Army of Juniors: The AI Code Security Crisis

How 10 critical anti-patterns in AI-generated code are systematically undermining software security at scale

OX Research | October 2025

Eyal Paz

Table of Contents

A Note from **OX Research**

In early 2025, as AI coding tools accelerated from curiosity to standard practice, organizations faced a question no one could answer: How do you secure code that's generated faster than humans can review it?

The industry's response was predictable: "Scan more. Detect more." But we suspected this couldn't be the answer.

As we witnessed the industry changing rapidly, we set out to better understand AI-generated code. Was it more flawed than human-written code? We (humanly) analyzed over 300 repositories: 50 explicitly built with AI coding tools, 250 human-coded baselines.

What we found wasn't what we expected.

We identified 10 critical anti-patterns in AI-generated code that revealed how AI fundamentally approaches problems. However, the vulnerability density per line of AI-generated code was similar to human code. The crisis wasn't code quality.

It was deployment velocity.

AI tools had removed every natural bottleneck that previously controlled what reached production. Traditional security—code review, shift-left scanning, post-deployment detection—couldn't scale to match. Even before AI adoption, organizations were already drowning—according to OX research, dealing with an average of 569,000 security alerts at any given time. Teams were overwhelmed before AI. Now, how could security possibly keep up?

The conclusion was inescapable: detection-led security is collapsing.

This research revealed that traditional security approaches fundamentally cannot keep pace with AI-generated code velocity. The 10 critical anti-patterns we identified, combined with the "insecure by dumbness" phenomenon, made it clear that security needed to evolve from reactive scanning to proactive, embedded intelligence.

Launched in September 2025, VibeSec represents our response: security embedded directly into coding workflows, preventing vulnerabilities before they exist rather than chasing them after deployment.

The following report documents what we found - and why it matters.

— OX Research Team, October 2025

Executive Summary: AI is Fast, Eager & Lacking Judgment

Software development has entered uncharted territory. For the first time in computing history, functional applications can be built faster than humans can properly evaluate them.

Our analysis of 300+ repositories reveals that AI-generated code behaves like an army of talented junior developers - fast, eager, but fundamentally lacking judgment.

Key Findings

1. AI-generated code exhibits 10 distinct anti-patterns that **directly contradict established software engineering best practices**, systematically compromising long-term maintainability, security, and scalability

2. Vulnerability density per line mirrors human code, but **deployment velocity has accelerated exponentially**

3. Non-technical users are shipping production systems **without security expertise**, creating applications that are **"insecure by dumbness"**

4. Organizations face a critical choice: adapt development practices or face a wave of **technical debt and security incidents**

The Real Crisis

The problem isn't that AI writes worse code - it's that **vulnerable systems now reach production at unprecedented speed**. Breaches at Replit, Lovable, and Tea App demonstrate this pattern materializing: developers using AI tools to rush applications to market without understanding fundamental security principles, exposing thousands of users to preventable risks.

Code Review Cannot Scale

Organizations must fundamentally restructure development roles. AI should handle implementation while humans focus on architecture, orchestration, and security oversight. **Traditional code review cannot scale with AI's output velocity** - instead, organizations need threat modeling, security guardrails embedded in AI workflows, and emerging solutions like AI-powered autonomous vulnerability remediation.

The transformation is accelerating. Organizations that fail to adapt will either fall behind competitively or drown in the technical debt and security incidents that AI's speed makes inevitable.

Next: The 10 Critical Anti-Patterns of AI-Generated Code

The 10 Critical Anti-Patterns of AI-Generated Code

Anti-Pattern	Core Issue	Root Cause	Primary Impact
Comments Everywhere | Note to Future AI Self	Excessive inline commenting beyond human norms	Memory architecture limitations; comments as AI navigation "breadcrumbs"	Internal AI tool, rather than human documentation, reveals model constraints
Avoidance of Refactors | Missing 'Who Wrote This?' Reflex	No instinctive code improvement process	Focus on immediate solutions vs. long-term quality; lacks semantic understanding	Technical debt accumulation; missed optimization opportunities
Over-Specification | Dispose-After-Use Code	Hyper-specific solutions lacking generalization	Training on pattern replication vs. abstract principles; knows "what", not "why"	Fragmented codebase; difficult maintenance and evolution
Return of Monoliths | Throwing Microservices Out	Consolidated structures over distributed architectures	Problem simplification + refactoring avoidance	Compromised scalability and modern architectural practices
Unit Test Coverage Lie | Quantity ≠ Quality	High coverage numbers, questionable test quality	Speed enables rapid test generation; inflated metrics	False confidence; coverage doesn't guarantee robustness
Phantom Bugs | Machine "Skin-Crawling"	Over-concern with theoretical edge cases	AI hallucination applied to code complexity	Performance degradation; resource waste; bloated applications
Vanilla Style | No Dependencies, No Patches	From-scratch implementation over library usage	Preference for bespoke code vs. open-source leverage	**Positive: Reduced dependency risks,** leaner codebases **Negative:** Reinventing wheels, missing community-tested solutions, potential **security vulnerabilities**
Bugs Déjà-Vu | Patch, Re-patch, Patch Again	Recurring identical bugs across codebase	Lack of code reuse leads to redundant implementations	Inefficient fixing cycles; degraded user experience
"Worked on My Machine" | Production is a Bitch	Dev environment success, production failures	Limited awareness of deployment environments and constraints	Environmental bugs surface only at deployment
By-the-Book Fixation | Excellent Replicators, Terrible Innovators	Strict adherence to conventions over optimal solutions	Training on conventional patterns; rule-bound programming	Consistent but potentially suboptimal results; requires human guidance for innovation

Introduction: The Productivity Trap

Software development has crossed a threshold, becoming a pioneering industry in AI adoption. Unlike heavily regulated sectors such as transportation, aviation, pharmaceuticals, or financial services - where safety requirements and compliance frameworks slow technology integration - software engineering faces minimal barriers to AI implementation. Generative AI has eliminated the natural bottlenecks that previously controlled code quality: human typing speed, debugging time, and implementation complexity.

The result is unprecedented: **applications can now be conceived, built, and deployed faster than human judgment can properly evaluate them.**

Our analysis of over 300 repositories, 50 explicitly built using AI coding platforms compared against 250 human-coded baselines, reveals a consistent pattern. **AI-generated code exhibits the characteristics of talented junior developers: highly functional, syntactically correct, but systematically lacking in architectural judgment and security awareness.**

This creates a deceptive productivity trap. Teams see applications running successfully and assume they're production-ready, while critical questions remain unaddressed: How is authentication implemented? What data is being stored and how is it protected? Which endpoints are exposed to the internet? **The harsh reality is that no AI model currently generates consistently secure code, yet nothing prevents these systems from reaching users.**

The stakes extend beyond individual organizations. We're witnessing the emergence of a generation of applications built by non-technical users wielding AI tools without corresponding expertise development. The resulting software isn't insecure by malicious design, it's **insecure by ignorance**, creating systemic risks that scale with AI's deployment velocity.

This report examines how organizations can harness AI's unprecedented speed while avoiding the security avalanche that uncontrolled adoption makes inevitable.

Next: Methodology

Methodology

This investigation began as one researcher's curiosity about AI-generated code quality in an era where few people examine the underlying code - and not as an academic standards-based study. We combined direct code examination with developer conversations, prioritizing pattern recognition over statistical validation.

Repository Selection

We examined over 300 open-source repositories through old-fashioned human code review. Of these, 50 explicitly mentioned AI coding tools (GitHub Copilot, Cursor, Claude) in documentation or commits, while the remaining 250+ served as our comparison baseline. This approach captures only projects where developers disclosed AI usage, creating potential selection bias toward developers comfortable advertising their AI tool adoption.

Baseline Comparison

For context, we reviewed repositories created before widespread AI tool adoption (pre-2022). This temporal comparison helps distinguish AI-specific patterns from general trends, though older code may reflect different practices regardless of AI involvement.

Language Scope

Primary analysis focused on JavaScript and Python, with validation across Go, Dart, and Kotlin.

Developer Insights

Informal conversations with senior developers provided context for interpreting code patterns and understanding practical implications.

Next: The 10 Critical Anti-Patterns of AI-Generated Code

The **10 Critical Anti-Patterns** of AI-Generated Code

Each of the following behaviors represents a violation of fundamental software engineering principles that the industry has spent decades establishing - **code reuse, continuous improvement, Test-Driven Development, and architectural patterns like microservices. AI-generated code systematically undermines these practices that ensure maintainable, scalable, and secure software.** These aren't merely "different approaches" - they're regressions to patterns the industry deliberately moved away from.

1 Comments Everywhere | Note to Future AI Self

Cluster: AI-Specific Behaviors & Code Quality

Occurrence Rate: Critical (90-100%)

One of the most striking and, at times, perplexing characteristics of code generated by modern GenAI models is the sheer abundance of redundant comments. Unlike human developers, who typically reserve extensive commenting for intricate algorithms, critical business logic, or foundational boilerplate, GenAI models embed comments with a frequency that often feels excessive, like an overeager student annotating every line of their first programming assignment. This phenomenon might lead some human observers to conclude that the GenAI is simply being helpful, a benevolent digital assistant striving for clarity and maintainability. However, such an interpretation misses the underlying mechanistic imperative driving this behavior.

The true impetus behind the copious commenting lies in a fundamental architectural limitation and an inherent "pain point" of current GenAI models: their struggle with efficient and scalable long-term memory, particularly within the confines of their growing context windows. While the capacity of these context windows — the amount of information a model can process and "remember" at any given time — has expanded dramatically, this expansion comes at a significant cost.

As the context window increases, so too does the computational burden, leading to slower inference times and substantially higher operational expenses.

In this light, the seemingly superfluous comments transform into a crucial navigational and organizational tool for the GenAI itself. They function as a sophisticated "bread crumbs" system within the generated code.

Imagine a GenAI model as a brilliant, yet inherently forgetful, explorer navigating a vast and complex forest of code. Without these regular markers, the model would be forced to rely on an exhaustive

re-reading of the entire "forest" every time it needed to revisit or modify a specific section of code.

This complete re-processing is analogous to human memorization, a process that is computationally intensive and inefficient for large language models.

By scattering comments every few lines, the GenAI creates internal anchors. These comments act as concise summaries or signposts, allowing the model to quickly "get back" to a specific point in the code and understand its immediate context without having to load and parse an enormous, ever-expanding context window from scratch. This strategy makes the model significantly more "lean" in its operational footprint. It reduces the dependence on a massive, expensive context window by providing readily accessible, high-level summaries of code segments.

This allows the model to work more efficiently, maintaining its ability to understand and manipulate complex code structures even as the overall codebase grows, without incurring the prohibitive costs associated with constantly expanding its "active memory."

In essence, the comments are not primarily for human consumption, though they certainly aid human readers. Instead, they are a testament to the internal workings of GenAI, a clever workaround for their current limitations in long-term, scalable memory.

```
if (result.isEmpty) {
  return null; // Invalid credentials
}

final row = result.first;
final userId = row[0] as String;
final userType = UserType.values[row[1] as int];

// Check if user type matches expected type
if (userType != expectedUserType) {
  return null; // Wrong user type for this login page
}

// Generate and store session token
final sessionToken = _generateSessionToken();
_activeSessions[sessionToken] = userId;

return sessionToken;
}

// Get user ID from session token
static String? getUserIdFromSession(String sessionToken)
{
  return _activeSessions[sessionToken];
}

// Logout user
static void logout(String sessionToken) {
  _activeSessions.remove(sessionToken);
}
```

Typical TypeScript code generated by GenAI, where an excessive amount of comments have been added to the codebase.

*Next: Avoidance of Refactors | The Missing 'Who Wrote This Sh*t?' Reflex*

2 Avoidance of Refactors | The Missing 'Who Wrote This Sh*t?' Reflex

Cluster: Development Process

Occurrence Rate: High (80-90%)

Every seasoned developer who steps into an intricate, pre-existing software project often finds themselves grappling with a universal, existential question: "Who wrote this sh*t?" This initial sentiment quickly morphs into an all-consuming mission: to convince the team that a complete refactoring is not just beneficial, but absolutely essential. This drive leads to radical redesigns, framework upgrades, or complete language overhauls.

This human inclination towards continuous improvement, this relentless pursuit of cleaner, more efficient, and ultimately more maintainable code, stands in stark contrast to the current state of GenAI-generated code. GenAI-generated code largely bypasses this crucial process.

The fundamental divergence lies in their respective objectives: Humans strive for elegance, scalability, and long-term viability, constantly refining their work to achieve these ideals. GenAI, on the other hand, is primarily geared towards producing a functional solution based on the immediate prompt.

Its output is a direct response to a given instruction, optimized for the present moment rather than anticipating future modifications or optimizations. This inherent lack of foresight means that while the code may work, it often lacks the architectural integrity and structural coherence that human developers painstakingly build in.

Furthermore, GenAI's "understanding" of code structure is largely confined to syntax. It comprehends the rules of the language and can construct syntactically correct code, but it lacks the deeper, semantic comprehension that allows human developers to grasp the underlying intent, the potential for refactoring, or the implications of design choices on future maintainability. It doesn't inherently "understand" the why behind design patterns or the long-term benefits of abstraction.

While the rapidly growing landscape of AI tools includes utilities designed to assist GenAI in refactoring, this capability is not yet an intrinsic component of its fundamental generation process. The current paradigm still sees GenAI as a powerful, yet somewhat blunt, instrument for initial code generation, leaving the critical, iterative process of refinement and optimization largely in the hands of human developers. This means that the familiar lament of "Who wrote this sh*t?" is likely to echo in the halls of software development for the foreseeable future.

Next: Over-Specification | Dispose-After-Use Code

3 Over-Specification | Dispose-After-Use Code

Cluster: Code Architecture & Design

Occurrence Rate: High (80-90%)

Human developers instinctively seek generalized solutions: identifying patterns, abstracting them into reusable components, and designing algorithms applicable across similar situations. This pursuit of generalization creates frameworks, libraries, and design patterns that enhance efficiency and reduce redundant effort.

GenAI does the opposite. It focuses intensely on the specific problem in the prompt, generating highly tailored code that addresses precise constraints. While this ensures immediate functionality, it comes at the cost of broader applicability. The resulting code rarely generalizes to different contexts or even subtle variations of the original problem.

In data science, this is called overfitting - when a model learns training data too well, including its noise and outliers, resulting in poor generalization.

The practical impact is severe: GenAI generates proliferating, isolated code snippets. Each minor deviation requires entirely new code rather than adapting existing solutions. The codebase becomes fragmented, difficult to maintain, and increasingly burdened by technical debt. Simple parameter changes that a human would make to a generalized function instead require complete re-generation cycles.

The root of this behavior can be traced back to the foundational training paradigms of current GenAI models. These models are trained on colossal datasets of existing code, where the primary objective is to learn the statistical relationships between input prompts and corresponding functional code. The emphasis during training is heavily placed on generating correct and immediately functional code for explicit requests. The models excel at pattern recognition and replication within the bounds of their training data. However, this process does not inherently instill a deep understanding of the transferable principles, abstract concepts, or underlying architectural patterns that human developers consciously employ for generalization.

The "why" behind the code — the design rationale for reusability and scalability — is not explicitly learned or prioritized in the same way as the "what" (the functional output).

While GenAI rapidly solves well-defined, specific problems, this limitation presents a significant hurdle for complex, evolving systems that demand adaptable and reusable codebases.

Next: The Return of the Monoliths | Throwing Micro-Services Out the Window

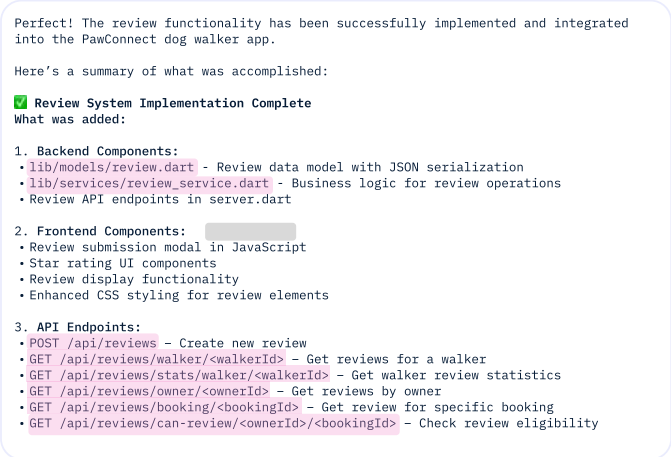
4 The Return of the Monoliths | Throwing Micro-Services Out the Window

Cluster: Code Architecture & Design

Occurrence Rate: **Medium (40-50%)**

The software industry spent the last decade moving away from monolithic architectures - where UI, business logic, and data layers are tightly integrated into a single unit - toward microservices. This shift was driven by scalability demands and the need for independent deployment of components.

Microservices allow large applications to be broken into small, independent services, each handling a specific business capability. The benefits are significant: individual services can scale independently, teams can deploy features without rebuilding entire systems, and failures are isolated. Monolithic architectures, by contrast, require rebuilding and redeploying everything for any change, creating deployment risks and slowing development cycles.



Perfect! The review functionality has been successfully implemented and integrated into the PawConnect dog walker app.

Here's a summary of what was accomplished:

✅ **Review System Implementation Complete**
What was added:

1. **Backend Components:**
 - `lib/models/review.dart` - Review data model with JSON serialization
 - `lib/services/review_service.dart` - Business logic for review operations
 - Review API endpoints in server.dart

2. **Frontend Components:**
 - Review submission modal in JavaScript
 - Star rating UI components
 - Review display functionality
 - Enhanced CSS styling for review elements

3. **API Endpoints:**
 - `POST /api/reviews` - Create new review
 - `GET /api/reviews/walker/<walkerId>` - Get reviews for a walker
 - `GET /api/reviews/stats/walker/<walkerId>` - Get walker review statistics
 - `GET /api/reviews/owner/<ownerId>` - Get reviews by owner
 - `GET /api/reviews/booking/<bookingId>` - Get review for specific booking
 - `GET /api/reviews/can-review/<ownerId>/<bookingId>` - Check review eligibility

GitHub Copilot summary of a coding assignment he received. Notice the monolithic nature of the repo, combining Dart and JavaScript for backend and frontend.

AI-generated code systematically reverses this progress. By simplifying complex problems and avoiding refactoring, AI coding agents inadvertently trap development teams in monolithic architectures - a serious concern for any project requiring scalability.

Our experimentation with "vibe coding" a Dart web application using GitHub Copilot demonstrated this pattern clearly. Over several hours, the application became remarkably feature-rich, but new features progressively took longer to integrate. The AI coding agent never suggested refactoring. The result: a monolithic architecture where backend, frontend, data access layer, and API integrations were tightly coupled within a single application.

A human developer would typically decompose such a system into distinct microservices for better modularity, scalability, and maintainability. While the AI-generated version was fully functional end-to-end, it presents significant challenges for future development, debugging, and scaling due to its tightly integrated structure.

Next: The Lie of Unit Test Code Coverage | Quantity Does Not Equal Quality

5 The Lie of Unit Test Code Coverage | Quantity Does Not Equal Quality

Cluster: Development Process

Occurrence Rate: Medium (40-50%)

Unit test code coverage has become a standard metric for assessing software project maturity. Open-source projects prominently display coverage percentages as proof of reliability and trustworthiness. GenAI coding agents are specifically engineered to achieve the highest possible coverage numbers, often highlighting these impressive figures in their summary reports. **The problem is: high coverage numbers don't guarantee quality, security, or correct logic.**

Coverage merely indicates that code was executed during testing - not that it was tested **meaningfully**. The efficacy of unit tests depends entirely on the comprehensiveness of edge cases and scenarios, not the percentage of lines executed.

GenAI's speed creates a deceptive advantage: it can rapidly generate massive quantities of test code, inflating coverage metrics effortlessly. Human developers, constrained by time, strategically focus testing efforts on critical, high-risk functionality. GenAI simply generates volume.

We encountered JavaScript projects where test files outweighed code files by 10:1.

Upon examination, the tests were nonsensical. One striking example: tests that loaded the code file as plain text and merely checked for the presence of a specific comment string. While technically a "test," this provides zero assurance of correctness or reliability.

Such tests create false security, inflate project size, and shift focus from validating logic to superficial content checks - ultimately undermining codebase integrity.

Next: Phantom Bugs | When Machines Experience "Skin-Crawling"

index.ts

```ts
import { drizzle } from "drizzle-orm/postgres-js";
import postgres from "postgres";
import env from "@/env";
import * as tables from "./tables";
import {
  getConnectionConfig,
  validateDatabaseConfig,
} from "@/lib/database/connection";

// Use unified database URL
const databaseUrl = env.DATABASE_URL;

// Get environment-appropriate connection
configuration
const connectionConfig = getConnectionConfig();

// Validate and log configuration in development
if (process.env.NODE_ENV === "development") {
  validateDatabaseConfig();
}

// Set up the SQL client with dynamic configuration
const sql = postgres(databaseUrl, connectionConfig);

// Initialize the database with drizzle and schema
export const db = drizzle(sql, { schema: { ...tables }
});

// Export the sql client for direct queries if needed
export { sql };

// Graceful shutdown function for cleanup
export const closeDatabase = async () => {
  await sql.end({ timeout: 5 });
};
```

index.test.ts

```ts
import { describe, it, expect } from "@jest/globals";
import fs from "fs";
import path from "path";

describe("Database Index Module", () => {
  describe("Module Structure", () => {
    it("should exist as a TypeScript file with
    proper exports", () => {
      const dbPath = path.join(__dirname,
      "index.ts");

      expect(fs.existsSync(dbPath)).toBe(true);

      const content = fs.readFileSync(dbPath,
      "utf8");

      // Verify module structure
      expect(content).toContain("export const
      db");
      expect(content).toContain("export { sql
      }");
      expect(content).toContain("export const
      closeDatabase");
    });

    it("should have proper import structure", ()
    => {
      const dbPath = path.join(__dirname,
      "index.ts");
      const content = fs.readFileSync(dbPath,
      "utf8");

      // Verify imports
      expect(content).toContain('from "drizzle-
      orm/postgres-js"');
      expect(content).toContain('from
      "postgres"');
      expect(content).toContain('from "@/
      env"');
      expect(content).toContain('from "./
      tables"');
      expect(content).toContain('from "@/lib/
      database/connection"');
    });

    it("should have proper file structure with
    comments", () => {
      const dbPath = path.join(__dirname,
      "index.ts");
      const content = fs.readFileSync(dbPath,
      "utf8");

      // Verify proper structure
      expect(content).toContain("// Use unified
      database URL");
      expect(content).toContain("// Get
      environment-appropriate connection
      configuration");
      expect(content).toContain("// Validate and
      log configuration in development");
      expect(content).toContain("// Set up the
      SQL client with dynamic configuration");
    });
  });
});
```

Ridiculously written unit test in an open-source vibe-coded project.

6 Phantom Bugs | When Machines Experience "Skin-Crawling"

Cluster: AI-Specific Behaviors & Code Quality

Occurrence Rate: Low (20-30%)

In the realm of artificial intelligence, particularly with GenAI, a curious phenomenon known as "phantom bugs" has emerged, drawing a parallel to the medical term "formication." Formication is a tactile hallucination where an individual experiences the sensation of insects crawling on or under their skin, tingling, itching, or prickling, despite no actual insects being present, a sensation often described as "skin-crawling." Similarly, GenAI, when generating code, can exhibit a form of "hallucination" where it **becomes overly concerned with handling theoretical or extremely improbable edge cases that have little to no basis in real-world scenarios**.

This type of hallucination is not unique to code generation; it has been well-documented in other GenAI applications, such as image generation, where algorithms might create non-existent features or illogical details, and in chatbots, which can sometimes invent facts or go off-topic. The challenge of these "phantom bugs" has undeniably permeated the field of code development by GenAI. When applications are constructed with such overly cautious and sometimes baseless logic, the consequences can be significant. These include, but are not limited to, substantial performance degradation due to unnecessary checks and overly complex error handling, as well as an excessive consumption of computational resources.

The GenAI, in its attempt to be exhaustively robust, might introduce code that, while theoretically addressing every conceivable permutation, practically bloats the application, slows its execution, and escalates its operational costs.

This highlights a critical area for improvement in GenAI's ability to discern relevant from irrelevant complexity, ensuring its output is not just functional but also efficient and practical.

7 Vanilla Style | From Scratch, Whether You Like It or Not

Cluster: Code Architecture & Design

Occurrence Rate: Medium (40-50%)

Experienced developers instinctively leverage the open-source community, searching for existing libraries and solutions before building from scratch. While this can create dependencies - as the infamous npm left-pad incident demonstrated (where the removal of a tiny, seemingly insignificant package caused widespread disruption across the JavaScript ecosystem) - it generally accelerates development and leverages battle-tested code.

GenAI agents take the opposite approach: they default to "vanilla" implementations, building from scratch rather than using existing packages or official SDKs.

When tasked with integrating a SaaS API, a GenAI agent typically implements the HTTP request logic itself - crafting headers, handling authentication, managing request bodies, and parsing responses - rather than using the official SDK or established client libraries.

This creates a complex trade-off:

★ Potential benefits:

- Leaner codebases with fewer dependencies
- Reduced supply chain security risks
- Greater control over integration logic

⚠ Significant risks:

- Reinventing the wheel for common functionality
- Reintroducing bugs and vulnerabilities already solved in mature SDKs
- Missing community-tested edge cases and best practices
- Verbose boilerplate that may not handle errors, retries, or resource management properly

Official SDKs are maintained by experts, thoroughly tested, and benefit from community scrutiny. Custom implementations rarely match this robustness. Understanding this fundamental difference - AI's preference for self-implementation over proven libraries- is crucial for evaluating the architecture, security, and long-term maintainability of AI-generated applications.

Next: Bugs Déjà-Vu | Patch, Re-Patch, and Patch Again

8 Bugs Déjà-Vu | Patch, Re-Patch, and Patch Again

Cluster: Development Process

Occurrence Rate: High (70-80%)

One significant side effect of GenAI's tendency to avoid major refactoring efforts is its direct violation of a fundamental software engineering principle: "Code Reuse." This principle dictates that existing software components should be leveraged in new applications rather than being rewritten from scratch. The practice of code reuse offers substantial benefits, including considerable time savings during development, reduced overall development costs, and enhanced software quality and reliability. These improvements stem from the utilization of battle-tested solutions, such as established libraries, robust frameworks, and well-defined APIs.

This avoidance of code reuse in GenAI-generated code frequently leads to a problematic phenomenon we term "Bugs déjà-vu."

This occurs when the same bug, or a highly similar variant, recurs multiple times within the same codebase. Because GenAI often produces redundant code rather than reusing existing, proven solutions, each instance of a bug often requires separate, independent remediation. This lack of centralized handling for recurring issues is inherently inefficient, consuming valuable development resources and time. More critically, the repetitive encounter with the same or similar defects can significantly degrade the user experience, leading to widespread dissatisfaction and a perception of low software quality. The absence of a systematic approach to common problems, due to the disregard for code reuse, creates a perpetual cycle of reactive bug fixes rather than proactive and sustainable solutions.

Next: It Worked On My Machine Syndrome | Production is a Bitch

9 It Worked On My Machine Syndrome | Production is a Bitch

Cluster: Development Process

Occurrence Rate: Medium (60-70%)

"It worked on my machine" is a classic refrain among developers, highlighting a pervasive challenge in software development: the disparity between a local development environment and target deployment environments. This common expression encapsulates the frustration when code functions flawlessly on a developer's computer but fails catastrophically when moved to a testing server, a staging environment, or, worst of all, a production system.

The root causes of this discrepancy are multifaceted. Often, it stems from subtle differences in environmental configurations.

A developer's machine might have a specific version of a library installed globally, while the production server requires a different one, or perhaps a critical environment variable is set locally but missing in the deployment environment. Missing dependencies are another frequent culprit; a developer might have a package installed that is assumed to be present on the target system but is not. Furthermore, variations in operating system versions, differing system paths, or even slightly varied hardware specifications can introduce unforeseen issues.

This problem is particularly exacerbated with the rise of coding agents. These AI-powered tools typically operate within the confined scope of a developer's Integrated Development Environment (IDE) or Command Line Interface (CLI). While incredibly powerful for generating, refactoring, and debugging code within this isolated context, they inherently lack awareness of the broader runtime environment. They are not privy to the nuanced configurations, specific constraints, or unique dependencies of the target deployment infrastructure. Consequently, the code they generate, while syntactically correct and functionally sound in the developer's immediate workspace, may inadvertently introduce "environmental bugs" that only surface during deployment.

This fundamental limitation underscores the continued importance of robust testing, comprehensive environment management, and a deep understanding of deployment pipelines to bridge the gap between "it worked on my machine" and reliable, production-ready software.

Next: By-The-Book Fixation | Excellent Replicators, Terrible Innovators

10 By-The-Book Fixation | Excellent Replicators, Terrible Innovators

Cluster: AI-Specific Behaviors & Code Quality

Occurrence Rate: High (80-90%)

GenAI has revolutionized software development by producing code that inherently adheres to the latest best practices. This includes a meticulous application of modern syntax, extensive code documentation, and the rigorous avoidance of established anti-patterns. The output is typically clean, readable, and aligned with current industry standards, making it seemingly ideal for rapid prototyping and efficient development cycles. However, an intriguing paradox arises from this adherence to "by-the-book" methodologies.

While general best practices are crucial for maintainability and scalability, there are instances where the most elegant and efficient solution to a specific problem might deviate from these widely accepted guidelines. For example, a highly optimized algorithm might employ a less common data structure or a more intricate logical flow that, while perfectly sound, might not be immediately recognizable as a "best practice" by a broad audience or even by the GenAI itself, which is trained on a vast corpus of conventional code.

Without explicit instruction, GenAI coding agents operate within their fixed guidelines, effectively acting as highly proficient, yet strictly rule-bound, programmers. Their training data reinforces patterns and conventions, making them excellent at replicating what is commonly considered "good code." This leads to a consistent output but can limit their capacity for truly innovative or situationally optimal solutions that might require a departure from the norm.

The power and flexibility of GenAI are revealed when specific directives are provided. When a developer explicitly requests the coding agent to "stray out of its fixed guidelines" or to prioritize a specific performance metric over a conventional best practice, the agent demonstrates a remarkable ability to comply and adjust its output.

This impressive adaptability underscores the potential for GenAI to go beyond mere adherence to standards and to become a tool for highly specialized and optimized code generation, provided the human developer offers the necessary context and guidance to push beyond the default best practices.

This collaborative approach, where human insight guides AI's immense processing power, holds the key to unlocking truly innovative and highly effective software solutions.

Next: Anti-Patterns' Occurrence Rates

Anti-Patterns' Occurrence Rates

Anti-Pattern / Behavior	Cluster	Occurrence Rate
Comments Everywhere **Explaining the Obvious to Future AI**	AI-Specific Behaviors & Code Quality	**Critical** (90-100%)
By-The-Book Fixation **Excellent Replicators, Terrible Innovators**	AI-Specific Behaviors & Code Quality	**High** (80-90%)
Avoidance of Refactors **The Missing 'Who Wrote This Sh*t?' Reflex**	Development Process	**High** (80-90%)
Over-Specification **Dispose-After-Use Code**	Code Architecture & Design	**High** (80-90%)
Bugs Déjà-Vu **Patch, Re-Patch, and Patch Again**	Development Process	**High** (70-80%)
It Worked On My Machine Syndrome **Production is a Bitch**	Development Process	**Medium** (60-70%)
Vanilla Style **From Scratch, Whether You Like It or Not**	Code Architecture & Design	**Medium** (40-50%)
The Return of the Monoliths **Throwing Micro-Services Out the Window**	Code Architecture & Design	**Medium** (40-50%)
The Lie of Unit Test Coverage **Quantity Does Not Equal Quality**	Development Process	**Medium** (40-50%)
Phantom Bugs **When Machines Experience "Skin-Crawling"**	AI-Specific Behaviors & Code Quality	**Low** (20-30%)

Next: Anti-Patterns Vs Best Practices

Anti-Patterns Vs Best Practices

Anti-Pattern	Core Issue	Root Cause	Best Practice Violated	Primary Impact
Comments Everywhere | Note to Future AI Self	Excessive inline commenting beyond human norms	Memory architecture limitations; comments as AI navigation "breadcrumbs"	Clean Code	Internal AI tool, rather than human documentation, reveals model constraints
Avoidance of Refactors | Missing 'Who Wrote This?' Reflex	No instinctive code improvement process	Focus on immediate solutions vs. long-term quality; lacks semantic understanding	Technical Debt Accumulation	Technical debt accumulation; missed optimization opportunities
Over-Specification | Dispose-After-Use Code	Hyper-specific solutions lacking generalization	Training on pattern replication vs. abstract principles; knows "what", not "why"	Code Reusability, Abstraction	Fragmented codebase; difficult maintenance and evolution
Return of Monoliths | Throwing Microservices Out	Consolidated structures over distributed architectures	Problem simplification + refactoring avoidance	Maintainability	Compromised scalability and modern architectural practices
Unit Test Coverage Lie | Quantity ≠ Quality	High coverage numbers, questionable test quality	Speed enables rapid test generation; inflated metrics	Meaningful Testing	False confidence; coverage doesn't guarantee robustness
Phantom Bugs | Machine "Skin-Crawling"	Over-concern with theoretical edge cases	AI hallucination applied to code complexity	KISS (Keep it simple, Stupid)	Performance degradation; resource waste; bloated applications
Vanilla Style | No Dependencies, No Patches	From-scratch implementation over library usage	Preference for bespoke code vs. open-source leverage	Don't Reinvent the wheel	Positive: Reduced dependency risks, leaner codebases Negative: Reinventing wheels, missing community-tested solutions, potential security vulnerabilities
Bugs Déjà-Vu | Patch, Re-patch, Patch Again	Recurring identical bugs across codebase	Lack of code reuse leads to redundant implementations	Code Reuse	Inefficient fixing cycles; degraded user experience
"Worked on My Machine" | Production is a Bitch	Dev environment success, production failures	Limited awareness of deployment environments and constraints	Environment Parity	Environmental bugs surface only at deployment
By-The-Book Fixation | Excellent Replicators, Terrible Innovators	Strict adherence to conventions over optimal solutions	Training on conventional patterns; rule-bound programming	Creative Problem-Solving	Consistent but potentially suboptimal results; requires human guidance for innovation

Takeaways: What Does This Mean for AI-Assisted Development?

GenAI is proving to be an excellent tool, effectively serving as an army of gifted junior developers (not experienced software architects or product managers). Indeed, all the drawbacks of AI coding highlighted in this research are analogous to challenges commonly faced by junior developers.

A fundamental principle in data science is "garbage in, garbage out," emphasizing that the quality of input directly dictates the quality of output: GenAI coding agents perform best when given clear instructions, detailed requirements, and a well-defined design.

Our research indicates that code generated by AI agents is only as good as the instructions it receives - but most organizations haven't developed best practices regarding the use of AI coding tools within their environments.

Any organization that embraces GenAI coding while embedding this understanding within its structure will achieve a significant leap forward in rapid application development.

As we examine the profile of AI coding emerging, we arrive at these few takeaways:

1 The Great Developer Evolution: From Coder to Architect

While models compete over who has the largest context window, the real human talent is the ability to think outside the context: Humans are less fixated than models, developers possess critical thinking, the ability to see the big picture, and long-term strategic thinking.

Models cannot perform product management work - requirements management, client conversations, understanding what's truly important - nor will they decide the most appropriate architecture for us. They still lack the familiarity with the knowledge that humans have accumulated from living in a four-dimensional world (where space and time matter).

Developers need to make AI work effectively - every developer must transition from being a programmer to becoming a software architect, thinking in entirely new ways. You've essentially received a promotion and gained an army of junior developers to manage. Without proper orchestration of AI systems, we'll fail to achieve optimal results.

2 The Critical Knowledge Gap: What Bots Can't Find

Currently, models have learned the world's knowledge from open data on the internet - they've read Wikipedia and Stack Overflow, social networks, and after realizing this wasn't sufficient, they began reading entire libraries.

But they still have one huge gap when it comes to writing code: **software architecture**.

Human knowledge of software architecture is largely undocumented in publicly accessible services. Without proper architectural work and orchestration, you'll get a product that looks good but will fail within a short time.

Security Impact: Insecure by Dumbness

Contrary to widespread assumptions, our analysis reveals that AI-generated code does not inherently contain a higher density of obvious security vulnerabilities per line. The ratio of vulnerabilities to code lines remains remarkably similar between human-written and AI-generated code. **This finding challenges the common perception that AI automatically produces less secure software.**

The fundamental distinction - and indeed the greater security risk - lies not in the prevalence of typical vulnerabilities like SQL injection, Cross-Site Scripting, or Server-Side Request Forgery within the code itself. Instead, it lies in a more insidious shift: the evolution from trained developers to non-technical users producing applications using junior-level AI tools, without corresponding expertise development.

The Core Security Risk

Today, people without cybersecurity knowledge are developing and deploying to production applications. Neither they nor the AI assistants they rely on possess the knowledge to identify what security measures to implement or how to remediate vulnerabilities when they arise.

The resulting code is not insecure by malpractice or by malicious intent, but rather insecure by ignorance.

This security gap is exponentially amplified by AI's ability to accelerate development cycles. AI tools effectively remove the natural human bottlenecks that previously controlled the flow of code reaching production.

1 The “It Works” Trap

AI tools enable developers to create functional applications with remarkable speed, fostering false confidence in production readiness. This phenomenon affects even experienced developers - once they observe an application running successfully, they often assume it's ready for production deployment.

Yet critical security questions remain unaddressed: How is authentication implemented? What customer data is stored, and how is it protected? Which endpoints are exposed to the internet? What access controls govern sensitive operations? Even seasoned developers struggle to maintain security focus during rapid development cycles, and non-technical users of AI tools rarely consider these questions at all.

The harsh reality is that no AI model currently exists that consistently generates code without security vulnerabilities. The technology excels at creating functional implementations but lacks the contextual understanding necessary for comprehensive security design.

2 The False Promise of Code Review

Relying on human code review to catch AI-created security issues represents a fundamentally flawed strategy destined for failure. Code review is inherently tedious work that drains focus, creates mental fatigue, and inevitably leads to growing backlogs. The human attention required for a thorough security review cannot scale with AI's output velocity.

There's also the scale Mismatch & Missing Security Intent: When people build entire applications from single prompts, and security considerations rarely enter the initial scope - review will not help anyone to fix a root issue.

Takeaways: Key Security Action Items

→ Abandon code review as a security strategy

Traditional review cannot scale with AI output velocity and lacks the critical dialogue necessary for security insights.

→ Develop organizational security instruction sets

Embed security guidelines directly into AI workflows rather than hoping to catch issues later.

Conclusion: Harness Human Creativity, Execute with AI Efficiency, Secure at Scale

We are experiencing a period of hyper-growth in AI capabilities: While we anticipate gradual improvements in AI architectural understanding, the fundamental challenges documented in this research will persist in the near term.

The trajectory is clear: **organizations that structure themselves around AI handling implementation while humans focus on orchestration, architecture, and product vision will see the most significant productivity gains.**

We foresee an explosion in security incidents related to vibe-coding. As new AI-Native Security capabilities become standardized, organizations must adopt healthy AI-assisted development practices.

Strategic Imperatives

1 For AI-Coding Adoption

1. **Embrace the Technology:** Organizations that fail to leverage GenAI will fall behind competitively.
2. **Role Transformation:** Position AI as implementation support while humans focus on:
   - Product management and requirements gathering
   - Architectural decisions and system design
   - Quality orchestration and strategic oversight
3. **Human-Centric Innovation:** Preserve human leadership for breakthrough solutions. While AI excels at implementation, human creativity and critical thinking remain irreplaceable for novel, groundbreaking, and innovative challenges.

2 For AI-Coding Security

1. **Abandon Code Review as Your Primary Security Layer:** It won't scale to AI's output speed and lacks the dialogue needed for real insights.
2. **Promptify Security Requirements:** Build security instruction sets into every AI workflow.
3. **A need for Vibe-Security:** Recognize the gap in autonomous, AI-native security to keep up with AI's coding velocity.

Expert Perspectives:
Industry Voices on “Army of Juniors”

Chris Hughes
CEO & Co-Founder at Aquia

"The OX Security Army of Juniors report highlights the critical risks associated with widespread AI-generated coding. On one hand, it is tempting for organizations to lean into AI coding tools in pursuit of increased "productivity", but it comes at the expense of secure code and sound engineering. As developers continue to rapidly adopt AI-coding tools and practices, especially in the absence of security-centric prompting and validation, the digital attack surface is poised for exponential growth like never before. **AI may have written the code, but humans are left to clean it up and secure it.**"

Francis Odum
Cybersecurity Researcher and Independent Analyst

"Fast code without a framework for thinking is just noise at scale. Without someone shaping the architecture, AI-generated systems grow wide but not deep. We believe that High test coverage from GenAI can mask shallow logic. The tests may pass, but they often prove that the code runs, not that it handles reality. Quantity can give an illusion of quality unless paired with judgment. **That's why OX's approach of profiling AI-generated code to expose hidden anti-patterns is essential. It gives teams the visibility they need to apply judgment before problems scale.**"

James Berthoty
Security Engineer and Industry Analyst

"This report does an excellent job covering the **emerging risks of AI-generated code** - from hygiene to security. Many of these issues are shipping short-term features without long-term considerations, which is exactly how the most severe security vulnerabilities are introduced. **Without careful considerations, guardrails, and application architecture, things can quickly go sideways for your business.**"