

Prompt	Regen.	Score	Vulns.	Outcome
P1: Auth	2	5	6	Best-effort accept
P2: Blog	0	100	0	Passed first attempt
P3: E-commerce	2	45	5	Best-effort accept
P4: Profile	1	100	0	Passed first attempt
P5: Dashboard	0	100	0	Passed first attempt

The proactive reviewer achieved a perfect security score of 100/100 for three of five prompts (Blog, Profile, Dashboard), eliminating all code-level vulnerabilities. For these prompts, the reviewer either passed the code on the first attempt or after a single regeneration, demonstrating strong effectiveness for applications that do not inherently require insecure architectural patterns.

However, for prompts that inherently require complex security patterns (authentication, authorization, admin panels), the proactive reviewer struggled. For P1 (Auth), the reviewer accepted code after 2 regeneration attempts with 2 persistent high-severity issues (client-side authentication, hardcoded admin credentials), yielding a score of 5. For P3 (E-commerce), the reviewer accepted after 2 attempts with 5 remaining vulnerabilities (score: 45), including client-side admin controls and unvalidated inventory actions. These results suggest that the proactive reviewer's effectiveness is bounded by the code agent's ability to generate secure alternatives for inherently complex security patterns within the regeneration loop.

A notable discrepancy was observed for P1: the UI warning displayed "2 persistent low-risk findings," while the reviewer output contained 2 HIGH severity issues. This indicates that the repeated-issue detection mechanism's summary does not accurately reflect the severity of accepted issues, potentially misleading users about the risk level of the generated code.

D. Fix-via-AI Mechanism Results

The fix-via-AI mechanism was tested on the three reactive runs where vulnerabilities beyond the dependency issue were found (P1, P2, P3). Table IV summarizes the outcomes.

TABLE IV: Fix-via-AI Outcomes (Code-Level Vulnerabilities Only)

Prompt	Pre	Post	Δ	Fixed	New	Net
P1: Auth	0	20	+20	6	5	+1
P2: Blog	51	70	+19	5	4	+1
P3: E-commerce	51	5	−46	5	6	−1

The fix-via-AI mechanism produced mixed results. For P1 (Auth) and P2 (Blog), the mechanism improved the security score (+20 and +19 respectively) by resolving original vulnerabilities, though both runs introduced new vulnerabilities during the fix process. The net vulnerability count decreased by 1 in each case.

Critically, for P3 (E-commerce), the fix-via-AI mechanism *degraded* security, reducing the score from 51 to 5 by introducing 2 critical vulnerabilities (client-controlled privilege escalation and forgeable session cookies) while attempting to

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

fix the original issues. The fixed code replaced client-side localStorage storage with server-side cookie-based sessions but implemented them insecurely, demonstrating that the code agent can introduce more severe vulnerabilities while attempting remediation.

For P4 (Profile) and P5 (Dashboard), the fix-via-AI mechanism was not applicable as no code-level vulnerabilities were found (scores of 100/100). These prompts naturally produce secure code that does not require remediation.

E. Vulnerability Distribution by Category

Table V presents the distribution of code-level vulnerabilities across security categories for all runs, as reported by the LLM analyzer (excluding template-level dependency findings). Categories are assigned by the analyzer based on the primary security domain affected by each finding.

TABLE V: Code-Level Vulnerability Distribution by Category (LLM Analyzer Classification)

Category	Reactive (Pre)	Reactive (Post)	Proactive
Authentication	1	2	1
Authorization	6	6	5
API Security	2	4	0
Input Validation	1	0	2
Secrets Management	0	1	0
Security (General)	3	2	3
Crypto	0	1	0
Total	**13**	**16**	**11**

Authorization issues (missing access controls, client-side privilege enforcement) constituted the largest vulnerability category across all conditions, comprising 46% of pre-fix findings. Notably, the reactive post-fix condition shows an increase in total vulnerabilities (13 $\rightarrow$ 16), reflecting the fix-via-AI mechanism's tendency to introduce new issues while resolving existing ones. API security vulnerabilities increased from 2 to 4 post-fix, while authorization issues remained constant at 6, suggesting that the code agent struggles most with implementing proper authorization boundaries. The proactive condition shows zero API security findings, indicating that the proactive reviewer effectively addresses this category.

F. Key Findings

The evaluation yields several significant findings:

Finding 1: Proactive mode effectiveness is prompt-dependent. The proactive reviewer achieved perfect results (100/100) for 3 of 5 prompts but failed to improve security for inherently complex prompts (Auth: 5/100, E-commerce: 45/100). The average proactive score across all prompts was 70, compared to the average reactive pre-fix score of 60.4.

Finding 2: Fix-via-AI can degrade security. The E-commerce fix reduced the security score from 51 to 5 by introducing critical vulnerabilities (privilege escalation, session forgery) while attempting to resolve existing issues. This demonstrates that agent-based remediation is not uniformly beneficial and can produce regressive outcomes for complex security patterns.

Finding 3: Some prompts naturally produce secure code. The Profile and Dashboard prompts achieved 100/100 in both reactive and proactive conditions, with zero code-level vulnerabilities detected. This suggests that prompts without authentication or authorization requirements naturally produce code with fewer architectural vulnerabilities.

Finding 4: The proactive reviewer can pass code with significant vulnerabilities. In 2 of 5 proactive runs (Auth, E-commerce), the reviewer's isSecure flag was set to true despite the presence of high-severity vulnerabilities. The repeated-issue detection mechanism accepted these as "best-effort," and the UI severity messaging was inaccurate, potentially misleading users.

Finding 5: Pattern-matching static analysis is insufficient for AI-generated code. Our evaluation of Semgrep with community rule sets produced zero findings across all test cases, despite the confirmed presence of critical vulnerabilities including hardcoded credentials, client-side authentication, and plaintext password storage. This establishes that AI-generated code produces architectural vulnerabilities requiring semantic understanding to detect, motivating the LLM-based analysis approach.

Finding 6: Sandbox template dependencies produce false positive security findings. The hono@¡4.12.14 vulnerability (HTML injection in hono/jsx SSR) appeared in all 10 runs, constituting the only consistent finding across the entire evaluation. However, investigation revealed that hono was not a dependency of the AI-generated code—it is a transitive dependency of Next.js's Turbopack dev server, installed as part of the sandbox template infrastructure. The dependency audit ran npm audit in the sandbox's home directory (/home/user), which contained the template's package.json and node_modules, rather than in the generated application's directory (/home/user/app). This caused template-level vulnerabilities to be incorrectly attributed to the generated code, inflating vulnerability counts and deflating security scores by 5 points across all runs. Furthermore, this false positive caused the fix-via-AI mechanism to fail on runs where hono was the only vulnerability detected: the code agent attempted to resolve the vulnerability via terminal commands (npm update), but the system treated the absence of file writes as an error condition. This finding demonstrates that dependency audits in sandboxed environments must explicitly scope their analysis to the generated code's dependency tree to avoid misattributing infrastructure vulnerabilities to user code.

G. Extended Evaluation (Round 2)

To supplement the initial evaluation with more detailed, functional prompts, we conducted a second round of experiments using 5 prompts (P6–P10) designed to target under-tested vulnerability categories: XSS (P6: Social Media Feed), file upload/secrets (P7: File Sharing), crypto/authorization (P8: Healthcare Portal), SSRF/secrets/crypto (P9: Financial Budget Tracker), and injection/authorization (P10: Real-time Chat). These prompts were substantially more detailed than Round 1 (5–8 sentences with explicit functionality requirements) and

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

explicitly mandated that all features be fully functional with real data persistence rather than static mockups.

Each prompt was tested under both reactive and proactive conditions, yielding 10 additional experimental runs (runs 11–20). Table VI presents the security scores for Round 2.

TABLE VI: Extended Evaluation Security Scores (Round 2, P6–P10)

Prompt	Condition	Score	Crit.	High
P6: Social Media	Reactive (pre-fix)	29	0	3
	Proactive	41	0	2
P7: File Sharing	Reactive (pre-fix)	11	1	2
	Proactive	0	1	3
P8: Healthcare	Reactive (pre-fix)	100	0	0
	Proactive	0	3	3
P9: Financial	Reactive (pre-fix)	15	0	4
	Proactive	35	0	3
P10: Chat	Reactive (pre-fix)	100	0	0
	Proactive	25	0	3

Round 2 Aggregate Results. The average reactive pre-fix score across Round 2 was 51.0, comparable to Round 1's 60.4. The average proactive score was 20.2, substantially lower than Round 1's 70.0. Fix-via-AI was applied in 3 of 5 reactive runs (P6, P9, P10), with mixed results: P6 improved from 29 to 67 (+38), P9 improved from 15 to 35 (+20), and P10 degraded from 25 to 0 (introduced critical vulnerabilities). The proactive mode fell back to reactive in 4 of 5 runs (P6, P7, P8, P10) after exhausting regeneration attempts, compared to 2 of 5 falls back in Round 1.

Comparison with Round 1. Table VII summarizes key metrics across both evaluation rounds.

TABLE VII: Comparison of Round 1 and Round 2 Results

Metric	Round 1	Round 2
Avg Reactive (pre-fix)	60.4	51.0
Avg Proactive	70.0	20.2
Fix-via-AI success rate	2/3	2/3
Proactive perfect scores (100)	3/5	1/5
Proactive fallbacks	2/5	4/5
Key difference	Round 2 prompts explicitly required functional apps	

Finding 7: Detailed functional prompts do not improve and may worsen security outcomes. Round 2 prompts were 5–8 sentences with explicit requirements for working features ("Every button and form must be fully functional – no placeholder or demo behavior"). Despite increased specificity, the average proactive score decreased substantially (70.0 → 20.2) and proactive fallbacks increased (2/5 → 4/5). This suggests that more detailed prompts may push the agent toward more complex architectures that are harder to secure, or that the current prompt engineering for security constraints is insufficient for functional applications.

Finding 8: Generated applications frequently lack functional correctness. A significant observation across both evaluation rounds was that generated applications often exhibited functional defects independent of security issues. In reactive mode, many applications rendered static placeholder UIs with non-functional buttons (e.g., login forms that accepted any input without validation, feature buttons that triggered no state changes, forms that displayed confirmation messages without persisting data). Some applications became unresponsive, stuck in loading states indefinitely. These functional failures occurred despite explicit prompt instructions requiring functional features and represent a distinct failure mode from security vulnerabilities. This observation aligns with broader industry findings that AI code generation platforms produce apps where "buttons don't work" – a usability concern that is orthogonal to but equally important as security assurance.

This functional quality issue has implications for security evaluation: an application that scores 100/100 on security may still be unusable if its core features do not function. Furthermore, static placeholder behavior (e.g., hardcoded demo credentials, mock data, non-connected forms) may reduce the apparent attack surface during security analysis, potentially producing inflated security scores for non-functional applications. Future security evaluation frameworks should consider incorporating functional correctness metrics alongside security assessment.

VI. Discussion

A. Architectural Effectiveness

The VibeSec architecture's effectiveness stems from its dual-mode approach to security assessment. The proactive mode addresses the root cause of vulnerabilities by intercepting insecure code before it is saved, while the reactive mode provides comprehensive post-generation analysis for cases where proactive review is not desired or has fallen back.

Our evaluation demonstrates that proactive mode achieves perfect security scores (100/100) for prompts that do not require inherently insecure architectural patterns (Blog, Profile, Dashboard), effectively preventing vulnerabilities from reaching the user. However, for prompts requiring complex security mechanisms (Auth, E-commerce), the proactive reviewer accepted code with significant vulnerabilities after exhausting regeneration attempts, highlighting that the approach is bounded by the code agent's capacity to generate secure alternatives.

The regeneration loop with repeated-issue detection represents a key design decision. Without it, the proactive reviewer and code agent can enter an infinite loop where the same vulnerabilities are flagged across regeneration attempts because the reviewer has no memory of previous feedback. The repeated-issue detection mechanism resolves this by accepting code when all remaining issues have been previously flagged, providing a practical best-effort approach. However, our evaluation revealed that the severity messaging for accepted issues can be misleading (e.g., displaying "low-risk" for high-severity findings), indicating a need for more accurate risk communication.

The fix-via-AI mechanism demonstrates that agent-based remediation outperforms regex-based auto-fix approaches in

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

some cases, achieving score improvements of +20 (Auth) and +19 (Blog). By sending vulnerability context to a code generation agent that understands the full application, the system produces working fixes that respect the application's architecture. The FIX_MODE_PROMPT ensures that multi-file applications maintain their structure during fixes, addressing a common failure mode where agents consolidate code into a single file. However, the E-commerce result (score degradation from 51 to 5) demonstrates that the mechanism can introduce more severe vulnerabilities while attempting fixes, representing a significant risk that must be communicated to users.

B. Proactive vs. Reactive Trade-offs

The dual-mode architecture exposes a fundamental trade-off. Proactive mode can prevent vulnerabilities from reaching the user entirely, achieving a 100/100 score for 3 of 5 test prompts. However, for the remaining 2 prompts, proactive mode produced scores of 5 and 45 respectively, both lower than the reactive pre-fix scores (0 and 51). This suggests that the regeneration process can sometimes produce worse code than the initial generation, as the code agent may introduce new vulnerabilities while attempting to address reviewer feedback.

The average proactive score (70.0) exceeds the average reactive pre-fix score (60.4) but is lower than the average reactive post-fix score (31.7 for fixable runs, or 60.4 including the 100-score runs where fix was not applicable). This nuanced result indicates that neither mode strictly dominates the other; rather, their effectiveness depends on the interaction between the prompt's security requirements and the code agent's ability to satisfy them.

The fallback mechanism—where proactive mode falls back to reactive when regeneration fails—ensures that users always receive generated code, even if the security reviewer cannot resolve all issues. This design choice prioritizes availability over absolute security, recognizing that a generated application with known vulnerabilities is more useful than no application at all.

C. Limitations

Several limitations warrant consideration. The evaluation used intentionally brief prompts to simulate real-world usage on AI code generation platforms. While this approach captures the typical user experience, it may produce less functional applications than would be generated with more detailed specifications, potentially affecting the vulnerability landscape. The generated applications may not be fully representative of production AI-generated code.

The proactive security reviewer's performance depends on the LLM's ability to accurately distinguish exploitable vulnerabilities from theoretical concerns. Our evaluation revealed that the reviewer's isSecure assessment can be inaccurate: in 2 of 5 proactive runs, the reviewer passed code that still contained high-severity vulnerabilities. False negatives in the reviewer allow vulnerable code to pass review, potentially creating a false sense of security.

The LLM analysis layer's performance depends significantly on prompt engineering and model selection. Different prompts or models might yield different detection rates, introducing variability into the system's performance.

The absence of static analysis from the pipeline, while motivated by empirical evaluation (Semgrep produced zero findings on AI-generated React code), represents a trade-off. Static analysis tools may still provide value for non-React code generation or for vulnerability classes not covered by community LLM training data. Future work could explore fine-tuned static analysis rules specifically designed for AI-generated code patterns.

The fix-via-AI mechanism relies on the code generation agent's ability to understand vulnerability context and produce correct fixes. Our evaluation demonstrated that this mechanism can degrade security (E-commerce: 51 $\rightarrow$ 5) by introducing critical vulnerabilities while attempting to resolve existing issues.

Our evaluation revealed a significant false positive issue in the dependency audit layer. The npm audit command ran in the sandbox's home directory, which contained the template's package.json and node_modules from the Next.js development server, rather than in the generated application's directory. This caused a transitive template dependency vulnerability (hono@¡4.12.14) to be incorrectly attributed to the generated code in all 10 experimental runs, deflating security scores by 5 points and causing fix-via-AI failures on runs where this was the only detected vulnerability. This finding highlights the importance of properly scoping dependency audits in sandboxed environments and has been addressed in the implementation by directing the audit to the generated application's directory.

The small sample size (5 prompts $\times$ 2 conditions = 10 runs) limits the statistical power of our conclusions. While the evaluation reveals important patterns, larger-scale studies are needed to generalize the findings across the broader landscape of AI code generation prompts.

D. Future Research Directions

Several promising directions for future research emerge from this work:

Integration of dynamic application security testing (e.g., OWASP ZAP) within the sandbox environment could complement the static and LLM-based analysis phases by detecting runtime vulnerabilities.

Real-time adaptation of analysis rules based on observed vulnerability patterns in AI-generated code could improve detection rates and reduce false positives over time.

Exploration of privacy-preserving analysis techniques could enable security assessment without exposing user code to external services.

Longitudinal studies comparing vulnerability patterns across different LLM code generation models could reveal whether newer models produce more secure code and inform the design of model-specific security rules.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

VII. Conclusion

This paper has presented VibeSec, a dual-mode security architecture for AI-generated web applications. The proposed framework addresses the critical gap between rapid AI-powered code generation and necessary security validation through two complementary approaches: proactive review before code is saved and reactive analysis after code is delivered.

Our evaluation across 5 security-sensitive prompts and 2 conditions (10 runs total) demonstrates that the proactive mode achieves perfect security scores (100/100) for prompts without inherently complex security requirements, but struggles with authentication and authorization patterns (5/100 and 45/100). The fix-via-AI mechanism improved security scores for 2 of 3 applicable runs (+20 and +19 points) but critically degraded security in one case (51 to 5), introducing critical vulnerabilities during remediation. The reactive analysis pipeline consistently identified vulnerabilities across all runs, with the LLM-based analysis detecting architectural flaws that pattern-matching static analysis (Semgrep) entirely missed, producing zero findings across all test cases. Additionally, we identified that the dependency audit incorrectly attributed sandbox template infrastructure vulnerabilities to generated code, producing consistent false positives across all runs and causing fix-via-AI failures on secure code.

The architecture extends an existing open-source code generation platform [18], demonstrating that comprehensive security validation can be embedded directly into AI code generation workflows without disrupting the seamless user experience. These results establish that dual-mode security assessment is feasible and beneficial for AI-generated applications, while highlighting that agent-based security remediation carries its own risks and that the effectiveness of proactive review is bounded by the code agent's ability to generate secure alternatives for complex patterns. This integration represents a significant step toward trustworthy AI-generated software and establishes a foundation for future research in this critical domain.

References

[1] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do Users Write More Insecure Code with AI Assistants?" in Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, 2023, pp. 2785-2799.

[2] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "An Empirical Cybersecurity Evaluation of GitHub Copilot's Code Contributions," in 2022 IEEE Symposium on Security and Privacy, 2022, pp. 1456-1470.

[3] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, "How Secure is Code Generated by ChatGPT?" in 2023 IEEE International Conference on Systems, Man, and Cybernetics, 2023, pp. 312-319.

[4] N. Perry, M. Srivastava, D. Boneh, and M. S. Bernstein, "Examining the Impact of AI Assistants on Developer Security Practices," in Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems, 2024, pp. 1-15.

[5] B. Chess and G. McGraw, "Static Analysis for Security," IEEE Security & Privacy, vol. 2, no. 6, pp. 76-79, 2004.

[6] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" in 2013 35th International Conference on Software Engineering, 2013, pp. 672-681.

[7] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with Applying Vulnerability Prediction Models," in Proceedings of the 2015 Symposium and Bootcamp on the Science of Security, 2015, pp. 1-9.

[8] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks," in International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, 2020, pp. 23-43.

[9] M. Zimmermann, C. A. Staicu, C. Tenny, and M. Pradel, "Small World with High Risks: A Study of Security Threats in the npm Ecosystem," in 28th USENIX Security Symposium, 2019, pp. 995-1010.

[10] A. Cheshkov, P. Zadorozhny, and R. Levichev, "Evaluation of ChatGPT Vulnerability Detection Capabilities," in 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering, 2023, pp. 543-547.

[11] S. Ahmed, M. M. Rahman, and S. I. Ahamed, "On the Limitations of Large Language Models for Code Vulnerability Detection," arXiv preprint arXiv:2310.08456, 2023.

[12] OWASP Foundation, "OWASP Top Ten Web Application Security Risks," 2021. [Online]. Available: https://owasp.org/Top10/

[13] S. Z. Almutairi and A. A. Almutairi, "A Systematic Review of Security Vulnerabilities in Web Applications Generated by Low-Code Platforms," IEEE Access, vol. 11, pp. 45231-45248, 2023.

[14] T. Schäfer, A. Apel, and D. Beyer, "Security Analysis of AI-Generated Code: A Case Study on GitHub Copilot," in 2024 IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1-12.

[15] R. Croft, M. A. Babar, and M. Kholoosi, "An Empirical Study of Security Issues in AI-Generated Code," ACM Transactions on Software Engineering and Methodology, vol. 33, no. 2, pp. 1-28, 2024.

[16] J. He, W. Ma, and Y. Chen, "LLMSec: A Framework for Evaluating Large Language Models in Security Code Generation," in Proceedings of the 2024 Network and Distributed System Security Symposium, 2024, pp. 1-17.

[17] L. Zhang, Y. Wang, and X. Liu, "Automated Security Assessment for AI-Generated Applications: Challenges and Opportunities," IEEE Transactions on Dependable and Secure Computing, vol. 21, no. 1, pp. 156-172, 2024.

[18] A. Erdeljac, "Vibe: AI App Generator," 2025. [Online]. Available: https://github.com/code-with-antonio/nextjs-vibe