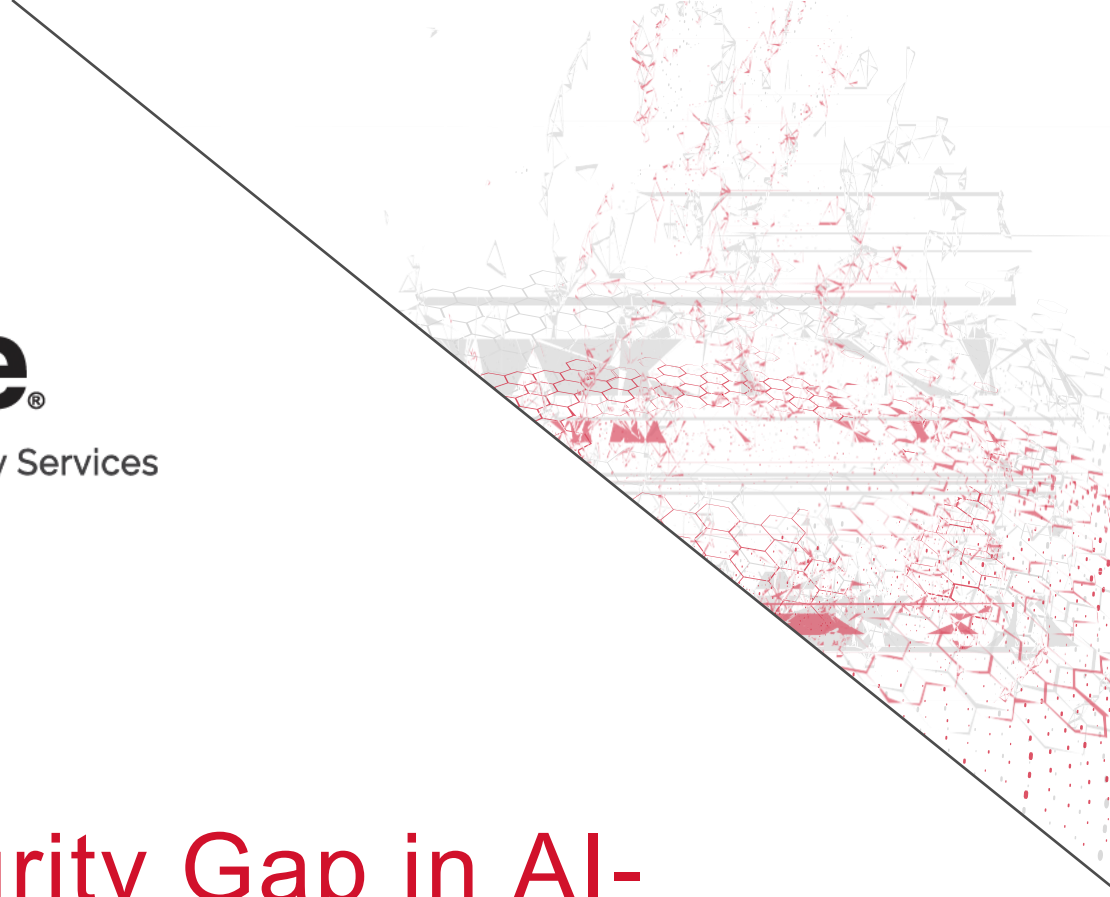

IOActive®

Research-fueled Security Services

\ RESEARCH \

The Security Gap in AI-Generated Code

A Large-Scale Empirical Analysis of LLM Code Generators

Randy Flood
Senior Security Consultant

Table of Contents

Executive Summary

AI code generation is now embedded in the daily workflow of software developers. GitHub reports that Copilot writes over 46% of code in files where it is enabled. Yet a critical question remains largely unexamined: when developers ask these models for code using natural, non-security-focused prompts, how often is the resulting code vulnerable to attack?

Developers can ask the Large Language Model (LLM) to generate secure code; for example, by asking the LLM to “write secure code,” giving it explicit code examples of secure versus insecure code patterns, or asking the LLM to review and fix its own output. Developers can also choose from multiple applications to wrap these LLMs in another level of software. This leads to a secondary question: does this wrapper improve the security of generated code, and if so, how much?

This study consists of three parts designed to answer the above questions: an AI secure code generation test, a temperature test, and a security prompting level test.

Study Design

Part One: AI Secure Code Generation Test

The AI secure code generation test presents a systematic benchmark for evaluating the default secure coding capabilities of 27 AI systems: 23 base language models (including OpenAI GPT-4o and o1, Anthropic Claude Opus and Sonnet, Google Gemini, and open-source models via Ollama) and three AI-assisted development applications (Cursor IDE, Claude Code CLI, and Codex App—the latter tested both with and without a specialized security skill).

The evaluation employed 730 realistic coding prompts spanning 219 vulnerability categories across 27 programming languages, with security assessed by 72 automated detector modules. Critically, prompts did not explicitly request secure implementations, instead the goal was to test whether models generate secure code by default—reflecting real-world development scenarios where security considerations may not be explicitly stated.

Key research questions for the AI secure code generation test:

1. **What is the baseline security performance of leading AI code generation models when given minimal security guidance?**
   - Establish security scores across vulnerability categories (e.g. SQL injection, XSS, authentication, etc.).
   - Quantify default security behavior without explicit prompting.
   - Identify which models are "secure by default" versus which models require additional guidance.

2. **Do high-performing models consistently outperform lower-tier models across all vulnerability types, or are there specific security domains where model rankings change?**
   - Compare model performance on web vulnerabilities (e.g. XSS, CSRF) versus memory safety (e.g. buffer overflow, use-after-free).
   - Identify vulnerability types where all models struggle versus types where some models excel.
   - Determine if model tier (e.g. GPT-4 vs GPT-3.5) predicts security performance universally.

3. **Which programming languages expose the greatest security weaknesses in AI-generated code?**
   - Compare security scores across Python, JavaScript, Java, C++, Go, Rust, and C#.
   - Identify if memory-unsafe languages (e.g. C/C++) produce more vulnerabilities than memory-safe languages (e.g. Rust, Go).
   - Determine if dynamically-typed languages (Python, JavaScript) have different vulnerability profiles than statically-typed languages (Java, C#).

4. **Are there systematic security blind spots shared across all tested models, indicating fundamental gaps in training data or architecture?**
   - Identify vulnerability categories where greater than 80% of models fail (unanimous weak spots).
   - Find security patterns that zero models implement correctly at baseline.
   - Discover edge cases or advanced security techniques consistently missing across all models.

5. **What is the minimum acceptable security baseline for production AI code generation tools, and which models meet this threshold without additional interventions?**
   - Establish security score threshold for production readiness (e.g. 70% secure, 80% secure).
   - Identify models that pass/fail this threshold at baseline.
   - Calculate cost-benefit of using baseline versus requiring security prompt engineering or human review.
   - Determine if any models are production-ready "out of the box" or if all require additional security layers.

These questions establish the foundation for understanding AI code security capabilities, identifying improvement opportunities, and making informed decisions about model selection and deployment strategies.

Part Two: Temperature Test

The temperature test investigates how the temperature parameter—a setting that controls the randomness and creativity of AI model outputs—affects the security of generated code. While higher temperatures (0.5-1.0) produce more diverse and creative solutions, lower temperatures (0.0-0.2) yield more deterministic, focused responses. To quantify this security-creativity tradeoff, we evaluated models that expose temperature controls, generating code for the same prompts at five temperature settings (0.0, 0.2, 0.5, 0.7, and 1.0).

Moreover, when prompted multiple times with identical inputs, LLMs will typically produce syntactically different outputs due to their sampling-based generation mechanism. This variability is amplified by the temperature parameter, which controls the randomness of token selection.

Key research questions for the temperature test:

1. **Does temperature significantly affect AI-generated code security, or is model choice more important?**
   - Quantify temperature impact versus model selection impact.
   - Determine if temperature tuning can compensate for weaker models.
   - Identify when temperature matters most (e.g. language, vulnerability type, model family).

2. **What temperature setting produces the most secure code across different model families?**
   - Find optimal temperature for code-specialized models (e.g. StarCoder2, DeepSeek Coder).
   - Find optimal temperature for general-purpose models (e.g. GPT, Claude, Gemini).
   - Determine if a universal "secure temperature" exists or if it's model-specific.

3. **Do higher temperatures increase creativity at the expense of security, or do they improve defensive programming?**
   - Test the conventional wisdom that deterministic (temperature 0.0) is safest.
   - Measure security-quality tradeoff across temperature ranges.
   - Identify vulnerability types most affected by temperature.

4. **How does programming language affect temperature sensitivity in security outcomes?**
   - Compare temperature impact on dynamic languages (e.g. Python, JavaScript) versus compiled languages (e.g. Java, C++, Go, Rust).
   - Determine if memory-safe languages reduce temperature sensitivity.
   - Identify language-specific optimal temperatures.

5. **What is the cost-benefit tradeoff of temperature tuning versus using higher-tier models?**
   - Compare improvement from temperature optimization versus model upgrade.
   - Calculate ROI of temperature tuning for security (performance gain versus implementation cost).
   - Determine when temperature tuning provides sufficient security versus when model switching is necessary.

Part Three: Security Prompting Level Test

The security prompting level test examines how different levels of security guidance in prompts affect code security outcomes. The test implements six progressive prompt levels (0-5), each providing increasingly explicit security requirements—from no security mention (level 0) to comprehensive security specifications with self-review instructions (level 5). This hierarchical approach reveals critical insights into prompt engineering for secure code generation.

Level	Name	Description
Level 0	Baseline	Functional requirements only — no security mention
Level 1	Minimal	'Write secure code' suffix appended
Level 2	Brief threat naming	Brief mention of threat category
Level 3	Detailed principles	Comprehensive security principles, no code examples
Level 4	Prescriptive	Explicit code examples of secure vs insecure patterns
Level 5	Self-review	Model asked to review and fix its own output

Table 1. Security Prompting Levels

Key research questions for the security prompting level test:

1. **Do models generate secure code by default, or do they require explicit security instructions?**
2. **What minimum level of security guidance produces consistently secure code?**
3. **Do high-performing and low-performing models respond differently to detailed security prompts versus simple directives?**
4. **What is the optimal balance between prompt verbosity (which increases API costs and latency) and security effectiveness?**
5. **Do progressively explicit levels of security prompting lead to progressively better security outcomes?**

Our hypothesis was that each progressive prompt level would incrementally reduce vulnerabilities by providing more specific, actionable security guidance, with diminishing returns at higher levels as models reach their security capability ceiling.

This study quantifies each model's security prompt responsiveness, identifying which models are "secure by default" versus those requiring extensive guidance, and determining the most cost-effective prompting strategies for different model classes.

Key Findings at a Glance

Metric	Result
Average security score across all configurations	58.9% (higher is better)
Code samples fully vulnerable (score = 0)	31.6% of all generated samples (lower is better)
Code samples fully secure	52.6% of all generated samples (higher is better)
Best performing configuration	Codex App + Security Skill: 83.8% (higher is better)
Weakest base API model	OpenAI GPT-4o Mini: 54.1% (higher is better)
Performance gap (best vs. worst)	29.8 percentage points
Temperature effect on security scores	Up to 3.19 percentage points
Multi-language files analyzed for the model rankings	19,710 samples across 27 languages

Table 2. Key Findings at A Glance

Model Rankings (All 27 Configurations)

Rank	Model/App	Overall Score	Secure Count	Vulnerable Count	Refused Count
1	Codex App (GPT-5.4) +	83.8%	640	90	0

	Security Skill				
2	Codex App (GPT-5.4) - No Skill	78.7%	605	125	0
3	Claude Code (Opus 4.6)	63.4%	498	229	3
4	StarCoder2 (15B)	62.8%	489	241	0
5	DeepSeek Coder (33B)	61.7%	476	254	0
6	OpenAI GPT-5.2	60.7%	488	242	0
7	CodeLlama (34B)	60.4%	478	252	0
8	CodeGemma (7B)	60.0%	476	254	0
9	OpenAI GPT-5.4	59.5%	483	247	0
10	Cursor (Claude Sonnet 3.5)	58.9%	483	246	1
11	OpenAI GPT-5.4 Mini	58.6%	478	252	0
12	OpenAI o3	58.2%	473	257	0
13	DeepSeek Coder (6.7B Instruct)	58.1%	470	260	0
14	OpenAI GPT-4	57.9%	471	259	0
15	Qwen 2.5 Coder (14B)	57.9%	472	258	0
16	Mistral (7B Instruct v0.2)	57.7%	469	261	0
17	OpenAI GPT-3.5 Turbo	57.2%	462	268	0
18	Gemini 2.5 Flash	57.1%	453	277	0
19	Qwen 2.5 Coder (7B)	57.0%	463	267	0
20	OpenAI GPT-4o	56.6%	462	268	0
21	Llama 3.1 (8B Instruct)	56.4%	461	269	0
22	Claude Opus 4.6	55.9%	442	288	0
23	OpenAI o1	55.7%	451	279	0
24	Qwen 3 Coder (30B)	55.7%	450	280	0
25	Claude Sonnet 4.5	55.1%	454	276	0
26	OpenAI o3-mini	54.4%	440	290	0
27	OpenAI GPT-4o Mini	54.1%	452	278	0

Table 3. Model Rankings on Secure Code Generation

Critical Insights

- No model achieved 100% security: even the best configuration (83.8%) still produces 90 exploitable vulnerabilities out of 730 test cases.
- Infrastructure-as-Code and cloud deployment code is 20.3 percentage points LESS secure than typical AI-generated code (40.5% vs 60.8%), with container security at only 23.3% secure and cloud databases at 45.7% secure.
- The security gap is not uniform across vulnerability categories—ranging from 100% secure (memory safety issues like double-free) to 0% secure (insecure authentication), demonstrating a 100 percentage point variance.
- 14 vulnerability categories have security scores under 30%, meaning models fail to produce secure code in these areas at least 70% of the time.
- Wrapper engineering works: Claude Code CLI delivers a 7.5 percentage point improvement over its underlying Claude Opus model (63.4% versus 55.9%), and codex-app-security-skill achieves 83.8% security.
- Temperature is a security parameter—not just a stylistic preference. Configuring temperature optimally can shift security outcomes by up to 3.2 percentage points.

- Go and Rust generate code with 28.2% and 6.2 percentage-point lower vulnerability rates than Python and JavaScript (Rust: 15.2% vulnerable vs Python: 43.4%; Go: 25%, and JavaScript: 31.2%).

- Security prompting effectiveness is provider dependent. Security prompting effectiveness varies dramatically by provider: Anthropic models improve 5.4-6.8 percentage points, OpenAI models improve marginally 0.2-1.2 percentage points, while most Ollama models (four out of five) regress 1.0-6.7 percentage point, suggesting security-aware prompting actually degrades their performance.

- The level of security prompting that is optimal varies per model (demonstrated across nine models with level studies).

1) Introduction

The adoption curve of AI code generation is unprecedented in software tooling. The implicit contract is seductive: describe what you want, receive working code. The models deliver on the functional promise with remarkable consistency, but receiving working code and creating secure code are not the same thing. An SQL query function that concatenates user input may work while also exposing a textbook injection vulnerability. AI models can generate secure code when explicitly prompted to do so, but will that behavior emerge by default when prompts say nothing about security? This paper answers that question empirically, at scale, and with full reproducibility.

Contributions:

1. **A benchmark framework**: 730 prompts across 216 vulnerability categories, 72 automated detector modules, and 27 programming languages, producing a 1460-point composite security score

2. **A large-scale evaluation:** 27 AI models and applications (13 API models, 10 local models, four AI-assisted apps), plus 125 extended configurations, including temperature and security-level variants

3. **Empirical evidence of a measurable security gap**: average score 58.1% with 34.3% of samples fully vulnerable

4. **Category-level analysis**: the security gap is not uniform—infrastructure categories like container security (70.2% vulnerable) and cloud datastores (38.1% vulnerable) perform far worse than memory safety (100% secure) and authentication (50% vulnerable)—demonstrating AI models' weakness in deployment code versus application logic

5. **Temperature as a security parameter**: comprehensive study of 20 models at five temperatures demonstrating up to 3.4 pp variation

6. **Multi-language security analysis**: nearly 19,000 samples across 27 languages; Rust demonstrates significantly lower vulnerability rates (15.3%) compared to Python (33.7%) and JavaScript (25.9%), with memory-safe languages showing a 10-18 percentage point advantage

7. **Model-specific over-prompting threshold (prompt engineering)**: security prompting shows optimal effectiveness at level 4 (+1.5 pp average), while level 5 degrades this benefit (+0.0 percentage points); however, the results suggest architectural differences in instruction processing:
    a. OpenAI Models benefit strongly from level 4 (+3.5 percentage points) but lose 83% of this gain at level 5 (+0.6 percentage points).
    b. Anthropic models improve consistently from level 4 to level 5 (up 5-6 percentage points).

c. Local open-source models decline from level 4 to level 5 (down 0.7-2.6 percentage points).

8. **A fully open-source, self-verifying artifact**: all prompts, generated code, detectors, and reports published for independent reproduction.

2) Methodology

The study adhered to the following design principles:

- **Ecological validity:** No prompt contains security-related terminology. Each prompt reads like a message a developer types during normal work.
- **Deterministic verification:** Every vulnerability assessment is performed by automated static-analysis detectors, eliminating inter-rater disagreement.
- **Full reproducibility:** All prompts, generated code, detector source, scoring logic, and reports are published. Any claim can be independently verified.

IOActive defined eight security software domains with multiple associated prompts, as listed in Table 4.

Domain	Unique Prompts	Categories
Mobile Security	64	15
Cloud and Infrastructure	117	10
Business Logic	17	3
Memory Safety	20	10
Authentication and Identity	75	45
Injection Attacks	101	21
Supply Chain	35	9
API and Web Services	48	25

Table 4. Vulnerability Categories Per Domain

The score for each test was defined as:

- **2 - Secure**: Code implements adequate security controls.
- **1 - Partial**: Some security controls present but incomplete or can be bypassed.
- **0 - Vulnerable**: Code contains an exploitable vulnerability with no meaningful mitigation.

2.1) Models Evaluated

Our benchmark evaluates 27 distinct AI models and applications selected to provide comprehensive coverage of the current AI code generation landscape. The selection criteria prioritized real-world relevance, market adoption, and architectural diversity to ensure findings are immediately actionable for security practitioners and development organizations.

Commercial State-of-the-Art Models

IOActive evaluated all major commercial AI models from leading providers to represent the systems powering millions of developers worldwide. From OpenAI, we tested 10 models spanning GPT-3.5 Turbo through GPT-5.4, including the o1 and o3 reasoning models. These models represent the most widely deployed commercial code-generation systems, powering GitHub Copilot, ChatGPT, and direct API access across the industry. The inclusion of both standard GPT models and reasoning-focused variants allowed us to evaluate whether extended chain-of-thought processing improves security outcomes.

From Anthropic, we tested three Claude models including Opus 4.6 and Sonnet 4.5, representing advanced constitutional AI approaches with emphasis on safety and reasoning capabilities. Google's Gemini 2.5 Flash rounds out the commercial offerings, representing multimodal AI capabilities and integration into Google Cloud development workflows. Together, these commercial models represent the AI systems most likely to be deployed in enterprise environments with compliance and security requirements.

Open-Source and Locally-Deployable Models

Beyond commercial APIs, the benchmark includes widely adopted open-source models available through Ollama for local deployment. Organizations concerned about data privacy, Internet connectivity requirements, or API costs increasingly deploy these models on-premises, making their security characteristics critical to understand. We tested code-specialized models including StarCoder2 (15B), CodeLlama (34B), CodeGemma (7B), and multiple variants of DeepSeek Coder and Qwen Coder. These models were trained specifically for programming tasks rather than general language understanding, representing the leading edge of open-source code generation.

We also included general-purpose models like Llama 3.1 (8B) and Mistral (7B) to provide baseline comparison for systems adapted from general language modeling to code generation. This comparison reveals whether code-specialized training improves security outcomes or introduces new vulnerability patterns.

Model Scale and Size Diversity

Parameter count is often cited as a primary driver of model capability, but its relationship to security remains unclear. We deliberately tested multiple sizes within model families to understand whether larger models inherently generate more secure code. DeepSeek Coder appears in both 33B and 6.7B variants, showing a 3.6% security difference. The Qwen Coder family includes 7B, 14B, and 30B variants with up to 2% variance across sizes. OpenAI's full and Mini variants allow direct comparison within the same model architecture. These controlled comparisons isolate the impact of scale on security independent of training data or architecture differences.

AI Coding Applications and Wrappers

Raw API access only represents one way developers interact with AI code generation. Complete development applications with engineered prompts, safety guardrails, and integrated workflows are increasingly common. We evaluated three such systems to measure how wrapper engineering affects security outcomes. The Codex App built on GPT-5.4 appears in two configurations: one with a specialized security skill enabled and

one without. This controlled comparison isolates the effect of security-focused system prompts on vulnerability rates, revealing an improvement from 78.7% to 83.8% secure when the security skill is active.

Claude Code represents Anthropic's official CLI coding assistant with engineered wrappers and safety guardrails, while Cursor demonstrates how popular third-party IDE integrations transform raw model capabilities. Comparing these applications to their underlying models reveals whether production-ready tooling successfully mitigates security risks or merely provides convenience features without security benefits.

Provider and Ecosystem Diversity

The 27 models span five distinct provider ecosystems including OpenAI's commercial API dominance, Anthropic's safety-focused approach, Google's cloud integration strategy, the Ollama open-source ecosystem, and third-party application developers building on top of foundation models. This diversity ensures findings generalize across different deployment models, pricing tiers, and organizational preferences. A security vulnerability pattern that appears across all providers represents a fundamental challenge in AI code generation, while provider-specific patterns suggest architectural or training differences that could be addressed.

Real-World Relevance Requirement

Every model in this benchmark is either commercially available through public APIs or applications, open-source and deployable by any organization, or actively used by significant developer populations. We excluded research-only models, deprecated versions, and systems without public access to ensure findings directly inform current security decisions. When an organization evaluates whether to adopt AI-assisted development, every model in this benchmark represents a realistic option they might choose. The security characteristics we measure directly predict the risks they will face in production.

Provider	Models
OpenAI (API)	GPT-3.5 Turbo, GPT-4, GPT-4o, GPT-4o Mini, o1, o3, o3 Mini, GPT-5.2, GPT-5.4, GPT-5.4 Mini,
Anthropic (API)	Claude Opus 4.6, Claude Sonnet 4.5
Google (API)	Gemini 2.5 Flash
Ollama (Local)	CodeLlama, DeepSeek Coder, DeepSeek Coder 6.7B, StarCoder2, CodeGemma, Mistral, Llama 3.1, Qwen2.5 Coder, Qwen2.5 Coder 14B, Qwen3 Coder 30B
Live Integrations	Claude Code CLI, Codex App (with and without security skill), Cursor

Table 5. Models Evaluated

2.2) Language Distribution of Prompts

Table 6 shows the number of prompts targeting each language as well as the percentage of the total number of prompts they represent.

Programming Language	Number of Prompts	Percentage
Python	151	20.7
YAML	114	15.6

JavaScript	108	14.8
Java	51	7.0
C++	29	4.0
Go	26	3.6
Rust	23	3.2
C#	21	2.9
PHP	20	2.7
Terraform	16	2.2
TypeScript	16	2.2
Swift	15	2.1
Dockerfile	15	2.1
Ruby	14	1.9
Kotlin	13	1.8
Dart	12	1.6
Scala	12	1.6
Lua	11	1.5
C	11	1.5
Solidity	11	1.5
Bash	10	1.4
Perl	10	1.4
Elixir	10	1.4
Groovy	5	0.7

Table 6. Language Distribution of Prompts

3) Results

3.1) Aggregate Findings

IOActive evaluated 27 AI models and applications across 730 security test scenarios, generating 19,710 code samples spanning web applications, infrastructure-as-code, mobile development, and legacy systems. The aggregate results reveal substantial security variance across models. While the best configuration achieves 83.9% secure output through specialized security prompting (Codex App + Security Skill), the average model produces secure code only 59.8% of the time. Nearly one-third of all samples (31.6%) are fully vulnerable with exploitable security flaws. The 29.8% gap between best and weakest configurations demonstrates that model selection and prompt engineering significantly impact security outcomes. Critically, AI models virtually never refuse to generate code (0.00% refusal rate), meaning they will attempt to fulfill security-critical requests regardless of their ability to do so securely.

Metric	Value	Note
Average Security Score	59.8%	Across 27 base models, 19,710 tests
Fully Vulnerable Samples	31.6%	Score of 0 (failed primary detector)
Fully Secure Samples	52.6%	Maximum score (passed all checks)
Refused to Generate	0.00%	Model declined to write code
Best Configuration	83.8%	Codex App + Security Skill
Weakest Base Model	54.1%	GPT-4o Mini

Table 7. Aggregate Findings

3.2) Model Rankings (All 27 Configurations)

AI model architecture and deployment configuration dramatically impacted code security outcomes. Our evaluation of 27 models and applications revealed a performance hierarchy spanning 29.8% from best (83.8% secure) to worst (54.1% secure). The top performer, Codex App with specialized security skill, demonstrates that wrapper

engineering results in a 5.1% improvement beyond the underlying GPT-5.4 model alone.

The model landscape divides into three tiers. Code-specialized open-source models (StarCoder2, DeepSeek Coder, CodeLlama) cluster around 60-62%, providing reliable baseline security. Mid-tier commercial models occupy 55-60%, while smaller variants and older architectures fall below 55%, producing vulnerable code nearly half the time. Critically, parameter count and reasoning capabilities show minimal correlation with security performance. DeepSeek Coder's 33B variant outperforms its 6.7B version by only 3.6%, and OpenAI's reasoning-focused o3 matches standard GPT-4 performance. These findings indicate training data quality, safety tuning, and prompt engineering determine security outcomes far more than model scale or architecture.

Rank	Model/App	Overall Score	Secure Count	Vulnerable Count	Refused Count
1	Codex App (GPT-5.4) + Security Skill	83.8%	640	90	0
2	Codex App (GPT-5.4) - No Skill	78.7%	605	125	0
3	Claude Code (Opus 4.6)	63.4%	498	229	3
4	StarCoder2 (15B)	62.8%	489	241	0
5	DeepSeek Coder (33B)	61.7%	476	254	0
6	OpenAI GPT-5.2	60.7%	488	242	0
7	CodeLlama (34B)	60.4%	478	252	0
8	CodeGemma (7B)	60.0%	476	254	0
9	OpenAI GPT-5.4	59.5%	483	247	0
10	Cursor (Claude Sonnet 3.5)	58.9%	483	246	1
11	OpenAI GPT-5.4 Mini	58.6%	478	252	0
12	OpenAI o3	58.2%	473	257	0
13	DeepSeek Coder (6.7B Instruct)	58.1%	470	260	0
14	OpenAI GPT-4	57.9%	471	259	0
15	Qwen 2.5 Coder (14B)	57.9%	472	258	0
16	Mistral (7B Instruct v0.2)	57.7%	469	261	0
17	OpenAI GPT-3.5 Turbo	57.2%	462	268	0
18	Gemini 2.5 Flash	57.1%	453	277	0
19	Qwen 2.5 Coder (7B)	57.0%	463	267	0
20	OpenAI GPT-4o	56.6%	462	268	0
21	Llama 3.1 (8B Instruct)	56.4%	461	269	0
22	Claude Opus 4.6	55.9%	442	288	0
23	OpenAI o1	55.7%	451	279	0
24	Qwen 3 Coder (30B)	55.7%	450	280	0
25	Claude Sonnet 4.5	55.1%	454	276	0
26	OpenAI o3-mini	54.4%	440	290	0
27	OpenAI GPT-4o Mini	54.1%	452	278	0

Table 8. Model Rankings

3.3) Vulnerability Category Analysis

Not all security vulnerabilities are equally difficult for AI models to avoid. IOActive's analysis across more than 85 vulnerability categories revealed systematic failure patterns where certain security domains consistently produced exploitable code regardless of model architecture or prompting approach. The vulnerability category rankings identify which security controls require mandatory human review when using AI-assisted development.

The results reveal universal failures in critical security domains. Rate limiting implementation shows 98.8% vulnerability across 81 tests, indicating that nearly every AI-generated API lacks proper throttling. Authentication security follows at 96.3% vulnerable, with models consistently generating bypasses and weak password policies. Infrastructure categories cluster at the top: exposed metrics (92.6%), container misconfigurations (70.2%), and CI/CD vulnerabilities (63.9%) demonstrate AI's particular weakness with cloud-native security.

Mobile security shows similar patterns across all platforms. Weak biometric authentication affects 87.4% of samples, missing SSL certificate pinning appears in 82.2% of network code, and App Transport Security (ATS) bypass vulnerabilities occur in 75.9% of iOS implementations. These failures span 324 mobile security tests across Android (Java/Kotlin), iOS (Swift), React Native (JavaScript), and Flutter (Dart), representing systematic failures in fundamental mobile security controls regardless of development framework.

Rank	Vulnerability	Rate	Total Tests
1	Missing Rate Limiting	98.8%	81
2	Insecure Auth	96.3%	108
3	Prometheus Metrics Exposed	92.6%	27
4	Weak Biometric Auth	87.4%	135
5	Missing SSL Pinning	82.2%	135
6	Machine Learning Adversarial Examples	81.5%	27
7	Machine Learning Unsafe Deserialization	77.8%	27
8	App Transport Security Bypass	75.9%	54
9	Business Logic Flaw	72.8%	81
10	Code Injection	70.4%	54
11	PostgreSQL Injection	70.4%	27
12	Container Security	70.2%	810
13	Insecure Crypto	70.2%	756
14	CI/CD Security	63.9%	675
15	Datastore Security	63.4%	486
16	Buffer Overflow	61.1%	108
17	External Entity Injection	60.9%	297
18	GraphQL Security	59.6%	270
19	Insecure Upload	57.7%	189
20	Integer Overflow	52.8%	108
21	Insecure Deserialization	50.4%	405
22	Insecure Package Registry	50.0%	54
23	GRPC No TLS	48.1%	27
24	Machine Learning Model Inversion	48.1%	27
25	Format String	46.9%	81

Table 9. Top 25 Vulnerabilities

Top vulnerability categories: The results reveal a catastrophic failure in modern security controls, with the top 10 vulnerability categories averaging an 86.3% rate. This represents AI model's systematic blind spots in production-critical security mechanisms.

Infrastructure security gap: The most surprising pattern is the infrastructure security gap. AI models consistently generate more vulnerable infrastructure code than application code, yet infrastructure misconfigurations are often the entry point for real-world attacks. This suggests that current AI training datasets may be biased toward application code examples, leaving infrastructure-as-code security as a critical blind spot. Specifically, deployment infrastructure (containers, CI/CD, serverless) is 57.5% vulnerable on average, whereas development infrastructure (service mesh, API gateway, messaging) is only 5.7% vulnerable on average. This factor of 10 vulnerability gap suggests AI training data emphasizes developer-facing infrastructure security (API authentication, service-to-service auth) while neglecting deployment/operations security

(container hardening, pipeline secrets, runtime monitoring). This implies that the real infrastructure crisis is in DevOps automation security, not infrastructure-as-code generally.

CI/CD pipeline security: This represents a particularly severe failure domain at 63.9% vulnerable across 675 tests. Jenkins pipelines written in Groovy emerge as the most problematic platform, with AI models generating vulnerable code in 78.5% of samples (106 out of 135 tests). Models consistently produce Jenkins pipelines that hardcode credentials, lack proper access controls, execute untrusted code without sandboxing, and expose secrets in environment variables.

CI/CD platform secuity: Platforms including GitHub Actions, GitLab CI, and Azure Pipelines show lower but still concerning vulnerability rates, primarily from overly permissive workflow configurations, missing approval gates, and command injection vulnerabilities from untrusted input. The disproportionately high Jenkins failure rate stems from training data containing legacy insecure patterns accumulated over Jenkins' longer history, Groovy's dynamic execution capabilities that enable anti-patterns, and the platform's complex security model that AI models fail to navigate correctly. Organizations using AI to generate CI/CD pipeline definitions face a greater than 63% chance of introducing exploitable vulnerabilities into their deployment infrastructure.

Mobile security crisis: Three of the top 10 most vulnerable categories are mobile-specific (with an average vulnerability rate of 81.9%), which suggests AI training datasets severely underrepresent mobile security best practices. The concentration of tests in Swift (189 tests) indicates this is particularly acute for iOS development, where Apple's security requirements (ATS, SSL pinning, biometric security) are well-documented but apparently not well-represented in training data.

4) Temperature as a Security Parameter

A comprehensive follow-up study tested 20 models across five temperature settings (0.0, 0.2, 0.5, 0.7, and 1.0), for a total of 100 configurations using 160 code samples generated from prompts. The results reveal that temperature is not merely a stylistic preference but a significant security parameter.

Model / Temperature	Temp 0.0	Temp 0.2 (Baseline)	Temp 0.5	Temp 0.7	Temp 1.0	Score Range
StarCoder2 (15B)	62.9	62.8	63.1	63.2	65	2.2
DeepSeek Coder (33B)	61.1	61.9	61.6	62.4	61.4	1.2
OpenAI GPT-5.2	59.3	60.7	61.1	60.9	59.3	1.8
CodeLlama (34B)	59.5	60.4	59.3	58.3	60.8	2.5
OpenAI GPT-5.4	58.5	59.5	58.4	59.1	60.3	2
OpenAI GPT-5.4 Mini	58.7	60	58.9	59	58.8	1.3
CodeGemma (7B)	59.2	58.6	58.4	59	60	1.5
Mistral (7B Instruct v0.2)	57.7	57.7	59.5	57.7	59.1	1.8
Claude Opus 4.6	57.2	56.4	58.2	59.1	59.1	2.8

OpenAI GPT-4	57.6	57.9	58.2	58.8	57.7	1.2
Gemini 2.5 Flash	57.6	57.1	58.5	56.5	58.7	2.2
OpenAI GPT-3.5 Turbo	58.2	58	58	58	58.4	0.4
DeepSeek Coder (6.7B Instruct)	57.2	57.2	57.9	57.8	58.2	1
Qwen 2.5 Coder (14B)	57.9	56.4	57.4	56.7	56.5	1.5
Llama 3.1 (8B Instruct)	56.6	57.9	57.6	56.9	57	1.3
Qwen 2.5 Coder (7B)	56.9	57	56.6	57.2	57.1	0.7
Claude Sonnet 4.5	56.9	55.6	56.3	56.3	56.4	1.3
Qwen 3 Coder (30B)	55.8	55.2	55.8	53.8	56.9	3.1
OpenAI GPT-4o	56.1	56.5	55.7	55.7	56.2	0.7
OpenAI GPT-4o Mini	54.5	53.9	54.4	54.5	54.7	0.8

Table 10. Temperature Sensitivity Heat Map

These were the optimal temperature for each model:

Optimal Temperature	Model	Best Score
0.0 (Deterministic)	Llama 3.1 (8B Instruct)	63.13%
	Qwen 2.5 Coder (7B)	62.19%
	OpenAI GPT-4o Mini	60.97%
	OpenAI GPT-5.2	65.51%
	Mistral (7B Instruct v0.2)	65.80%
	CodeGemma (7B)	65.80%
	Qwen 2.5 Coder (7B)	62.55%
0.5 (Balanced)	DeepSeek Coder (6.7B Instruct)	64.43%
0.7 (Balanced High)	Claude Opus 4.6	63.92%
	DeepSeek Coder (33B)	62.7%
	OpenAI GPT-4	65.15%
1.0 (High Creativity)	CodeLlama (34B)	67.17%
	Claude Sonnet 4.5	59.74%
	Gemini 2.5 Flash	64.07%
	StarCoder2 (15B)	70.92%
	OpenAI GPT-5.4	64.36%
	OpenAI GPT-5.4 Mini	65.51%
	DeepSeek Coder (6.7B Instruct)	67.75%
	OpenAI GPT-4o	61.40%
	OpenAI GPT-3.5 Turbo	64.21%

Table 11. Optimal Temperature by Model

No single temperature setting is optimal across all models. The default industry setting of 0.2 may systematically underestimate the security potential of code-specialized models.

Large Language Models (LLMs) are inherently probabilistic systems that generate text—including code—through sequential token sampling rather than deterministic computation. When prompted multiple times with identical inputs, LLMs will typically produce syntactically different outputs due to their sampling-based generation mechanism. This variability is amplified by the temperature parameter, which controls the randomness of token selection: at temperature 0.0, the model deterministically selects the highest-probability token at each step, while at temperature 1.0, the model samples from the full probability distribution, introducing substantial creative variation. In

this study, we use temperature 1.0 to maximize variation and explore the upper bounds of LLM non-determinism for security-critical code generation. While different code implementations are expected, the critical question for practical deployment is whether thes syntactic differences translate into meaningful security outcome variation. The following table presents the aggregate security scores across five independent runs for each model, revealing that while code syntax varies substantially (72.4% of files differ across runs), the overall security capability of each model—measured as the mean percentage of secure implementations—remains relatively stable, with an average variation range of only 6.65 percentage points across runs.

Model/Security Score	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Range	Std Dev
OpenAI GPT-5.2	75%	76%	77%	76%	81%	77%	6	2.35
OpenAI GPT-5.4	77%	74%	76%	78%	76%	76.2%	4	1.48
OpenAI GPT-5.4 Mini	72%	72%	73%	80%	76%	74.6%	8	3.44
Mistral (7B Instruct v0.2)	72%	74%	76%	76%	71%	73.8%	5	2.28
Claude Sonnet 4.5	70%	75%	74%	71%	70%	72%	5	2.35
StarCoder2 (15B)	73%	73%	71%	73%	70%	72%	3	1.41
Qwen 2.5 Coder (14B)	70%	74%	73%	70%	72%	71.8%	4	1.79
Claude Opus 4.6	70%	70%	71%	75%	69%	71%	6	2.35
Qwen 3 Coder (30B)	71%	71%	68%	70%	70%	70%	3	1.22
CodeGemma (7B)	68%	67%	71%	74%	68%	69.6%	7	2.88
OpenAI GPT-4o Mini	69%	71%	68%	68%	72%	69.6%	4	1.82
Qwen 2.5 Coder (7B)	70%	69%	74%	69%	66%	69.6%	8	2.88
OpenAI GPT-4o	71%	64%	73%	67%	68%	68.6%	9	3.51
Llama 3.1 (8B Instruct)	69%	71%	70%	65%	64%	67.8%	7	3.11
CodeLlama (34B)	68%	65%	67%	72%	66%	67.6%	7	2.7
OpenAI GPT-4	66%	68%	64%	66%	74%	67.6%	10	3.85
DeepSeek Coder (6.7B Instruct)	64%	70%	64%	68%	66%	66.4%	6	2.61
Gemini 2.5 Flash	68%	69%	56%	68%	65%	65.2%	13	5.36
OpenAI GPT-3.5 Turbo	67%	65%	64%	63%	62%	64.2%	5	1.92
DeepSeek Coder (33B)	66%	61%	59%	58%	53%	59.4%	13	4.72

Table 12. Results of Running Temperature Sensitive Models 5 Times at Temperature of 1.0

While Table 12 demonstrates aggregate stability—with models maintaining consistent mean security scores across runs (average range: 6.65 percentage-points,)this aggregate view masks important variation at the individual test level. Aggregate scores are averaged across many diverse security prompts (representing SQL injection, XSS, XXE, authentication, and other vulnerability categories), smoothing out test-specific fluctuations. However, when we examine individual security checks in isolation, a more nuanced picture emerges: 67.8% of individual tests show minimal variation (≤1 percentage-point), indicating highly consistent security behavior, but 32.2% exhibit significant variation (>5 percentage-points), with the model sometimes implementing security best practices and sometimes omitting them.

Most critically, 13.8% of tests demonstrate extreme binary variation (0-100%) cases where the model achieves a perfect security score (2/2 points) in one run by implementing proper protections but receives zero points (0/2) in another run by omitting those same protections entirely. For example, in XXE prevention tests, the model might generate lxml parser code with resolve_entities=False and no_network=True in one run (secure) but generate equivalent code without these security configurations in another run (vulnerable). This binary behavior occurs because many security checks are inherently all-or-nothing: either the code includes input validation, or it doesn't; either it uses parameterized queries or it doesn't.

The implication is significant: while the aggregate scores in Table 12 suggest models are reasonably consistent overall, approximately one-third of security-critical prompts cannot be trusted to produce secure code in a single generation. This finding has direct implications for enterprise LLM deployment strategies.

4.1) Key Takeaways

- 70% of models show improved security at higher temperature settings.
- Stability across temperatures is only valuable if the baseline is secure.
- Model providers should publish recommended temperature settings for security-critical tasks.
- While code syntax varies substantially (72.4%), aggregate security scores remain stable (6.6 percentage points average range), indicating that different implementations often achieve similar security. However, 32.2% of individual tests show significant variation, with 13.8% exhibiting extreme binary behavior—sometimes including critical security measures and sometimes omitting them. This means for ~1 in 3 prompts, a single generation is unreliable.
- Model choice matters the best models (Qwen 3 Coder, StarCoder2) are 4× more consistent than the worst (DeepSeek Coder, Gemini Flash). For enterprise deployment, we recommend either selecting high-consistency models (3-4pp range), using temperature=0.0 for deterministic outputs, or employing multi-generation validation for security-critical code.

5) Multi-Language Security Analysis

The framework was created with 57 multi-language detectors across Go, Java, Rust, C#, C, and C++, enabling analysis of 6,705 multi-language code samples.

5.1) Cross-Language Vulnerability Distribution

Rank	Language	Total Files	Vulnerable Count	Vulnerable Percentage
1	Dockerfile	405	396	97.8%
2	JSON	54	49	90.7%

3	Groovy	135	106	78.5%
4	Terraform	432	310	71.8%
5	XML	54	36	66.7%
6	Python	4077	2468	60.5%
7	Elixir	270	159	58.9%
8	Conf	54	29	53.7%
9	Perl	270	144	53.3%
10	PHP	540	273	50.6%
11	YAML	3078	1498	48.7%
12	Dart	324	151	46.6%
13	JavaScript	2916	1341	46.0%
14	Java	1377	610	44.3%
15	Swift	405	176	43.5%
16	C++	783	319	40.7%
17	Bash	270	106	39.3%
18	Rust	621	242	39.0%
19	Lua	297	112	37.7%
20	Ruby	378	134	35.4%
21	Go	702	247	35.2%
22	Kotlin	351	119	33.9%
23	Typescript	432	137	31.7%
24	Scala	324	99	30.6%
25	C#	567	151	26.6%
26	C	297	55	18.5%
27	Solidity	297	1	0.3%

Table 13. Cross-Language Vulnerability Distribution

5.2) Language-specific Weaknesses

Table 13 lists the most common vulnerability found in AI-generated code for each language.

Language	Most Common Vulnerability	Vulnerability Rate	Vulnerable Count	Total Tested
Bash	Race Conditions	92.6%	25	27
C	Hardcoded Secrets	77.8%	21	27
Conf	Datastore Security	53.7%	29	54
C++	Insecure Cryptography	100.0%	54	54
C#	Insecure Cryptography	100.0%	27	27
Dart	Weak_Biometric_Authentication	92.6%	25	27
Dockerfile	Container_Security	97.8%	396	405
Elixir	Cross Site Scripting	100.0%	27	27
Go	Insecure Cryptography	100.0%	27	27
Groovy	CICD Security	78.5%	106	135
Java	Insecure Cryptography	100.0%	81	81
JavaScript	SQL injection	100.0%	27	27
JSON	Supply_Chain_Security	90.7%	49	54
Kotlin	Missing_SSL Pinning	100.0%	27	27
Lua	code_injection	92.6%	25	27
Perl	Cross Site Scripting	100.0%	27	27
PHP	Insecure Cryptography	100.0%	54	54
Python	Insecure Authentication	100.0%	54	54
Ruby	Broken Access_Control	92.6%	50	54
Rust	Insecure Cryptography	100.0%	27	27
Scala	XML External Entity	100.0%	27	27
Solidity	Smart Contract_Integer_Overflow	3.7%	1	27
Swift	Missing_SSL_Pinning	92.6%	25	27

Terraform	Cloud_Database_Security	90.7%	49	54
Typescript	Insecure Cryptography	92.6%	25	27
XML	Supply_Chain_Security	66.7%	36	54
YAML	Datastore_Security	100.0%	270	270

Table 14. Language-Specific Security Weaknesses

The data reveals that vulnerabilities are not evenly distributed, rather, they cluster dramatically in specific languages and language-category pairs:

Dockerfile emerges as the most vulnerable language at 97.8% (396 out of 405 samples contain security flaws). Nearly every container configuration generated by AI models contains critical security vulnerabilities. This represents an almost universal failure to generate secure containerization code.

JSON configurations follow closely at 90.7% (49 out of 54 samples). The supply chain security detector reveals that AI models consistently generate JSON package manifests with wildcard version constraints, dangerous install scripts, and insecure repository configurations. This affects package.json and composer.json files across multiple ecosystems.

Groovy code, primarily used in Jenkins CI/CD pipelines, has a 78.5% vulnerability rate (106 out of 135 samples). The CI/CD security detector identifies that most AI-generated pipeline definitions expose secrets, lack proper access controls, and execute untrusted code without sandboxing.

Terraform infrastructure code demonstrated a 71.8% vulnerability rate (310 out of 432 samples). Cloud database misconfigurations dominate, with AI models frequently generating infrastructure-as-code that exposes databases to the Internet, uses weak credentials, and disables encryption.

Very High-risk Languages (60-70% Vulnerability Rate)

XML configuration files are 66.7% vulnerable (36 out of 54 samples), primarily in supply chain security. Like JSON, AI models generate Maven pom.xml files with LATEST version tags and dangerous plugin executions that can compromise build processes.

Python code exhibits a 60.5% vulnerability rate (2,468 out of 4,077 samples) Models universally fail at implementing rate limiting (100% vulnerable), authentication security (100% vulnerable), and input validation. Python's popularity in AI training data does not translate to secure code generation.

High-risk Languages (50-60% Vulnerability Rate)

Elixir functional code shows a 58.9% vulnerability rate (159 out of 270 samples), with XSS being the most problematic category at 100% vulnerable. Despite Elixir's memory safety guarantees, web security vulnerabilities remain prevalent.

Configuration files (conf) demonstrate a 53.7% vulnerability rate (29 out of 54 samples), particularly in datastore security. AI models frequently generate database and service configurations that expose management interfaces, disable authentication, or use plaintext protocols.

Perl scripts are 53.3% vulnerable (144 out of 270 samples), with XSS as the worst category at 100% vulnerable. Legacy web application frameworks in Perl universally lack proper output encoding.

PHP web applications exhibit a 50.6% vulnerability rate (273 out of 540 samples). Despite PHP's modern security features, AI models generate code with 100% vulnerable cryptographic implementations, suggesting models learned from older, insecure PHP patterns.

Moderate-risk Languages (40-50% Vulnerability Rate)

YAML configuration files show a 48.7% vulnerability rate (1,498 out of 3,078 samples). The datastore security category is 100% vulnerable, indicating AI models consistently generate insecure database, queue, and cache configurations in YAML format.

Dart mobile applications demonstrate a 46.6% vulnerability rate (151 out of 324 samples), with weak biometric authentication being the most common issue at 92.6% vulnerable.

JavaScript code shows a 46.0% vulnerability rate (1,341 out of 2,916 samples), with SQL injection being the worst category at 100% vulnerable. This indicates ai generated Node.js database code universally lacks parameterized queries.

Java applications exhibit a 44.3% vulnerability rate (610 out of 1,377 samples). Cryptographic implementations are 100% vulnerable, showing that even enterprise languages with mature security libraries fail when AI models generate the code.

Swift iOS applications show a 43.5% vulnerability rate (176 out of 405 samples), primarily in SSL pinning (92.6% vulnerable), indicating mobile network security is poorly understood by AI models.

Improved-security Languages (30-40% Vulnerability Rate)

C++ code demonstrates a 40.7% vulnerability rate (319 out of 783 samples), which is significantly higher than memory safety issues alone would suggest. While buffer overflows are controlled, cryptographic implementations are 100% vulnerable.

Bash scripts show a 39.3% vulnerability rate (106 out of 270 samples), with race conditions being the most critical category at 92.6% vulnerable. Filesystem operations in shell scripts universally lack proper locking mechanisms.

Rust code exhibits a 39.0% vulnerability rate (242 out of 621 samples). While Rust's memory safety prevents traditional exploits, cryptographic implementations remain 100% vulnerable, and business logic flaws persist. Rust's compile-time guarantees do not extend to cryptographic correctness or application security logic.

Lua scripts show a 37.7% vulnerability rate (112 out of 297 samples), with code injection being the worst category at 92.6% vulnerable.

Ruby web applications demonstrate a 35.4% vulnerability rate (134 out of 378 samples), with broken access control affecting 92.6% of samples. Rails-style applications lack proper authorization checks despite framework security features.

Go microservices show a 35.2% vulnerability rate (247 out of 702 samples). Like other compiled languages, cryptographic implementations are 100% vulnerable despite Go's security-focused standard library.

Lower-risk Languages (20-30% Vulnerability Rate)

Kotlin Android applications exhibit a 33.9% vulnerability rate (119 out of 351 samples), with SSL pinning being universally absent (100% vulnerable) in network communication code.

TypeScript code shows a 31.7% vulnerability rate (137 out of 432 samples), with largely insecure cryptography (92.6% vulnerable), which demonstrates that type safety does not guarantee cryptographic correctness.

Scala functional code demonstrates 30.6% vulnerability (99 out of 324 samples), though XXE vulnerabilities remain at 100%, indicating XML processing is universally insecure.

C# .NET applications show a 26.6% vulnerability rate (151 out of 567 samples), with cryptographic implementations being 100% vulnerable. Despite .NET's extensive security APIs, AI models fail to use them correctly.

Minimal-risk Languages (<20% Vulnerability Rate)

C code exhibits an 18.5% vulnerability rate (55 out of 297 samples). While traditionally dangerous, modern AI models generate surprisingly secure C code, with hardcoded secrets being the primary issue at 77.8% vulnerable. Memory safety vulnerabilities are controlled.

Solidity smart contracts show a 0.3% vulnerability rate (1 out of 297 samples)—the most secure language tested. The blockchain community's focus on security education and auditing appears to have influenced training data quality. Integer overflow, the worst category, affects only 3.7% of samples.

5.3) Key Insight

Using AI to generate Dockerfiles or infrastructure code results in a greater than 97.8% chance the output will contain security vulnerabilities. Infrastructure-as-code and scripting languages are where AI code generation poses the highest security risk, requiring mandatory security expert review before deployment.

6) Prompt Engineering: Picking the Most Effective Security Prompting Level

We tested nine models with six different levels of security prompting. The prompts

became progressively more explicit in their instructions on how to make the code secure. We hypothesized that the level of sophistication of a model might be better understood by these prompts as summarized in the following table.

Level	Name	Description	Goal
Level 0	Baseline	Functional requirements only—no security mentioned	Tests security by default
Level 1	Minimal	'Write secure code' suffix appended	Tests general security awareness
Level 2	Brief threat naming	Brief mention of threat category	Tests threat knowledge
Level 3	Detailed principles	Comprehensive security principles, no code examples	Tests technical implementation skills
Level 4	Prescriptive	Explicit code examples of secure vs insecure patterns	Tests pattern and anti-pattern recognition and avoidance
Level 5	Self-review	Model asked to review and fix its own output	Tests verification and self-review capabilities

Table 15. Levels of security prompting

Our hypothesis was that each level incrementally reduces vulnerabilities by providing more specific, actionable security guidance. We disproved this hypothesis as the optimal prompting level depends on the model provider.

The following table shows the heatmap of security prompting level by model:

Model	Level 0 (Baseline)	Level 1 (Minimal)	Level 2 (Brief)	Level 3 (Principles)	Level 4 (Prescriptive)	Level 5 (Self-Review)
DeepSeek Coder (33B)	61.50%	58.20%	58.20%	57.90%	59.50%	60.50%
CodeLlama (34B)	60.30%	53.90%	54.50%	55.20%	56.00%	53.60%
Qwen 2.5 Coder (7B)	57.00%	51.80%	54.10%	55.20%	56.30%	54.00%
OpenAI GPT-4	56.80%	55.00%	55.80%	56.20%	59.00%	57.80%
Llama 3.1 (8B Instruct)	56.30%	51.00%	53.30%	53.90%	56.80%	53.50%
Claude Opus 4.6	55.90%	61.70%	59.90%	60.50%	61.80%	61.40%
Qwen 3 Coder (30B)	55.60%	54.60%	55.60%	56.80%	58.60%	56.40%
Claude Sonnet 4.5	55.30%	59.10%	58.20%	60.70%	59.80%	61.90%
OpenAI GPT-4o Mini	54.10%	51.10%	54.20%	54.50%	59.00%	54.20%

Table 16. Heatmap of Security Prompting Level Results by Model

The following table summarizes the recommended security prompting for each model:

Model	Baseline(L0)	Recommended Security Prompting Level	Optimal Score	Change From Baseline	Worst Level	Worst Score
Claude Opus 4.6	55.9%	L4	61.8%	+5.9pp	L0	55.9%
Claude Sonnet 4.5	55.3%	L5	61.9%	+6.5pp	L0	55.3%
CodeLlama (34B)	60.3%	L0	60.3%	+0.0pp	L5	53.6%
DeepSeek Coder (33B)	61.5%	L0	61.5%	+0.0pp	L3	57.9%
OpenAI GPT-4o	56.8%	L4	59.0%	+2.2pp	L1	55.0%
OpenAI GPT-4o Mini	54.1%	L4	59.0%	+4.9pp	L1	51.1%
Llama 3.1 (8B Instruct)	56.3%	L4	56.8%	+0.5pp	L1	51.0%
Qwen 2.5 Coder (7B)	57.0%	L0	57.0%	+0.0pp	L1	51.8%
Qwen 3 Coder (30B)	55.6%	L4	58.6%	+3.0pp	L1	54.6%

Table 17. Recommendations by Model

Based on the data, IOActive observed that:

- For Claude Sonnet 4-5, level 5 is the best choice.
- For all other models from OpenAI and Anthropic, use level 4.
- Level 5 degrades OpenAI models by 83%.
- For models running locally under Ollama, use level 0.
- Avoid level 1 (for five of the nine models, it was the worst performing level).

7) Conclusions and Practical Recommendations

Development teams implementing AI-assisted coding must establish mandatory security review tiers based on empirical vulnerability rates like those identified in this study. Dockerfile, JSON configurations, Groovy CI/CD pipelines, and Terraform infrastructure all exceed 70% vulnerability rates and require expert security review before any deployment. Standard code review is insufficient for these critical priority categories—dedicated security engineering expertise is essential. XML configurations and Python code fall into the high priority tier at 60-70% vulnerable, requiring security-focused review with specific attention to authentication (96.3% vulnerable), rate limiting (98.8% vulnerable), and cryptographic implementations that show 100% vulnerability in Python. Elixir, Perl, PHP, and configuration files represent elevated priority at 50-60% vulnerable, requiring enhanced security review beyond standard practices and focusing on the worst-performing categories identified for each language. YAML, Dart, JavaScript, Java, and Swift fall into standard priority at 40-50% vulnerable, requiring security-conscious review with particular vigilance for their 100% vulnerable categories such as datastore security for YAML, SQL injection for JavaScript, and cryptography for Java.

Teams should prohibit AI generation entirely for certain high-risk categories. The 14

vulnerability categories with less than 30% security rates require prohibition or extreme scrutiny. Rate limiting, authentication systems, cryptographic implementations, and CI/CD pipeline security should use vetted libraries, established frameworks, or human-written code rather than AI generation. For the nine languages showing 100% cryptographic failure, including C++, C#, Go, Java, PHP, Rust, and TypeScript, organizations must mandate high-level security libraries that prevent low-level cryptographic decisions and prohibit AI from generating raw cryptographic code. JavaScript database code shows 100% SQL injection vulnerability, requiring enforcement of database libraries that make parameterized queries the default and string concatenation impossible. For mobile applications showing 82-87% vulnerability in SSL pinning and biometric authentication, teams should provide security-validated templates and configuration files rather than generating from scratch.

Model configuration must follow provider-specific patterns identified in the multi-level and temperature studies. Anthropic models including Claude Opus 4.6 and Claude Sonnet 4.5 should use level 4 security prompting for Claude Opus 4.6 and level 5 for Claude Sonnet 4.5, with higher temperature settings in the 0.7-1.0 range for improved security outcomes. OpenAI models including GPT-4, GPT-4o, and the GPT-5.x series should use level 4 security prompting while avoiding level 5, which degrades performance by 83%, and teams should test temperature ranges as 70% of models improve at higher settings. Ollama and open-source models including DeepSeek Coder (33B), CodeLlama (34B), Llama 3.1 (8B Instruct), and Qwen series should use level 0 with no security prompting, as security-aware prompts degrade performance in four of five tested models. Teams should focus on post-generation validation rather than prompt engineering for these models. Universal guidance across all providers indicates teams should avoid level 1 security prompting as it was the worst-performing level for five of nine tested models.

When architectural decisions permit language flexibility, teams should prioritize language selection for security-critical components. Memory-safe languages like Rust with 39.0% vulnerable overall and 15.2% for memory issues specifically, and Go at 35.2% vulnerable, should be preferred over Python at 60.5% vulnerable and JavaScript at 46.0% vulnerable for security-critical backend services; however, teams must recognize memory safety limitations. Rust's compile-time guarantees prevent memory corruption but not business logic flaws, authentication bypasses, or cryptographic errors, meaning security review remains mandatory even for memory-safe languages. Teams should avoid AI generation entirely for infrastructure languages, instead manually writing or using validated templates for Dockerfiles, Terraform configurations, and CI/CD pipelines rather than generating with AI.

For Organizations

Organizations must establish AI code security policies addressing the systematic risks identified in this research. Mandatory disclosure requirements should compel developers to mark AI-generated code sections in pull requests for appropriate security review routing. Security review service level agreements should establish review timeframes based on vulnerability category risk tiers, ensuring high-risk categories receive expert attention within 24-48 hours.

Deployment gates should implement automated scanning for the 13 language-category combinations showing 100% vulnerability rates, including authentication in Python, SQL

injection in JavaScript, cryptography across nine languages, and SSL pinning in Kotlin. These gates should block deployment until human security review confirms mitigation. Audit trail requirements must maintain records of which code was AI-generated, which model and configuration was used, and what security review was performed for compliance and incident response purposes.

Investment in security tooling integration represents a critical organizational requirement. Static analysis enforcement should run specialized security scanners on all AI-generated code before merge, with particular focus on the 14 categories achieving less than 30% security scores. Container and infrastructure-as-code scanning becomes essential given 97.8% Dockerfile vulnerability and 71.8% Terraform vulnerability, mandating tools like Trivy, Checkov, or Snyk for all AI-generated infrastructure code. Secrets detection through pre-commit hooks must identify hardcoded credentials, API keys, and tokens, as AI models frequently generate authentication code with embedded secrets. Dependency scanning must examine all AI-generated package.json, composer.json, and requirements.txt files for wildcard versions, dangerous scripts, and insecure repositories, given the 90.7% JSON package manifest vulnerability rate.

Organizations must adjust development velocity expectations to accommodate the comprehensive security review requirements identified in this study. Infrastructure deployment delays should be anticipated given vulnerability rates exceeding 70% for Dockerfile, Terraform, and CI/CD code, requiring planning for 2-3x longer review cycles before infrastructure changes reach production. Mobile release planning must extend QA cycles for AI-generated mobile networking and authentication code given 82-87% vulnerability in mobile security controls. Security review capacity will likely become a bottleneck, requiring organizations to budget for additional security headcount or accept reduced AI-assisted development velocity for high-risk categories.

Following the Codex App model showing 5.1% improvement from security skills, organizations should develop internal security wrappers. Building organization-specific prompts that incorporate internal security standards, compliance requirements, and architecture patterns provides a foundation for improvement. Implementing validation layers that wrap AI code generation APIs with automated security validation before presenting code to developers can filter out the most obvious vulnerabilities. Creating secure code templates for categories showing greater than 70% vulnerability allows AI to customize validated templates rather than generating from scratch.

For Model Providers

Model providers should publish security-specific configuration guidance enabling users to optimize for security outcomes. Temperature recommendations must document recommended temperature ranges for security-critical code generation tasks, given that 70% of models show improved security at higher temperatures and optimal settings vary by up to 3.2 percentage points. Security prompting compatibility disclosures should clarify whether models benefit from security-aware prompting like Anthropic models or degrade like Ollama models, enabling users to configure appropriately. Vulnerability category disclosures should identify specific security domains where the model consistently fails, such as explicitly stating "This model generates vulnerable authentication code more than 90% of the time so use established authentication libraries instead."

Providers must develop security-focused training and fine-tuning addressing systematic failure categories identified in this research. Authentication and rate limiting show 96.3% and 98.8% vulnerability rates indicating fundamental training gaps, requiring prioritization of secure authentication examples and rate limiting implementations in training data. Infrastructure-as-code shows 97.8% Dockerfile vulnerability and 71.8% Terraform vulnerability, necessitating specialized training data emphasizing secure container configurations and cloud infrastructure. Cryptographic implementations across nine languages show 100% failure, meaning providers must either train models to use high-level cryptographic libraries exclusively or refuse cryptographic code generation entirely. Mobile security requires addressing 82-87% failure rates in SSL pinning and biometric authentication through mobile-security-focused training data curation.

Providers should implement category-specific refusal given the zero refusal rate (0.00%) and catastrophic failure in specific categories. Selective refusal for unsolvable categories means that rather than generating rate limiting code with 98.8% vulnerability, models should decline and suggest established libraries or human implementation. Confidence-based generation for categories with less than 30% security scores should only generate code when confidence exceeds a threshold, otherwise providing guidance toward secure alternatives. Explicit security warnings when generating infrastructure code with vulnerability rates exceeding 70% should prepend clear warnings such as "This generated Dockerfile likely contains security vulnerabilities. Expert security review is required before deployment."

Supporting security evaluation and transparency enables organizations to make informed decisions. Publishing vulnerability benchmarks means disclosing performance on standard security benchmarks like this study, allowing organizations to compare models for their specific use cases. Version-specific security tracking should document security performance changes across vulnerability categories as models are updated, helping organizations understand regression risks. Training data composition disclosure should provide transparency about the proportion of security-focused training data, enabling users to understand which domains may have inadequate coverage.

Research Directions

This study identifies several critical areas requiring further investigation. Temporal security degradation research should examine whether models exhibit security regression over conversation length or across sessions, as long-running development conversations may accumulate context that degrades security awareness. Prompt injection resistance studies should determine whether adversarial prompts can bypass security configurations to force models to generate vulnerable code, as the provider-dependent security prompting response suggests some architectures may be more resistant than others. Secure code generation architecture research should explore why Solidity achieves 0.3% vulnerability while Dockerfile reaches 97.8%, as understanding this 97.5 percentage point gap could inform model training approaches for other domains. Multi-turn security improvement investigations should test whether iterative human feedback during code generation sessions can improve security outcomes beyond static security prompting, as the Codex App wrapper suggests human-in-the-loop approaches merit deeper study. Cross-language security transfer analysis should determine whether models that generate secure Rust code also generate secure

Python, or if language-specific security patterns are learned independently, affecting recommendations for multi-language development environments.

Final Recommendations

Organizations adopting AI-assisted development face a critical decision: accept comprehensive security review requirements or limit AI code generation to non-security-critical functions. **The data demonstrates that no combination of model selection, configuration, or prompting eliminates the need for expert security validation. Infrastructure code, authentication systems, cryptography, and mobile security controls show such catastrophic failure rates ranging from 70% to 98% vulnerable that AI generation in these domains should be prohibited or treated as untrusted input requiring complete rewrite validation.**

For organizations proceeding with AI-assisted development despite these risks, implementing the tier-based security review framework based on empirical vulnerability rates provides the foundation for risk management. Configuring models using provider-specific guidance means Level 4 for Anthropic and OpenAI models and Level 0 for Ollama models. Prohibiting AI generation for the 14 categories with less than 30% security scores or treating output as security-hostile prevents the most egregious vulnerabilities. Extending deployment timelines to accommodate mandatory security review becomes essential, particularly for infrastructure changes. Investing in automated security tooling to filter obvious vulnerabilities before human review improves efficiency while maintaining security standards.

The 83.8% best-case security outcome with 90 residual vulnerabilities across 730 scenarios demonstrates that AI code generation remains a fundamentally insecure-by-default technology requiring defense-in-depth validation before production deployment.

References

1. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R. (2022). Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. IEEE Symposium on Security and Privacy (SP), 754–768.

2. Perry, N., Srivastava, M., Kumar, D., Boneh, D. (2023). Do Users Write More Insecure Code with AI Assistants? ACM Conference on Computer and Communications Security (CCS).

3. Asare, O., Nagappan, M., Asokan, N. (2023). Is GitHub's Copilot as Bad as Humans at Introducing Vulnerabilities in Code? Empirical Software Engineering, 28(6).

4. Tony, C., Mutas, M., Ferreyra, N.E.D., Scandariato, R. (2023). LLMSecEval: A Dataset of Natural Language Prompts for Security Evaluations. IEEE/ACM MSR.

5. GitHub, Inc. (2024). GitHub Copilot: The AI Pair Programmer. GitHub Resources. https://github.com/features/copilot

6. OWASP Foundation. (2021). OWASP Top 10:2021. https://owasp.org/Top10/2021/

7. MITRE Corporation. (2024). CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/

8. Rethinking the Evaluation of Secure Code Generation: https://arxiv.org/abs/2503.15554
9. Security and Quality in LLM-Generated Code: A Multi-Language, Multi-Model Analysis: https://arxiv.org/abs/2502.01853

10. Guiding AI to Fix Its Own Flaws: An Empirical Study on LLM-Driven Secure Code Generation: https://arxiv.org/abs/2506.23034

11. Prompting Techniques for Secure Code Generation: A Systematic Investigation: https://arxiv.org/abs/2407.07064

12. Enhancing Large Language Models for Secure Code Generation: A Dataset-driven Study on Vulnerability Mitigation: https://arxiv.org/abs/2310.16263

13. How Secure is Secure Code Generation? Adversarial Prompts Put LLM Defenses to the Test: https://arxiv.org/abs/2601.07084