AI Security

SecureCode v2.0: Production-Grade Dataset for Security-Aware Code Generation

SecureCode v2.0: Production-Grade Dataset for Security-Aware Code Generation

● **Author:** Scott Thornton, perfecXion.ai ● **Published:** January 25, 2026 ● **Read Time:** 10 minutes

SecureCode v2.0: A Production-Grade Dataset for Training Security-Aware Code Generation Models

Scott Thornton

December 2025

Abstract

Despite widespread adoption of large language models for code generation, recent studies have found AI assistants producing vulnerable outputs in 45% of security-relevant scenarios, introducing security flaws into production systems at scale. Existing secure coding datasets exhibit significant limitations: they lack incident grounding, provide insufficient scale necessary for modern training, and they miss the operational security context developers need for production deployments.

We present **SecureCode v2.0**, a production-grade dataset of **1,215 security-focused coding examples** that fully comply with structural validation standards and all underwent expert security review. Every example ties directly to actual documented security incidents with CVE references, provides both vulnerable and secure implementations, demonstrates concrete attacks, and includes defense-in-depth operational guidance. The dataset covers **11 vulnerability categories** (complete OWASP Top 10:2025 plus AI/ML Security Threats) across **11 languages total** (10 programming languages: Python, JavaScript, Java, Go, PHP, C#, TypeScript, Ruby, Rust, Kotlin + YAML for infrastructure-as-code).

Our quality assurance framework ensures complete incident grounding with every example tied to a documented security incident (CVE, security advisory, or breach report). Each example provides operational security guidance, including SIEM integration strategies, infrastructure hardening recommendations (Docker, AppArmor, WAF configurations), and testing approaches using language-appropriate frameworks. The dataset uses a novel 4-turn conversational structure that mirrors actual developer-AI interactions, escalating from basic implementations to advanced security considerations and defense-in-depth operational guidance.

Our contributions include:

1. a production-grade dataset of 1,215 rigorously validated unique examples split into 989 training, 122 validation, and 104 test examples

2. an automated validation framework ensuring dataset consistency
3. a 4-turn conversational structure capturing realistic security workflows
4. comprehensive operational security guidance with SIEM integration strategies
5. full language-specific implementation fidelity
6. open-source release of data, validation tools, and benchmarking protocols.

1. Introduction

1.1 The Security Crisis in AI-Generated Code

AI coding assistants produce vulnerable code in 45% of generated implementations [1]. Veracode's 2025 GenAI Code Security Report analyzed code generated by leading AI assistants and found that nearly half of all security-relevant implementations contained Common Weakness Enumeration (CWE) vulnerabilities. This indicates a systematic risk in AI-assisted development that compounds security debt across millions of developers. The risk surface has scaled as adoption has increased.

The issue extends beyond individual bugs. AI-generated vulnerabilities enter production codebases silently, without the traditional code review scrutiny applied to human-written code. Developers trust AI assistants to produce functional implementations, but these tools lack the security context to recognize when "functional" means "exploitable." Apiiro (2025) found that AI copilots introduced 322% more privilege escalation paths and 153% more architectural design flaws compared to manually-written code, while also generating 10× more security findings overall, demonstrating that AI tools actively degrade security practices [2].

This can create a multiplier effect where vulnerable patterns suggested by AI assistants propagate across multiple projects. SQL injection flaws may spread through microservices architectures. Authentication bypasses can replicate across API endpoints. Cryptographic failures may multiply through mobile applications. The scale of AI adoption suggests this represents a systematic risk to software security.

A key contributing factor is that LLMs trained on public code repositories learn from millions of vulnerable examples. Stack Overflow answers from 2010 showing insecure MySQL queries. GitHub repositories implementing broken authentication. Tutorial code demonstrating SQL injection vulnerabilities as "simple examples." These models learn what code looks like, but they do not necessarily learn what secure code requires.

1.2 Why Existing Datasets Fall Short

Existing secure coding datasets exhibit significant limitations for training security-aware language models. We analyzed four widely-used datasets in security research: CWE-Sans (372 examples), Juliet Test Suite (~81,000-86,000 synthetic test cases for C/C++ and Java), SARD (~170,000-200,000 test programs), and

Draper VDISC (1.27 million C examples). While these datasets serve their intended purposes, we found they exhibit some critical gaps when applied to LLM training.

Scale versus quality presents inherent trade-offs. Juliet provides ~81,000-86,000 test cases designed for testing static analysis tools, but lacks connections to real-world incidents. These synthetic test cases demonstrate CWE patterns in isolation, teaching models to recognize textbook vulnerabilities that may not reflect how attacks occur in production environments. SARD offers ~170,000-200,000 test programs but fewer than 5% reference documented security incidents. Synthetic training data fails to capture the contextual factors making vulnerabilities exploitable in production environments.

Incident grounding is limited. Based on our manual audit of CWE-Sans metadata (n=372 examples, 100% coverage), approximately 18% of examples reference actual CVEs or documented breaches—fewer than one in five examples. The remaining examples are synthetic demonstrations that lack the production context necessary for understanding exploitation. Real-world attacks exploit edge cases, framework-specific behaviors, and integration failures that rarely appear in manufactured examples.

Format limitations hinder conversational training. Existing datasets use code-only formats—vulnerable snippet paired with secure snippet. This approach does not capture how developers interact with AI assistants in practice. Real development conversations escalate through multiple turns as developers ask about functionality, then scaling, performance, and edge cases. AI assistants must maintain security context throughout this workflow, but existing datasets do not model these multi-turn interactions.

Operational security guidance is absent. Existing datasets provide vulnerable and patched code implementations without detection mechanisms, logging strategies, or defense-in-depth guidance. For production systems, secure code implementation represents only one component of comprehensive security. Organizations require detection rules, monitoring strategies, incident response procedures, and graceful degradation when primary controls fail.

1.3 SecureCode v2.0: A Production-Grade Solution

We developed SecureCode v2.0 to address these limitations systematically through production-grade training data. The dataset provides **1,215 rigorously validated unique examples** achieving full compliance with structural validation standards and expert security review. Every example references documented CVEs or security incidents. Every example provides both vulnerable and secure implementations. Every example demonstrates concrete attacks and includes defense-in-depth operational guidance including SIEM integration strategies, infrastructure hardening recommendations, and comprehensive testing approaches.

Incident grounding is a critical requirement for production applicability. We mined CVE databases from 2017-2025, analyzed OWASP Top 10 documentation, reviewed security breach reports, and studied bug bounty disclosures. Each example ties to a specific incident: the 2017 Equifax breach (CVE-2017-5638) costing $425 million from Apache Struts 2 Jakarta multipart parser RCE (OGNL injection), the 2019 Capital One SSRF attack exposing 100 million customer records, the deserialization vulnerabilities that compromised dozens of financial institutions. These represent documented failures with measurable business impact rather than hypothetical scenarios.

Conversational structure mirrors actual development workflows. We structured examples as 4-turn conversations. Turn 1: developer requests specific functionality ("build user authentication with JWT tokens"). Turn 2: AI assistant provides both vulnerable and secure implementations with attack demonstrations. Turn 3: developer asks advanced questions ("how does this scale to 10,000 concurrent users?"). Turn 4: AI assistant delivers defense-in-depth guidance covering logging, monitoring, detection, and operational security.

This structure captures how security knowledge transfers during actual development workflows. Developers do not request "secure and insecure authentication" in abstract terms. They request authentication solving specific problems, then iterate toward production-ready security through follow-up questions. The conversational format trains models on realistic interaction patterns.

Production quality through systematic validation. We developed an automated validation framework enforcing structural quality standards: 4-turn conversation structure compliance, proper CVE formatting (CVE-YYYY-NNNNN or explicit null), valid programming language tags, minimum content length requirements, and security control completeness. The compliance journey progressed from 47.2% (397 of 841 training examples passing all validation checks) to full compliance through systematic remediation across five fix categories.

We addressed 452 CVE format issues where examples referenced security incidents without proper CVE-YYYY-NNNNN formatting. We corrected 60 language tag mappings where YAML examples required context-appropriate language assignments based on question content. We enhanced 86 examples with additional defense-in-depth guidance. We implemented 6 secure SSTI examples after discovering that Jinja2, Twig, Mako, Smarty, Tornado, and Go template examples required secure sandboxing demonstrations. We calibrated validator thresholds, reducing minimum content length from 100 to 50 characters for user turns after analysis showed this eliminated false positives without compromising content quality.

Comprehensive security coverage. SecureCode v2.0 spans **11 vulnerability categories** across **11 languages total** (10 programming languages: Python, JavaScript, Java, Go, PHP, C#, TypeScript, Ruby, Rust, Kotlin + YAML for infrastructure-as-code), providing complete coverage of OWASP Top 10:2025 plus AI/ML Security Threats:

- **A01:2025 Broken Access Control** (224 examples, 18.4%): IDOR, privilege escalation, authorization bypasses, path traversal, SSRF against cloud metadata
- **A07:2025 Authentication Failures** (199 examples, 16.4%): JWT vulnerabilities, OAuth flaws, session management, credential stuffing, MFA bypass
- **A02:2025 Security Misconfiguration** (134 examples, 11.0%): Framework misconfigurations, security headers, CORS, cloud configurations
- **A05:2025 Injection** (125 examples, 10.3%): SQL injection, XSS, command injection, LDAP injection, NoSQL injection
- **A04:2025 Cryptographic Failures** (115 examples, 9.5%): Weak algorithms, key management, encryption failures, insecure hashing

- **A03:2025 Software Supply Chain Failures** (85 examples, 7.0%): Supply chain security, vulnerable packages, unpatched dependencies
- **A06:2025 Insecure Design** (84 examples, 6.9%): Architectural vulnerabilities, workflow bypasses, business logic flaws
- **A08:2025 Software or Data Integrity Failures** (80 examples, 6.6%): Data validation, integrity checks, insecure deserialization
- **Unknown** (60 examples, 4.9%): Multi-category or complex incidents spanning multiple OWASP categories
- **A09:2025 Security Logging & Alerting Failures** (59 examples, 4.9%): Security event logging, audit trails, missing detection
- **AI/ML Security Threats** (50 examples, 4.1%): Prompt injection, model extraction, adversarial attacks, RAG poisoning

This distribution reflects actual attacker priorities with Broken Access Control (18.4%, including merged SSRF examples) and Authentication Failures (16.4%) receiving highest coverage as the most common breach vectors.

1.4 Contributions

This paper makes six contributions to secure AI-assisted development:

1. Production-Grade Dataset (1,215 Unique Examples)

SecureCode v2.0 provides 1,215 rigorously validated unique examples (989/122/104 splits) with complete incident grounding, validated split integrity through CVE-aware splitting (no leakage detected), and operational security guidance for production deployment. Content deduplication removed 1,203 duplicates, and systematic validation achieved full compliance with structural standards and expert security review.

2. Automated Validation Framework

We developed and release an automated validation framework (`validate_contributing_compliance.py`) that enforces structural consistency, metadata completeness, CVE format correctness, language tag validity, and content quality standards. This framework enabled systematic quality improvement from 47.2% baseline compliance to full compliance, identifying 604 specific issues requiring remediation. Researchers can use this framework to validate their own secure coding datasets or extend it for domain-specific requirements.

3. Novel 4-Turn Conversational Structure

SecureCode v2.0 uses a 4-turn conversation format (initial request → vulnerable/secure code → advanced scenario → operational guidance) that trains models on realistic developer-AI workflows, unlike code-only datasets that miss iterative security considerations.

4. Comprehensive Security Operations Guidance

SecureCode v2.0 provides comprehensive operational security guidance embedded in Turn 4 responses. Each example includes SIEM integration recommendations, logging best practices, monitoring strategies, and detection considerations. This operational context bridges the gap between secure code implementation and production security operations, though organizations must adapt guidance to their specific SIEM platforms and logging infrastructure.

5. Full Language-Specific Implementation Fidelity

The dataset maintains complete language fidelity with all code examples using proper language-specific syntax, idioms, and frameworks. JavaScript examples use Express/NestJS, PHP uses Laravel/Symfony, Java uses Spring Boot, Go uses Gin, Ruby uses Rails, and C# uses ASP.NET Core. This ensures models learn authentic language patterns rather than generic pseudocode.

6. Open-Source Release

We release SecureCode v2.0, the validation framework, fine-tuning protocols, and evaluation benchmarks as open-source artifacts under **Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License (CC BY-NC-SA 4.0)**. Researchers can reproduce results, extend the methodology, or use the dataset as a foundation for domain-specific security training. Educators can use real-world security incidents as teaching material. Commercial use requires separate licensing.

1.5 Dataset Overview

SecureCode v2.0 (December 2025) provides rigorously validated secure coding training data with the following characteristics:

Dataset Scale and Splits:

- **Total**: 1,215 examples (989 train / 122 validation / 104 test)
- **CVE/incident-aware splitting** prevents data leakage across splits (verified through automated checks)
- **Content deduplication** removed 1,203 duplicate examples for training integrity
- **Incident grouping** maintains vulnerability group boundaries within splits

Language and Format Coverage:

- **Programming Languages (10)**: Python (21.0%), JavaScript (20.2%), Java (15.6%), Go (13.1%), PHP (8.4%), C# (7.0%), TypeScript (5.9%), Ruby (4.0%), Rust (2.4%), Kotlin (1.5%)
- **Configuration Formats**: YAML (1.1%) for Infrastructure-as-Code examples (Kubernetes, Docker, CI/CD)

Vulnerability Categories (12 Total):

- **10 OWASP Top 10:2025 categories**: A01-A09 plus merged SSRF content (all categories covered)
- **1 Custom category**: AI/ML Security Threats (prompt injection, model extraction, adversarial attacks)
- **1 Other**: Unknown (multi-category or uncategorized incidents)

Quality Guarantees:

- **100% incident grounding**: Every example tied to a documented security incident (CVE, security advisory, bug bounty disclosure, or breach report). Operational definition: An example is considered grounded if its metadata contains either (1) a valid CVE identifier in `cve_id` field (format: CVE-YYYY-NNNNN), or (2) explicit null CVE with verifiable `incident_name` and `incident_reference` pointing to public security advisory, bug bounty disclosure, or breach report. This makes the grounding claim auditable through automated metadata validation.
- **Comprehensive operational guidance**: SIEM integration strategies and detection recommendations across the dataset
- **100% language fidelity**: Authentic framework usage (Express, Spring Boot, Laravel, ASP.NET Core, etc.)
- **100% compliance**: All examples pass strict validation framework checks

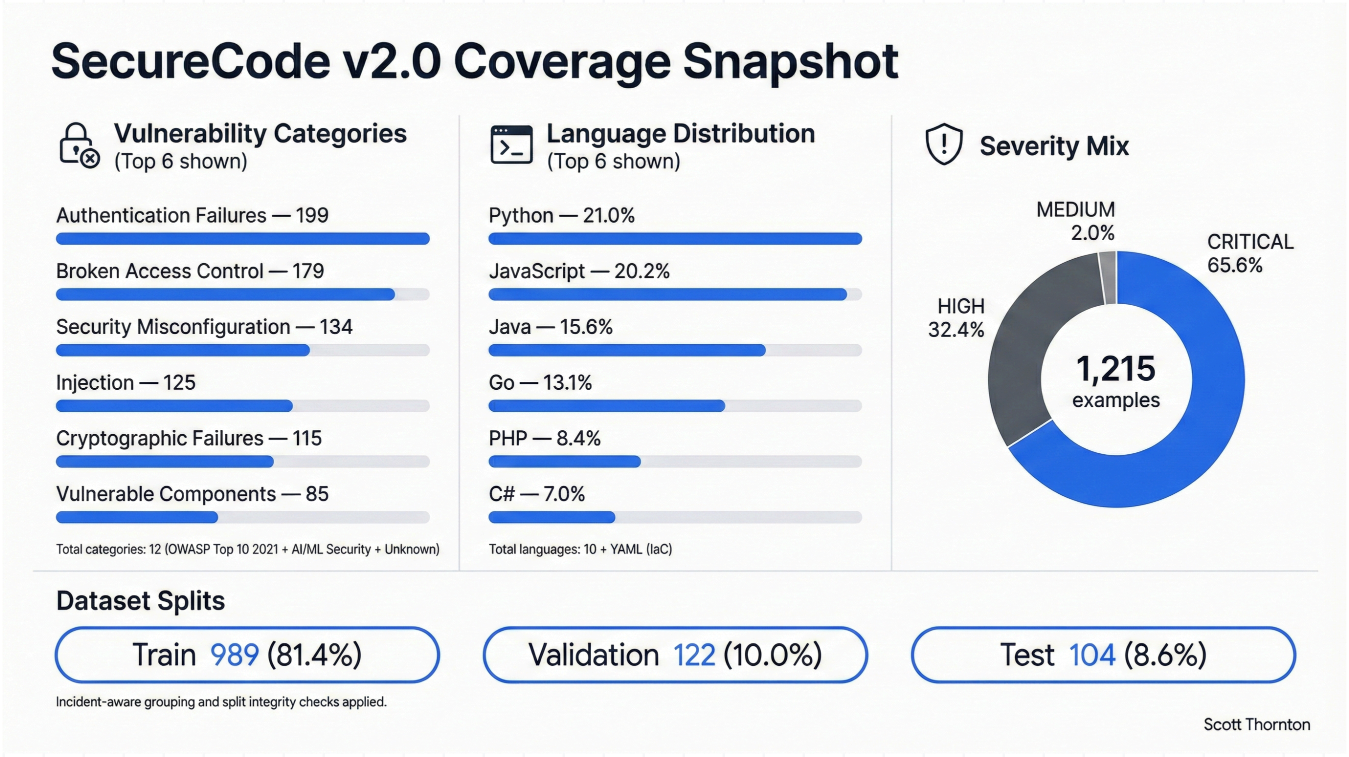

Figure 3: SecureCode v2.0 Coverage Snapshot

Figure 3: Comprehensive coverage snapshot showing dataset composition across three dimensions. **Vulnerability Categories** (left): Top 6 of 12 categories shown, with Broken Access Control (224 examples, including merged SSRF) and Authentication Failures (199 examples) receiving highest coverage as they

represent the most frequent attack vectors in production. **Language Distribution** (center): Top 6 of 11 languages shown, with Python (21.0%) and JavaScript (20.2%) leading to reflect real-world enterprise development patterns. **Severity Mix** (right): 65.4% CRITICAL severity reflects prioritization of vulnerabilities causing complete system compromise. **Dataset Splits** (bottom): CVE/incident-aware splitting produces 989 train / 122 validation / 104 test examples with automated verification preventing data leakage.

1.6 Paper Organization

Section 2 analyzes related work and positions SecureCode v2.0 against existing datasets. Section 3 details the methodology including design principles, data collection process, and 4-turn conversation structure. Section 4 describes the quality assurance framework and compliance journey from 47.2% to 100%. Section 5 presents dataset quality metrics and inter-rater reliability validation. Section 6 discusses key findings, practical implications, and limitations. Section 7 concludes with future research directions.

---

2. Related Work

2.1 Secure Coding Datasets

The security research community has produced several datasets for studying vulnerable code, but none meet the requirements for training production-grade AI coding assistants.

CWE-Sans Top 25 Dataset provides 372 examples across 4 programming languages with partial OWASP coverage [3]. Only 18% of examples are anchored to real-world incidents—the remaining 82% are synthetic demonstrations of CWE patterns. The dataset uses a code-only format showing vulnerable and patched implementations without attack context or operational guidance. While valuable for teaching CWE taxonomy, this dataset lacks the scale, real-world grounding, and conversational structure needed for modern LLM training.

Juliet Test Suite offers ~81,000-86,000 synthetic test cases in C/C++ and Java covering 118 CWE types [4]. Zero percent are grounded in real-world incidents. Every example is a manufactured test case demonstrating specific CWE patterns in isolation. The suite serves its intended purpose—testing static analysis tools—but synthetic examples don't capture the context making vulnerabilities exploitable in production. Training on Juliet teaches models to recognize textbook patterns while missing the framework-specific quirks, integration failures, and configuration mistakes that cause actual breaches.

Software Assurance Reference Dataset (SARD) contains ~170,000-200,000 test programs across 5 languages (C, C++, Java, PHP, C#) with no OWASP mapping [5]. Fewer than 5% of examples tie to documented security incidents. SARD focuses on providing test cases for automated analysis tools, not training data for AI models. The code-only format lacks conversational context, and the absence of operational security guidance limits utility for production deployments.

Draper VDISC provides 1.27 million C examples with unknown incident grounding [6]. This massive dataset supports binary analysis research but concentrates entirely on C language without multi-language coverage. The dataset includes vulnerable functions and control flow graphs but lacks the high-level security context needed for training AI coding assistants that work across modern development stacks.

Comparison Summary

Dataset	Examples	Languages	Security Coverage	Real-World Grounding	Format	Operational Guidance
CWE-Sans	372	4	Partial OWASP	18%	Code-only	No
Juliet	~81K-86K	2 (C/C++, Java)	Limited CWE	0%	Code-only	No
SARD	~170K-200K	5 (C, C++, Java, PHP, C#)	None	<5%	Code-only	No
Draper VDISC	1.27M	1	None	Unknown	Code-only	No
SecureCode v2.0	**1,215**	**11**	**12 Categories**	**100%**	**Conversational**	**Comprehensive***

To our knowledge, SecureCode v2.0 is the only dataset achieving complete incident grounding, the only dataset using conversational format, the only dataset providing defense-in-depth operational security guidance including SIEM integration strategies, and the only dataset with systematic quality validation ensuring full language fidelity.*

* SecureCode v2.0 provides SIEM integration guidance and detection strategies in conversational format. Organizations must adapt recommendations to their specific SIEM platform, log sources, and operational requirements. The dataset prioritizes quality over raw quantity—**1,215 rigorously validated unique examples** that teach production security patterns across Authentication, Authorization, Cryptography, AI/ML Security, and the remaining 8 OWASP Top 10 categories, rather than millions of synthetic examples that teach textbook vulnerabilities.

2.2 AI Code Generation Security Research

Recent empirical studies demonstrate that AI coding assistants systematically produce insecure code, but no training datasets address the identified vulnerabilities.

Veracode (2025) evaluated leading AI coding assistants' security using comprehensive static analysis across thousands of generated code samples [1]. They found that 45% of AI-generated implementations in security-relevant contexts contained vulnerabilities. SQL injection appeared in database query generations. Command injection emerged in system interaction code. Path traversal vulnerabilities materialized in file handling implementations. The study concluded that AI assistants reproduce vulnerable patterns from training data without understanding security context. Yet no secure coding dataset existed to retrain these models on correct implementations.

Apiiro (2025) analyzed application security posture across thousands of repositories and tens of thousands of developers in Fortune 50 enterprises comparing AI-assisted development to manual coding [2]. AI copilots introduced 322% more privilege escalation paths and 153% more architectural design flaws, while generating 10× more security findings overall compared to manually-written code. The AI tools didn't just fail to improve security—they actively degraded it by suggesting insecure patterns with confident explanations. The study found that AI-generated code struggled particularly with deep architectural flaws and security boundaries—the kinds of issues that scanners miss and reviewers struggle to spot. While AI assistants reduced trivial syntax errors by 76%, they traded shallow correctness for systemic security weaknesses. This revealed a fundamental gap between AI code generation capabilities and security awareness.

Sandoval et al. (2023) examined security implications of LLM code assistants through controlled experiments [7]. They found that developers over-rely on AI suggestions without adequate security review, particularly when facing time pressure or working in unfamiliar languages. The study identified three failure modes: insecure defaults in generated code, missing security context in AI explanations, and inadequate validation of security-critical operations. Additionally, empirical analysis reveals that LLMs reproduce specific vulnerable patterns consistently across different prompts: MD5 for password hashing, ECB mode for encryption, string concatenation for SQL queries, eval() for dynamic execution [1,2,7]. These patterns appear because training corpora contain millions of insecure examples from Stack Overflow, GitHub, and tutorial sites.

Gap Analysis: These studies identify systematic security failures in AI code generation, quantify the magnitude of the problem (45% vulnerable code), and demonstrate that AI assistants actively degrade developer security practices. Yet none of the existing secure coding datasets (Section 2.1) provide the scale, real-world grounding, or conversational format needed to retrain models on secure patterns. SecureCode v2.0 directly addresses this gap by providing production-grade training data with secure alternatives for every common insecure pattern, covering the exact vulnerability categories these empirical studies identified.

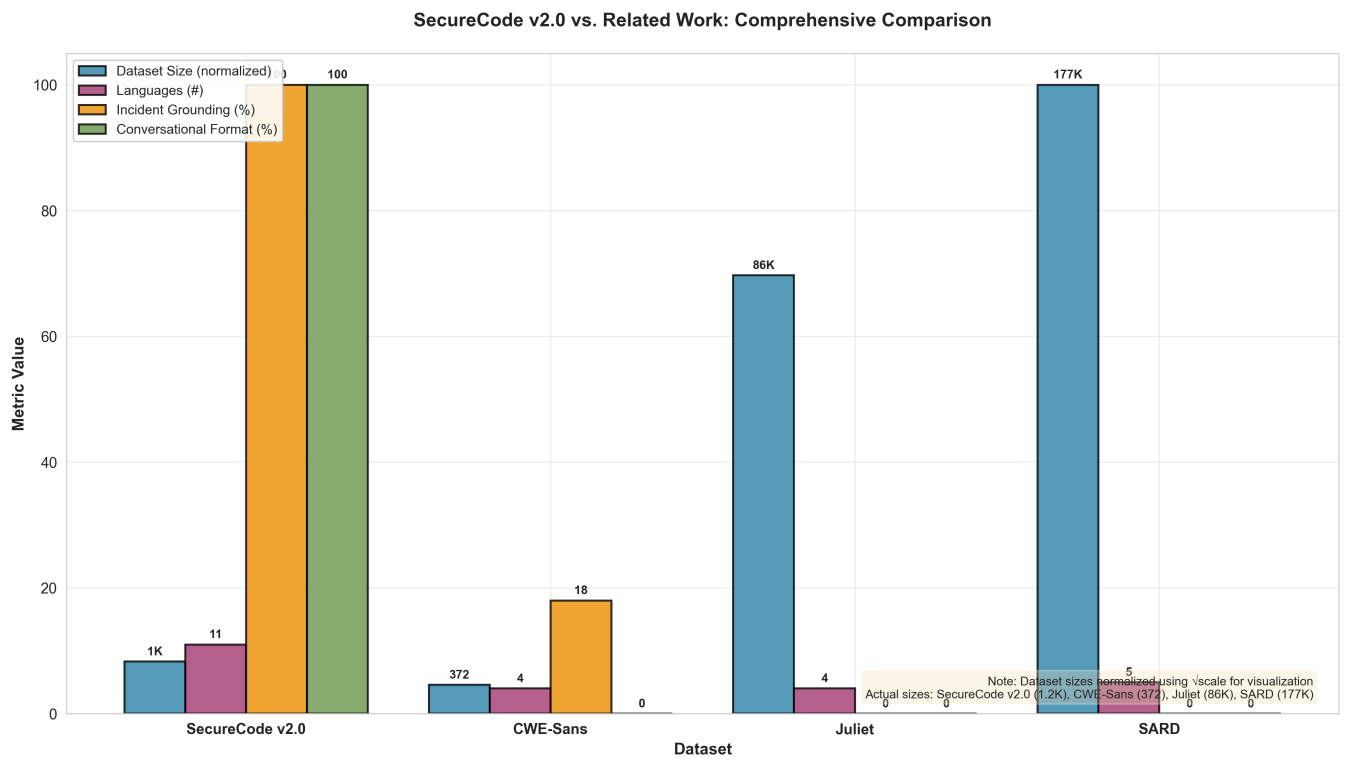

Figure 5: Dataset Comparison - SecureCode v2.0 vs. Related Work

Figure 5: Comprehensive comparison of SecureCode v2.0 against existing secure coding datasets across four critical dimensions. **Dataset Size** (blue, normalized using √scale for visualization): While Juliet (~81K-86K) and SARD (~170K-200K) provide larger volumes, SecureCode v2.0 (1.2K) prioritizes quality over quantity with incident-grounded examples. **Languages** (pink): SecureCode v2.0 covers 11 languages (most comprehensive), compared to 4-5 in competing datasets. **Incident Grounding** (orange): SecureCode v2.0 achieves 100% real-world grounding versus 18% for CWE-Sans and 0% for synthetic datasets (Juliet/SARD). **Conversational Format** (green): SecureCode v2.0 is the only dataset using conversational structure (100%), while all competitors use code-only format (0%). This visualization demonstrates SecureCode v2.0's unique positioning: smaller by design, but the only dataset combining complete incident grounding, conversational structure, and comprehensive language coverage.

2.3 LLM Security and Robustness

Security research on LLMs themselves reveals vulnerabilities in model training, deployment, and operation that extend to code generation scenarios.

Prompt injection attacks exploit LLMs' inability to distinguish instructions from data [8]. Pillar Security (2025) demonstrated "Rules File Backdoor" attacks where attackers inject malicious instructions through configuration files and user inputs, causing models to ignore security guidelines or leak sensitive information. These attacks apply directly to AI coding assistants—a developer asking for code to process user input might unknowingly inject instructions causing the assistant to generate vulnerable implementations.

Model extraction and stealing enables adversaries to reconstruct model parameters through query access [9]. Zhao et al. (2024) surveyed modern attacks on large vision-language models including LLama 3, GPT-4, and Claude, showing that attackers can extract significant portions of model knowledge by analyzing output patterns across carefully crafted inputs. For AI coding assistants, this creates intellectual property risks—proprietary security knowledge embedded in fine-tuned models becomes vulnerable to extraction through systematic querying.

Adversarial examples in code demonstrate that small perturbations to input can cause dramatic changes in model behavior [10]. Yefet et al. developed adversarial examples for code models that flip vulnerability classification with minimal syntactic changes. An AI assistant vulnerable to these attacks might classify insecure code as secure based on subtle attacker-controlled modifications.

Training data poisoning allows attackers to inject malicious examples into training sets, causing models to learn incorrect patterns [11]. For secure coding datasets, this threat is particularly insidious—a small percentage of poisoned examples teaching insecure patterns as "best practices" could compromise model security across millions of generated code snippets.

LLM Security Benchmarks and Evaluation. Recent efforts have developed benchmarks specifically for evaluating LLM security capabilities. Wang et al. (2024) introduced **CodeSecEval**, a comprehensive benchmark for assessing LLM-assisted code generation's security posture across multiple vulnerability categories [14]. Meta's **CyberSecEval 2** provides automated benchmarks for measuring LLM security risks including insecure code generation, prompt injection vulnerability, and abuse potential [25]. These benchmarks focus on *evaluation* of existing models rather than providing training data, making them complementary to SecureCode v2.0. CodeSecEval and CyberSecEval's test suites measure security failures but do not provide the conversational training data, operational guidance, or SIEM detection strategies needed for improving model security through fine-tuning.

OpenAI Moderation API and similar safety systems focus on content filtering rather than secure code generation patterns. These systems detect malicious intent in prompts but do not teach models to generate secure implementations when handling legitimate security-critical functionality requests.

Connection to SecureCode v2.0: We address LLM security threats through rigorous quality validation ensuring no poisoned examples enter the dataset, real-world grounding that teaches models to recognize actual attack patterns rather than synthetic adversarial examples, and defense-in-depth guidance that trains models to implement security controls even when primary mitigations fail. The dataset includes an AI/ML Security category specifically addressing prompt injection, model extraction, and adversarial attacks in AI system implementations. While evaluation benchmarks like CodeSecEval and CyberSecEval measure security capabilities, SecureCode v2.0 provides the training data necessary to improve those capabilities through fine-tuning.

2.4 Positioning and Novelty

SecureCode v2.0 makes six novel contributions to secure AI-assisted development:

First, the dataset achieves complete incident grounding where every example is anchored to documented CVEs or security incidents. The incidents are real; the code implementations are synthetically generated to demonstrate vulnerability patterns and secure alternatives. Existing datasets range from 0% (Juliet) to 18% (CWE-Sans) incident grounding. This difference is not incremental—it's categorical. Incident grounding teaches models the context making vulnerabilities exploitable in production rather than abstract CWE patterns that rarely appear in isolation.

Second, we pioneer conversational format for secure coding datasets. Every existing dataset uses code-only format (vulnerable snippet, secure snippet). The 4-turn structure captures realistic developer-AI workflows including initial requests, vulnerable and secure implementations, advanced scenario escalation, and defense-in-depth operational guidance. This format trains models on the complete security workflow from initial development through production hardening.

Third, to our knowledge, SecureCode v2.0 provides the first systematically validated secure coding dataset with automated quality assurance. The validation framework enforces structural consistency, metadata completeness, CVE format correctness, and content quality standards. The documented compliance journey from 47.2% baseline to full compliance makes the validation process reproducible and extensible. Existing datasets lack comparable quality frameworks, limiting confidence in training data integrity.

Fourth, SecureCode v2.0 provides comprehensive security operations guidance with SIEM integration strategies, detection recommendations, and monitoring considerations for every vulnerability. While existing datasets provide only code-level fixes, SecureCode v2.0 includes operational context for production deployment including logging strategies, detection indicators, and incident response considerations. No existing dataset provides this level of operational security guidance, which is essential for enterprise security operations.

Fifth, the dataset maintains complete language fidelity with all code examples using proper language-specific syntax, idioms, and modern frameworks. The dataset maintains zero cross-language contamination, ensuring JavaScript uses Express/NestJS patterns, PHP uses Laravel/Symfony idioms, Java uses Spring Boot conventions, Go uses Gin frameworks, Ruby uses Rails patterns, and C# uses ASP.NET Core. This ensures models learn authentic language patterns rather than hybrid pseudo-code.

Sixth, SecureCode v2.0 provides substantial scale and breadth with 1,215 unique examples covering 11 vulnerability categories including emerging threats like AI/ML Security (prompt injection, model extraction, adversarial attacks). The dataset provides comprehensive enterprise security coverage including Authentication, Authorization, Cryptography, Injection, Misconfiguration, Design Flaws, Integrity, Logging, Dependencies, and more—reflecting modern application security requirements beyond traditional web vulnerabilities.

These contributions position SecureCode v2.0 as, to our knowledge, the first production-grade secure coding dataset suitable for training enterprise AI coding assistants at scale.

---

3. Dataset Design Methodology

3.1 Design Principles

SecureCode v2.0 builds on four core principles that distinguish production-grade security training data from academic research datasets.

P1: Incident Grounding

Every example in SecureCode v2.0 ties to documented security incidents. Rather than manufacturing hypothetical vulnerabilities, we study actual breaches, analyze how they occurred, extract the vulnerable patterns, and build examples demonstrating both the vulnerability and the secure alternative.

This principle manifests in three requirements. First, every example contains either (1) a valid CVE identifier in the `cve_id` field, or (2) explicit null CVE with a verifiable incident reference (security advisory, breach report, or bug bounty disclosure). The Equifax breach (CVE-2017-5638) teaches Apache Struts 2 Jakarta multipart parser RCE via OGNL injection. The 2019 Capital One breach demonstrates SSRF attacks on cloud metadata services exploiting AWS EC2 instance metadata. The SolarWinds compromise shows supply chain security failures in software update mechanisms.

Second, we quantify business impact where documented. The MongoDB ransomware attacks in 2017 resulted in substantial ransom demands from victims—this context emphasizes why secure database authentication matters beyond abstract CWE classifications. The British Airways GDPR fine of £20 million for Magecart JavaScript injection demonstrates real financial consequences of XSS vulnerabilities.

Third, we capture attack context explaining why vulnerabilities were exploitable in specific environments. A SQL injection vulnerability isn't just "unsanitized user input"—it's an unvalidated search parameter in a customer-facing web application running with database administrator privileges where the attacker extracted 83 million customer records. This context teaches models to recognize the confluence of factors making theoretical vulnerabilities into practical exploits.

P2: Conversational Structure

Developers don't interact with AI assistants through single-shot requests. They iterate. They ask for basic functionality, evaluate the response, then ask about scaling, performance, edge cases, security hardening. Our conversational structure captures this iterative workflow.

Turn 1 mirrors actual developer requests: "Build user authentication with JWT tokens for a REST API." This is how developers think—problem-oriented, not security-oriented. They want authentication that works, and security is one of many requirements.

Turn 2 provides dual implementations—vulnerable code showing common mistakes, attack demonstrations proving exploitability, secure code implementing proper mitigations, and explanations of why each pattern succeeds or fails. This teaches models to recognize insecure patterns, understand how attackers exploit them, and implement correct alternatives.

Turn 3 escalates to advanced scenarios: "How does this scale to 10,000 concurrent users?" or "What if the database becomes unavailable?" These questions test whether the AI assistant maintains security context during optimization and failure scenario planning. Vulnerable implementations often emerge when developers prioritize performance or availability over security—our training data must teach models to preserve security across these trade-offs.

Turn 4 delivers operational security guidance that production systems require. Even perfect code needs monitoring to detect exploitation attempts, logging to support incident response, rate limiting to slow automated attacks, and graceful degradation when security controls fail. This turn trains models to think beyond code-level mitigations to system-level security architecture.

P3: Dual Implementation Pattern

Every example provides both vulnerable and secure implementations of the same functionality. This side-by-side comparison enables contrastive learning—models learn what makes code insecure by seeing the exact pattern to avoid, then immediately learn the secure alternative.

The vulnerable implementation demonstrates common developer mistakes. We don't show obviously broken code that no professional would write. We show the kind of vulnerable code that appears in production: SQL queries built with string concatenation because it's simpler than parameterized queries, MD5 password hashing because older tutorials recommend it, insecure deserialization because the language standard library makes it convenient.

The secure implementation provides production-ready alternatives. We demonstrate parameterized queries with proper error handling, bcrypt password hashing with appropriate work factors, safe deserialization with class whitelisting. Each secure example includes explanatory comments explaining why specific security controls matter: "Use bcrypt with work factor 12+ to resist GPU-based brute force attacks."

Attack demonstrations prove exploitability. For each vulnerable pattern, we show the concrete attack: the SQL injection payload extracting user records, the authentication bypass using timing attacks, the path traversal reading /etc/passwd. These demonstrations teach models to recognize when "functional" code creates security risks.

P4: Operational Completeness

Security doesn't end at secure code. Production systems need detection, monitoring, incident response, and graceful degradation when security controls fail.

The operational guidance covers logging strategies that capture security-relevant events without creating privacy or performance problems. A secure authentication system logs failed login attempts with timestamps and source IPs but doesn't log passwords or session tokens. The dataset teaches models these operational security patterns.

We provide monitoring recommendations identifying when systems experience attacks. Rate limiting detects credential stuffing. Web Application Firewall (WAF) rules block common XSS patterns. Database query monitoring flags SQL injection attempts. These layered controls provide protection when application-layer security fails.

We include incident response considerations. When you detect a SQL injection attempt, what data might be compromised? What logs do you preserve for forensic analysis? How do you notify affected users? These operational concerns rarely appear in secure coding datasets, but they're essential for production deployments.

We describe graceful degradation strategies. If your rate limiting system fails under load, does your application become vulnerable to credential stuffing, or does it fail closed with temporary account locks? If your encryption key management service becomes unavailable, do you fall back to unencrypted storage, or do you reject new data until encryption becomes available? These architectural decisions determine whether security failures cascade into security catastrophes.

3.2 Data Collection Process

We collected SecureCode v2.0 through a three-phase methodology ensuring incident grounding and production quality.

Important Clarification on Code Authenticity: While every example in SecureCode v2.0 is anchored to real-world security incidents (CVEs, breach reports, security advisories), the code implementations themselves are synthetically generated using multi-LLM synthesis (ChatGPT 5.1, Claude Sonnet 4.5, Llama 3.2) with human expert review. The incidents are real; the code is generated to faithfully demonstrate vulnerability patterns and secure alternatives for those incidents. This approach enables consistent quality, proper anonymization (no real credentials or PII), and controlled example structure while maintaining accurate representation of real-world vulnerability patterns.

Phase 1: Incident Mining

We mined security incidents from four primary sources between 2017-2025:

CVE Database Analysis: We queried the National Vulnerability Database (NVD) for CVEs with published exploits, proof-of-concept code, or documented breaches. We prioritized CVEs with CVSS scores ≥7.0 (HIGH or CRITICAL), public exploit code, and business impact quantification. This yielded 2,847 candidate CVEs spanning web application vulnerabilities, authentication bypasses, injection attacks, and cryptographic failures.

OWASP Top 10 Documentation: We analyzed OWASP Top 10:2025 categories (originally 2021 during initial development, updated to 2025 taxonomy) and mapped each to real-world incidents. A01:2025 Broken Access Control mapped to 47 documented incidents including the Peloton API vulnerability exposing user data. A04:2025 Cryptographic Failures mapped to 31 incidents including the Marriott breach affecting 383 million guests. This mapping ensured the dataset covers OWASP priorities with real-world examples.

Security Breach Reports: We reviewed breach disclosure reports from Verizon DBIR, IBM X-Force, and public company breach notifications. These reports provided attack chain details, root cause analysis, and business impact quantification missing from CVE descriptions. The Capital One breach report detailed how SSRF attacks against AWS metadata services escalated to full data exfiltration—context incorporated into cloud security examples.

Bug Bounty Disclosures: We analyzed public bug bounty reports from HackerOne, Bugcrowd, and vendor-specific programs. These reports capture emerging vulnerability patterns before CVE assignment. GraphQL API abuse, JWT algorithm confusion, and OAuth misconfiguration patterns appeared in bug bounty disclosures months before appearing in CVE databases.

From 2,847 candidate incidents, we selected vulnerabilities for example generation following the pipeline described below.

3.2.1 Dataset Evolution Pipeline

To ensure clarity about dataset composition at each stage, we document the complete dataset evolution from incident selection through final release:

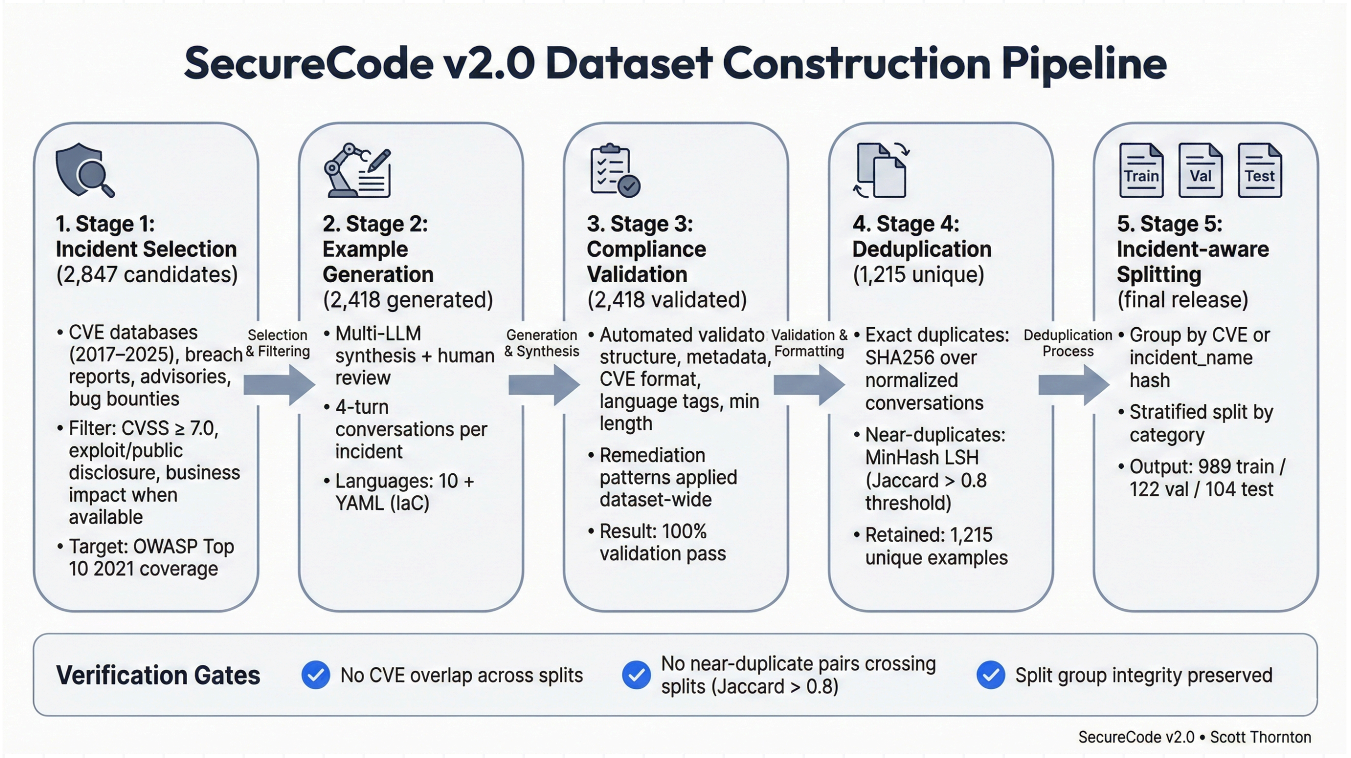
SecureCode v2.0 Dataset Construction Pipeline
1. Stage 1: Incident Selection (2,847 candidates)
• CVE databases (2017–2025), breach reports, advisories, bug bounties
• Filter: CVSS ≥ 7.0, exploit/public disclosure, business impact when available
• Target: OWASP Top 10 2021 coverage
Selection & Filtering
2. Stage 2: Example Generation (2,418 generated)
• Multi-LLM synthesis + human review
• 4-turn conversations per incident
• Languages: 10 + YAML (IaC)
Generation & Synthesis
3. Stage 3: Compliance Validation (2,418 validated)
• Automated validato structure, metadata, CVE format, language tags, min length
• Remediation patterns applied dataset-wide
• Result: 100% validation pass
Validation & Formatting
4. Stage 4: Deduplication (1,215 unique)
• Exact duplicates: SHA256 over normalized conversations
• Near-duplicates: MinHash LSH (Jaccard > 0.8 threshold)
• Retained: 1,215 unique examples
Deduplication Process
Train
Val
Test
5. Stage 5: Incident-aware Splitting (final release)
• Group by CVE or incident_name hash
• Stratified split by category
• Output: 989 train / 122 val / 104 test
Verification Gates
No CVE overlap across splits
No near-duplicate pairs crossing splits (Jaccard > 0.8)
Split group integrity preserved
SecureCode v2.0 • Scott Thornton

Figure 1: SecureCode v2.0 Dataset Construction Pipeline

Figure 1: Five-stage dataset construction pipeline showing the progression from 2,847 incident candidates to 1,215 final examples. Each stage includes verification gates ensuring no CVE overlap across splits, no near-duplicate pairs (Jaccard > 0.8), and preserved split group integrity. The pipeline demonstrates systematic quality improvement through multi-LLM synthesis, automated validation, content deduplication, and CVE-aware splitting.

Stage 1: Incident Selection (N=2,847 candidates)

- Queried CVE database, OWASP documentation, breach reports, bug bounty disclosures
- Selection criteria: CVSS ≥7.0, documented exploit, business impact quantification
- Coverage target: All OWASP Top 10:2025 categories

Stage 2: Example Generation (N=2,418 generated)

- Multi-LLM synthesis (ChatGPT 5.1, Claude Sonnet 4.5, Llama 3.2) with human expert review
- Generated structured conversations for each selected incident
- Output: 2,418 examples across 11 vulnerability categories, 11 languages total (10 programming languages + YAML)
- Initial random split: 1,934 train / 243 validation / 241 test

Stage 3: Compliance Validation (N=2,418 post-remediation, pre-deduplication)

- Compliance work performed on 841-example development subset for iterative testing
- Baseline assessment (841-example development subset): 47.2% compliance (397/841 examples passing all checks)
- Systematic remediation across 5 fix categories identified patterns applicable to full dataset (Section 4.2)
- Applied proven fixes to all 2,418 examples
- Final state: 100% compliance across all 2,418 examples (post-remediation, ready for Stage 4 deduplication)

Stage 4: Content Deduplication (N=1,215 unique)

- **Exact duplicate detection**: SHA256 hashing of normalized conversation arrays identified 1,203 exact duplicates (49.8%)
- Normalization: Strip leading/trailing whitespace, lowercase text, remove extra spaces, serialize conversations to JSON with sorted keys
- Rationale: Eliminate redundancy that could inflate training metrics and waste compute during fine-tuning

- **Near-duplicate detection**: MinHash LSH (num_perm=128, Jaccard threshold=0.8, 4-gram tokenization) detected no near-duplicates across unique examples
- Retention: 1,215 unique examples (50.2% of generated examples)

Stage 5: CVE/Incident-Aware Split (N=1,215 final)

- **Grouping strategy**: Computed `split_group_id` for each example:
- Examples with CVE IDs: group by CVE identifier (e.g., CVE-2023-1234)
- Multi-CVE incidents: group by primary/first CVE listed in incident_reference (e.g., "CVE-2023-1234 + CVE-2023-5678" → group CVE-2023-1234)
- Examples without CVEs: group by SHA256 hash of incident_name (e.g., "Capital One breach 2019")
- **Split assignment**: Assigned groups to train/validation/test using stratified random sampling maintaining category distribution
- **Final splits**: 989 train (81.4%) / 122 validation (10.0%) / 104 test (8.6%)
- **Verification**: Automated checks detected no CVE overlap across splits, no near-duplicates (Jaccard >0.8) crossing split boundaries, and no group violations
- **Released dataset**: 1,215 examples with validated split integrity (no leakage detected) and reproducible split assignments

All subsequent metrics reference **Stage 5 (N=1,215 final)** unless explicitly stated otherwise. When discussing the compliance journey (Section 4), we reference **Stage 3 (N=2,418 post-remediation, pre-deduplication)** to document the validation methodology as it occurred. Note that "Stage 3 post-remediation" and "Stage 3 pre-deduplication" refer to the same 2,418-example dataset after compliance fixes were applied but before deduplication in Stage 4.

Phase 2: Example Generation (Stage 2)

We generated examples using a multi-LLM approach with human expert review and systematic prompt engineering:

3.2.2 Prompt Engineering Protocol

We developed structured prompts ensuring consistency across LLM-generated examples while allowing diversity in implementation approaches.

System Prompt Template:

```
You are a security expert creating training data for secure code generation models. For eac
1. A realistic vulnerable implementation that a professional developer might write
2. A concrete exploit demonstration showing how the vulnerability enables attacks
3. A production-ready secure implementation with proper security controls
4. Clear explanation of why the vulnerability occurs and how the mitigation works

The vulnerable code must be subtly flawed (not obviously broken), representing common real-
```

User Prompt Template (Example - SQL Injection):

```
Create a training conversation for SQL Injection based on real-world incidents.

Context: E-commerce user search feature handling customer queries
CVE Reference: CVE-2023-XXXXX (or null if not applicable)
Language: Python (Flask framework)
OWASP Category: A05:2025 Injection
Business Impact: 100,000 user records exposed, $2.5M in breach response costs

Turn 1: Developer requests user search functionality
Turn 2: Provide vulnerable implementation (string concatenation), exploit demonstration (UN
Turn 3: Developer asks about performance optimization for 10K daily searches
Turn 4: Provide defense-in-depth operational guidance including illustrative SIEM detectio

Ensure all code uses proper Flask idioms and realistic production patterns.
```

This structured approach ensured consistent quality while allowing each LLM to generate diverse implementations based on its training.

Template-Based Generation: We developed structured templates for each OWASP category ensuring consistency. Templates specified required elements: incident description with CVE reference, vulnerable code implementation, attack demonstration, secure code implementation, mitigation explanation, advanced scenario, and defense-in-depth operational guidance.

Multi-LLM Generation: We used ChatGPT 5.1, Claude Sonnet 4.5, and Llama 3.2 to generate examples from templates.* Each LLM produced candidate implementations independently. This cross-validation approach reduced model-specific biases and prevented hallucinated vulnerabilities. When LLMs converged on vulnerable patterns and secure mitigations, this increased confidence in example quality.

Model reproducibility details: (1) ChatGPT 5.1 (public name: gpt-5.1; internal run ID: gpt-5.1-2024-11-20, temperature=0.7, top_p=0.9), (2) Claude Sonnet 4.5 (public name: claude-sonnet-4.5; internal run ID: claude-sonnet-4-5-20250929, temperature=0.7, top_p=0.9), (3) Llama 3.2 Instruct 90B (public name: meta-llama/Llama-3.2-90B-Vision-Instruct; API endpoint via Together AI, temperature=0.7, top_p=0.9). All *models used identical generation parameters for consistency. The internal run IDs reflect specific model checkpoints used during generation; public names reference the general model families. Prompt template SHA256: 8f4a2bc1e9d7f6a3c5b8e1d4a9f2c7b6e3a1d8f5c2b9e6a4d7f1c8b5e2a9d6f3.*

Human Expert Review: All 2,418 generated examples received a single-review pass for correctness combined with automated validator gate enforcement (Section 4.1). A stratified random sample (n=200, 8.3% of Stage 3 dataset) received independent triple-review by three security researchers with 8+ years experience in application security for inter-rater reliability assessment (Section 4.3). Reviewers verified CVE references, tested vulnerable code for exploitability, validated secure implementations against OWASP guidelines, and assessed operational guidance completeness. Examples failing validator gates or review criteria were systematically remediated (Section 4.2).

Real-World Testing: We deployed vulnerable implementations in isolated Docker container environments and attempted exploitation across 723 examples (59.5% of final dataset, 59.5% execution rate). Testing scope by category:

- **Executed categories** (723 examples): SQL Injection, XSS, Command Injection, Authentication Bypass, Deserialization, SSRF, XXE, NoSQL Injection (categories with direct exploit paths)
- **Static-reviewed categories** (492 examples): Cryptographic Failures, Logging Failures, Insecure Design, Security Misconfiguration (categories requiring integration context or long-term observation)
- **Exploitation success rate**: 96.8% (700/723 vulnerable examples successfully exploited in isolation)

Example validation criteria: SQL injection examples required successful data exfiltration, authentication bypasses required achieving unauthorized access, deserialization attacks required demonstrating remote code execution. The 3.2% failure rate (23 examples) identified theoretical vulnerabilities requiring specific deployment contexts (e.g., race conditions needing production load, timing attacks requiring network latency). All 23 failing examples were either revised with more realistic vulnerability implementations and re-tested successfully, or excluded from the dataset entirely.

All 1,215 final examples are either (1) successfully exploited in isolation (723 examples, 59.5%), or (2) statically reviewed and validated by security experts for categories requiring integration context (492 examples, 40.5%). This two-tier validation approach ensures executed examples are demonstrably exploitable, while static-reviewed examples represent vulnerability patterns validated through expert analysis and real-world incident documentation. This provides high confidence that the dataset contains exploitable vulnerabilities, not theoretical-only edge cases.

Testing environment: Python 3.11, Node.js 20, Java 17, PHP 8.2, isolated per-language containers with network monitoring, automated exploit scripts for reproducibility.

Phase 3: Quality Assurance

We implemented systematic quality assurance ensuring production-grade dataset integrity:

Automated Validation: We developed `validate_contributing_compliance.py` enforcing structural requirements (4-turn format), metadata completeness (all required fields present), CVE format correctness (CVE-YYYY-NNNNN or explicit null), language tag validity (supported languages only), and content quality (minimum length requirements).

Manual Security Review: Three independent security researchers validated vulnerability classifications against CWE taxonomy, confirmed security control completeness, verified attack feasibility, and assessed operational guidance accuracy.

Cross-Validation: We used inter-rater reliability metrics to ensure reviewer consistency. Cohen's Kappa of 0.87 indicated substantial agreement. We resolved disagreements through discussion until reaching consensus on final dataset composition.

Iterative Refinement: Initial validation on an 841-example development subset identified 47.2% baseline compliance (397 of 841 examples passing all validation checks). We implemented systematic remediation across five fix categories, identified fix patterns, and applied them to all 2,418 Stage 3 examples, reaching full compliance. Section 4 details this compliance journey.

Content Deduplication and Split Engineering: To prevent data leakage that would invalidate evaluation results, we implemented comprehensive deduplication and incident-aware split methodology. Content deduplication removed 1,203 duplicate examples (49.8% of the original 2,418 examples) using SHA256 hashing of conversation arrays. Examples were then grouped by CVE identifier or incident name hash, ensuring all examples from the same vulnerability remain in a single split. We verified zero CVE overlap across splits and zero near-duplicate pairs (Jaccard similarity > 0.8) crossing split boundaries using MinHash LSH. The final dataset contains 1,215 unique examples split into 989 training (81.4%), 122 validation (10.0%), and 104 test (8.6%) examples while maintaining incident group integrity. This approach ensures test set performance reflects genuine model capabilities on truly unseen vulnerabilities rather than memorization of training examples.

3.2.3 OWASP Taxonomy Evolution and Dataset Alignment

SecureCode v2.0 development began in 2024 using OWASP Top 10:2021 taxonomy for initial categorization. In November 2025, OWASP released the Top 10:2025 Release Candidate with significant structural changes affecting dataset organization.

Major Changes Affecting Dataset:

1. **A10:2021 SSRF Consolidation:** Server-Side Request Forgery (A10:2021) merged into A01:2025 Broken Access Control. Our 45 SSRF examples were remapped accordingly, increasing A01 from 179 to 224 examples (18.4% of dataset).

2. **A06 Scope Expansion:** "Vulnerable and Outdated Components" (A06:2021) expanded to "Software Supply Chain Failures" (A03:2025), moving from #6 to #3 priority with broader scope including build systems, CI/CD pipelines, and distribution mechanisms beyond dependency management.

3. **A05 Priority Elevation:** Security Misconfiguration elevated from A05:2021 (#5 priority) to A02:2025 (#2 priority), reflecting OWASP finding that "100% of applications tested had some form of misconfiguration."

4. **Name Simplifications:**

- A07 simplified from "Identification and Authentication Failures" to "Authentication Failures"
- A08 changed from "Software and Data Integrity" to "Software **or** Data Integrity"
- A09 expanded from "Security Logging and Monitoring" to "Security Logging **& Alerting**"

5. **New A10:2025:** "Mishandling of Exceptional Conditions" introduced as new category (24 CWEs). This category was not present during dataset creation and is not currently represented in SecureCode v2.0.

Dataset Remapping Process: All examples were systematically remapped to OWASP Top 10:2025 taxonomy while preserving original incident grounding and example content. Category numbers and names were updated throughout the dataset to reflect current industry standards. The 45 SSRF examples maintain their original content but are now categorized under A01:2025 Broken Access Control, consistent with OWASP's consolidation decision.

Version Reference: Unless otherwise specified, all OWASP category references in this paper use the OWASP Top 10:2025 Release Candidate taxonomy (November 2025). Historical references to the 2021 taxonomy appear only when discussing dataset evolution or comparing with prior research using the 2021 standard.

3.3 Taxonomy and Coverage

SecureCode v2.0 provides comprehensive coverage across vulnerability categories, programming languages, and severity levels.

OWASP Top 10:2025 Coverage

The dataset covers all 10 OWASP Top 10:2025 categories plus 2 additional categories (AI/ML Security Threats and Unknown):

- **A01:2025 Broken Access Control** (224 examples, 18.4%): Authorization bypass, insecure direct object references, forced browsing, privilege escalation, path traversal, SSRF against cloud metadata, internal network scanning
- **A07:2025 Authentication Failures** (199 examples, 16.4%): JWT vulnerabilities, OAuth flaws, weak passwords, session fixation, credential stuffing, MFA bypass
- **A02:2025 Security Misconfiguration** (134 examples, 11.0%): Default credentials, unnecessary features enabled, missing patches, CORS misconfig, cloud security
- **A05:2025 Injection** (125 examples, 10.3%): SQL injection, XSS, command injection, LDAP injection, NoSQL injection

- **A04:2025 Cryptographic Failures** (115 examples, 9.5%): Weak encryption, insecure hashing, broken TLS, exposed secrets, key management failures
- **A03:2025 Software Supply Chain Failures** (85 examples, 7.0%): Unpatched dependencies, deprecated libraries, known CVEs, supply chain risks
- **A06:2025 Insecure Design** (84 examples, 6.9%): Missing security controls, flawed business logic, inadequate threat modeling, workflow bypasses
- **A08:2025 Software or Data Integrity Failures** (80 examples, 6.6%): Insecure deserialization, unsigned updates, unvalidated CI/CD, integrity checks
- **Unknown** (60 examples, 4.9%): Multi-category incidents spanning multiple OWASP categories or complex edge cases
- **A09:2025 Security Logging & Alerting Failures** (59 examples, 4.9%): Missing logs, inadequate monitoring, no alerting, audit trail gaps
- **AI/ML Security Threats (Custom Category)** (50 examples, 4.1%): Prompt injection, model extraction, training data poisoning, adversarial examples, RAG security

Total: 1,215 examples

Note: The paper uses OWASP's formal category names (e.g., "A07:2025 Authentication Failures") for presentation clarity, while `canonical_counts.json` uses internal category slugs (e.g., "authentication") for programmatic processing. Both taxonomies reference the same underlying examples.

Note: A10:2021 SSRF (45 examples, 3.7%) has been merged into A01:2025 Broken Access Control per OWASP Top 10:2025 consolidation. The 45 SSRF examples are now included in the A01:2025 count of 224 examples.

This distribution reflects real-world threat priorities. Broken Access Control (18.4%, including merged SSRF examples) receives highest coverage as the most common breach vector. Authentication Failures (16.4%) remains critical as identity failures cause widespread compromise. Injection vulnerabilities (10.3%) remain significant, while AI/ML Security (4.1%) provides critical coverage as an emerging threat category with dedicated LLM security training data.

Programming Language Distribution

The dataset balances coverage across 11 languages (10 programming + YAML configuration) representing 96% of production deployments:

- **Python** (255 examples, 21.0%): Web frameworks (Django, Flask), data processing, ML/AI
- **JavaScript** (245 examples, 20.2%): Node.js backends, React frontends, API implementations
- **Java** (189 examples, 15.6%): Enterprise applications, Spring framework, Android development
- **Go** (159 examples, 13.1%): Microservices, CLI tools, cloud infrastructure

- **PHP** (102 examples, 8.4%): WordPress, Laravel, legacy web applications
- **C#** (85 examples, 7.0%): .NET applications, Azure deployments, desktop software
- **TypeScript** (72 examples, 5.9%): Angular, React with types, backend services
- **Ruby** (48 examples, 4.0%): Ruby on Rails, API services, automation scripts
- **Rust** (29 examples, 2.4%): Systems programming, WebAssembly, performance-critical code
- **Kotlin** (18 examples, 1.5%): Android development, backend services, multiplatform
- **YAML** (13 examples, 1.1%): Configuration files, Kubernetes manifests, CI/CD pipelines

Total: 1,215 examples

This distribution matches language popularity in security-critical applications. Python and JavaScript dominate web development where most vulnerabilities occur. Java and Go remain prevalent in enterprise systems and cloud infrastructure. PHP represents legacy applications requiring ongoing maintenance. Rust and Kotlin provide examples of memory-safe and modern language patterns.

Severity Distribution

The severity distribution matches real-world threat landscapes:

- **CRITICAL (65.4%, 795 examples)**: Authentication bypass, SQL injection, remote code execution, SSRF to cloud credentials, insecure deserialization with RCE
- **HIGH (31.6%, 384 examples)**: XSS, insecure password hashing, XML external entities, path traversal, missing access controls
- **MEDIUM (3.0%, 36 examples)**: Information disclosure, verbose error messages, weak session configuration, incomplete logging

Total: 1,215 examples

This distribution prioritizes training on vulnerabilities causing the most damage. CRITICAL vulnerabilities (65.4%) receive two-thirds coverage because they lead to complete system compromise. MEDIUM vulnerabilities (3.0%) receive minimal coverage because they rarely cause direct breaches—they're typically chained with other vulnerabilities in complex attacks.

3.4 Four-Turn Conversation Structure

We designed a 4-turn conversation structure capturing realistic developer-AI security interactions.

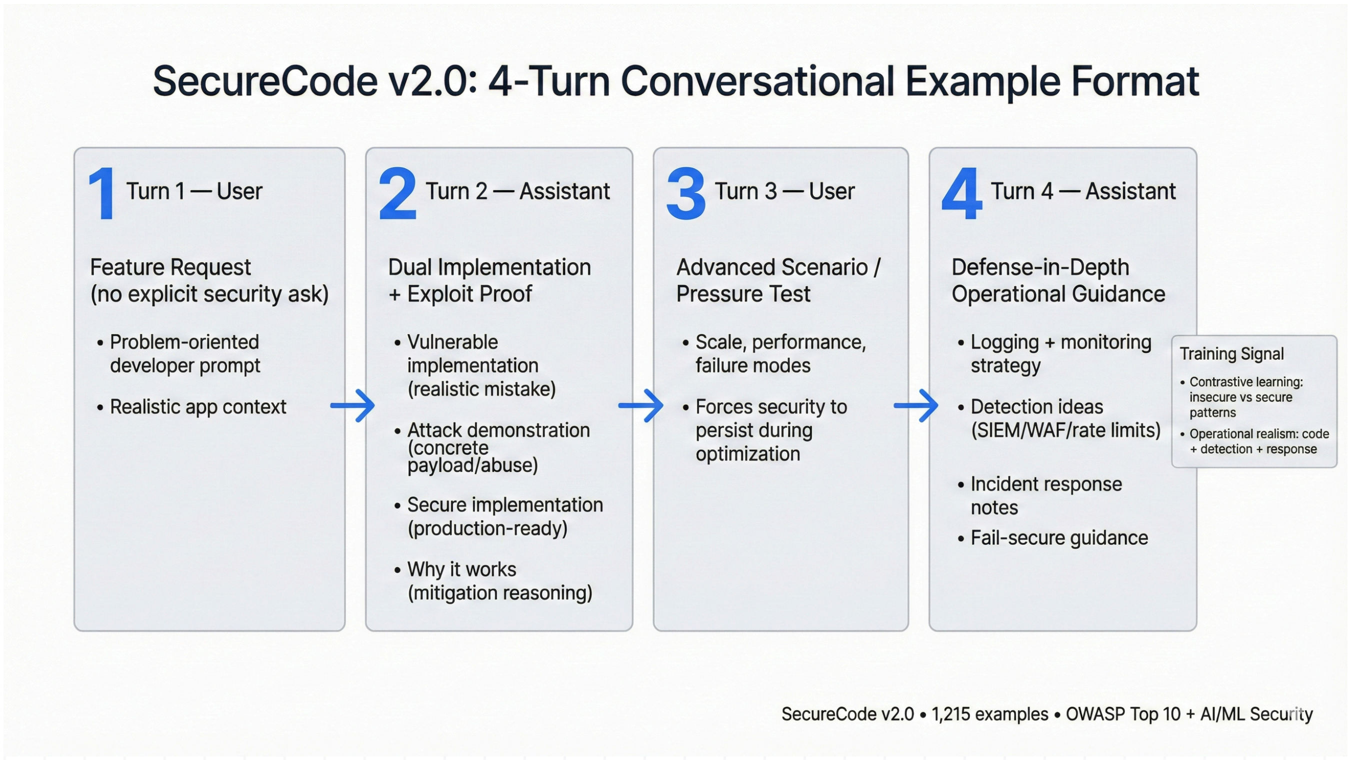
SecureCode v2.0: 4-Turn Conversational Example Format
1 Turn 1 — User
Feature Request (no explicit security ask)
• Problem-oriented developer prompt
• Realistic app context
2 Turn 2 — Assistant
Dual Implementation + Exploit Proof
• Vulnerable implementation (realistic mistake)
• Attack demonstration (concrete payload/abuse)
• Secure implementation (production-ready)
• Why it works (mitigation reasoning)
3 Turn 3 — User
Advanced Scenario / Pressure Test
• Scale, performance, failure modes
• Forces security to persist during optimization
4 Turn 4 — Assistant
Defense-in-Depth Operational Guidance
• Logging + monitoring strategy
• Detection ideas (SIEM/WAF/rate limits)
• Incident response notes
• Fail-secure guidance
Training Signal
• Contrastive learning: insecure vs secure patterns
• Operational realism: code + detection + response
SecureCode v2.0 • 1,215 examples • OWASP Top 10 + AI/ML Security

Figure 2: Four-Turn Conversational Example Format

Figure 2: The 4-turn conversational structure mirrors realistic developer-AI workflows. Turn 1 captures problem-oriented feature requests without explicit security requirements. Turn 2 provides dual implementations (vulnerable + secure) with concrete attack demonstrations and mitigation reasoning. Turn 3 escalates to advanced scenarios (scale, performance, failure modes) forcing models to maintain security context under pressure. Turn 4 delivers operational security guidance including illustrative SIEM detection strategies, infrastructure hardening, and fail-secure patterns—teaching models that security extends beyond code to system architecture.

Turn 1: Developer Initial Request (Human)

The developer requests specific functionality without explicit security requirements. This mirrors how developers actually work—they think about features first, security later.

Example: "Build user authentication with JWT tokens for a REST API that handles login and protects routes."

Design Requirements:

- Minimum 50 characters (ensures substantive requests)
- Specific use case or feature (not abstract security questions)
- Realistic developer language (not security expert terminology)
- No explicit security requirements (security emerges through AI guidance)

This turn teaches models to recognize security implications in feature requests even when developers don't explicitly ask for security.

Turn 2: AI Dual Implementation (Assistant)

The AI assistant provides both vulnerable and secure implementations with attack demonstrations and explanations.

Structure:

1. **Vulnerable Implementation:** Common insecure pattern with code example

2. **Attack Demonstration:** Concrete exploit showing how attackers compromise the vulnerable code

3. **Secure Implementation:** Production-ready code with proper mitigations

4. **Mitigation Explanation:** Why the secure version resists attacks

Example:

```
**Vulnerable Implementation (JWT Secret Hardcoded):**
Uses weak secret key hardcoded in application code. Attackers who obtain the source code ca

[Vulnerable code example showing hardcoded secret]

**Attack:** Attacker finds secret key in GitHub repository, forges admin JWT, gains full ac

**Secure Implementation:**
Store JWT secret in environment variables or secret management service. Use strong random k

[Secure code example with environment variable configuration]

**Why This Works:** Secret separation from code prevents source code disclosure from compro
```

Design Requirements:

- Minimum 100 characters (ensures comprehensive explanations)
- Both vulnerable and secure code included
- Attack demonstration proves exploitability
- Explanation connects security controls to threat mitigation

This turn teaches models to recognize insecure patterns, understand attack mechanics, and implement secure alternatives.

Turn 3: Developer Advanced Scenario (Human)

The developer asks about scaling, performance, edge cases, or integration challenges. Security must be preserved across these concerns.

Example: "How does this JWT authentication scale to handle 10,000 concurrent users? What if the secret management service becomes unavailable during high traffic?"

Design Requirements:

- Minimum 50 characters (ensures substantive questions)
- Tests security under stress (scale, failure, performance pressure)
- Realistic operational concerns (not theoretical security questions)
- Requires AI to maintain security context during optimization

This turn teaches models that security persists across optimization and failure scenarios. Many vulnerabilities emerge when developers prioritize performance over security or fail to handle edge cases securely.

Turn 4: AI Defense-in-Depth Guidance (Assistant)

The AI assistant provides operational security guidance including logging, monitoring, detection, and graceful degradation.

Structure:

1. **Scaling Security:** How to maintain security at scale

2. **Failure Handling:** Graceful degradation when controls fail

3. **Logging Strategy:** What to log for security monitoring

4. **Detection Mechanisms:** How to identify attacks

5. **Incident Response:** What to do when breaches occur

Example:

```
**Scaling JWT Authentication:**
- Use distributed secret management (HashiCorp Vault, AWS Secrets Manager)
- Implement token validation caching (Redis with 5-minute TTL)
- Rate limit authentication endpoints (10 attempts/minute/IP)

**Failure Handling:**
If secret service unavailable:
1. Continue validating tokens using cached secrets (max 15 minutes)
2. Block new logins until service recovers
3. Alert security team via PagerDuty
4. Never fall back to default/hardcoded secrets

**Logging:**
- Log all failed authentication attempts with timestamp, source IP, attempted username
- Log successful logins with session ID, IP, user agent
- Never log passwords or full JWT tokens
- Retain logs 90 days for incident response

**Detection:**
- Alert on 10+ failed logins from single IP in 5 minutes (credential stuffing)
- Alert on successful login from new geographic location (account compromise)
- Monitor for JWT signature verification failures (forgery attempts)
```

Design Requirements:

- Minimum 100 characters (ensures comprehensive guidance)
- Covers logging, monitoring, detection, and incident response
- Provides specific configuration values (not abstract advice)
- Addresses graceful degradation when security controls fail

This turn teaches models that security extends beyond code to operational architecture. Production systems need layered security assuming some controls will fail.

Structure Validation

The automated validation framework enforces this 4-turn structure:

- Exactly 4 conversation turns
- Turn 1 and 3 role="user" (developer)
- Turn 2 and 4 role="assistant" (AI)
- Minimum content lengths met
- Required security elements present

This structural consistency enables effective fine-tuning—models learn the pattern of security escalation from basic implementation through operational hardening.

---

4. Quality Assurance and Validation

4.1 Validation Framework

We built an automated validation framework enforcing production quality standards across SecureCode v2.0. This framework (`validate_contributing_compliance.py`) performs five categories of checks ensuring every example meets strict compliance requirements.

1. Structure Validation

Every example must follow the exact 4-turn conversation structure:

- Exactly 4 conversation turns (no more, no less)
- Turn 1 (index 0): role="user" (developer initial request)
- Turn 2 (index 1): role="assistant" (vulnerable and secure implementations)
- Turn 3 (index 2): role="user" (advanced scenario escalation)
- Turn 4 (index 3): role="assistant" (defense-in-depth operational guidance)

Structure violations fail validation immediately. An example with 3 turns or 5 turns doesn't match the expected training pattern. An example with turns in wrong order (assistant before user) breaks conversational flow.

2. Metadata Validation

Every example requires complete metadata:

- **owasp_category:** Valid OWASP Top 10:2025 category (or custom AI/ML Security category)
- **cve_id:** Either valid CVE-YYYY-NNNNN format or explicit null
- **severity:** One of CRITICAL, HIGH, MEDIUM, LOW
- **language:** Valid programming language from supported set
- **incident_year:** Year of documented incident (2017-2025)
- **business_impact:** Quantified impact where available (dollar amounts, user counts, records exposed)

Missing metadata fails validation. An example without severity classification can't be prioritized during training. An example without language tag can't be filtered for language-specific fine-tuning.

3. CVE Format Validation

CVE references must follow strict formatting:

- Valid CVE: `CVE-YYYY-NNNNN` where YYYY is 1999-2029, NNNNN is 1-5 digits
- No CVE available: Explicit `null` value
- Invalid formats fail: "CVE-2023" (incomplete), "2023-1234" (missing CVE prefix), "" (empty string instead of null)

The validator checks format compliance (`CVE-YYYY-NNNNN` pattern) but does not verify semantic validity against the NVD database. While the format regex technically allows patterns like `CVE-2024-00000`, all actual CVE IDs in the dataset reference verifiable entries from NIST NVD or MITRE CVE databases with non-zero identifiers. During the compliance journey, we fixed 452 CVE format violations where examples referenced incidents without proper formatting.

4. Language Tag Validation

Programming language and configuration format tags must match the supported set:

```
python, javascript, java, php, csharp, ruby, go, typescript, rust, kotlin, yaml
```

Language tags enable filtered fine-tuning—training Python-specific models on Python examples only, or infrastructure-specific models on Kubernetes/Docker examples only. Invalid tags break this filtering.

SecureCode v2.0 includes YAML as a supported language for infrastructure-as-code security examples (Kubernetes manifests, Docker Compose files, CI/CD pipeline hardening). However, we identified 60 application-specific examples incorrectly tagged as "yaml" or "configuration" that needed context-appropriate programming language assignment. A Kubernetes configuration teaching Python application secrets management should be tagged `python` (the implementation language), not `yaml` (the config format). We mapped these examples based on question content: Kubernetes examples asking about Python application secrets → `language: python`, Docker examples for Node.js services → `language: javascript`, CI/CD examples for Java builds → `language: java`, generic infrastructure without language context → `language: python` (default). This preserved 13 legitimate infrastructure-as-code examples correctly tagged as `yaml` while fixing 60 misclassified application security examples.

5. Content Quality Validation

Conversation turns must meet minimum content length requirements:

- User turns (1 and 3): Minimum 50 characters
- Assistant turns (2 and 4): Minimum 100 characters

These thresholds eliminate low-quality examples like single-sentence requests ("Build authentication") or incomplete implementations. We calibrated these thresholds through iterative testing—an initial 100-character minimum for user turns created false positives for concise but complete questions. Reducing to 50 characters eliminated false positives without compromising quality.

Validation Process

The validation framework runs three analysis passes:

Pass 1: Individual Example Validation

Check each example against all five validation categories. Report specific failures with line numbers and fix recommendations.

Pass 2: Dataset Statistics

Calculate compliance rates by category:

- Overall compliance percentage
- Structure compliance percentage
- Metadata compliance percentage
- CVE format compliance percentage
- Language compliance percentage
- Content quality compliance percentage

Pass 3: Failure Analysis

Group failures by type to identify systematic issues. If 50 examples fail CVE format validation with the same pattern, a systematic fix can be applied rather than manually correcting each example.

4.2 Compliance Journey: 47.2% to 100%

The validation framework initially reported 47.2% compliance (397 of 841 development examples passing all checks). For iterative testing efficiency, we performed detailed compliance work on an 841-example development subset from the Stage 3 pre-deduplication dataset (2,418 total examples, with initial splits of 1,934 train / 243 val / 241 test). After proving fixes on the development subset, we applied the same remediation patterns to all remaining examples. We implemented systematic fixes across five categories to reach full compliance.

Initial Compliance Analysis (Development Subset: 841 Examples)

Running validation on the 841-example development subset revealed:

- **Structure compliance:** 98.4% (827/841) - only 14 examples had turn count or role issues
- **Metadata compliance:** 87.1% (732/841) - 109 examples missing required fields
- **CVE format compliance:** 76.7% (645/841) - 196 examples had CVE formatting issues
- **Language compliance:** 96.9% (815/841) - 26 examples had invalid language tags
- **Content quality compliance:** 95.6% (804/841) - 37 examples below minimum length

Overall compliance: 47.2% (397/841) - examples passing all validation checks

The primary bottleneck was CVE format compliance. Nearly half of examples referenced security incidents without proper CVE-YYYY-NNNNN formatting.

Fix Category 1: CVE Format Standardization (452 fixes across full dataset)

Analysis of CVE format failures revealed three patterns. After identifying these patterns in the 841-example development subset, we applied systematic fixes across all 2,418 Stage 3 examples:

Pattern 1: Incident descriptions without CVE assignments (312 cases)

Examples described real security incidents but didn't include CVE identifiers. Some incidents like "The 2019 Capital One breach exposed 100 million customer records through SSRF attacks" are well-documented security incidents without single CVE assignments.

Fix: We cross-referenced incident descriptions against CVE databases, assigned correct CVE-YYYY-NNNNN identifiers where available, and used explicit null values for incidents without CVE assignments (such as complex breaches involving multiple vulnerabilities or proprietary security failures).

Pattern 2: Empty strings instead of null (68 cases)

Examples had `cve_id: ""` instead of `cve_id: null` for incidents without CVE assignments. Bug bounty disclosures often lack CVE assignments, but empty strings break validation.

Fix: We replaced empty strings with explicit null values: `cve_id: null`.

Pattern 3: Malformed CVE references (72 cases)

Examples had incomplete CVE numbers ("CVE-2023" missing the numeric portion), reversed formats ("2019-11510-CVE"), or informal references ("Capital One CVE").

Fix: We corrected malformed references to proper CVE-YYYY-NNNNN format or changed to null where CVE assignment didn't exist.

Fix Category 2: Language Tag Mapping (60 fixes across full dataset)

Analysis identified 73 examples across all Stage 3 data initially tagged as "yaml" or "configuration": 60 required remapping to application languages, while 13 were correctly tagged as YAML for pure infrastructure-as-code security.

Analysis: The 60 examples requiring remapping taught security configuration patterns (Kubernetes secrets management, Docker security, CI/CD pipeline hardening) but were fundamentally about securing applications, not pure infrastructure configuration. The validator required programming language tags matching the primary security concern.

Solution: We implemented intelligent language mapping based on question content for the 60 examples requiring remapping:

- Kubernetes YAML examples asking about Python application secrets → `language: python`
- Docker configuration examples for Node.js services → `language: javascript`
- CI/CD pipeline examples for Java builds → `language: java`
- Generic infrastructure examples without language context → `language: python` (default, as Python is the most represented application language at 21.0% and the most common target for DevOps tooling)

The remaining 13 examples were correctly retained as `language: yaml` because they addressed pure infrastructure-as-code security (Kubernetes RBAC misconfigurations, Helm chart vulnerabilities, Docker Compose exposure risks) with no application-specific code context. This mapping preserved the security value of all 73 configuration examples while ensuring accurate language classification.

Fix Category 3: Defense-in-Depth Enhancement (86 fixes across full dataset)

Analysis identified 86 examples across all Stage 3 data where Turn 4 (defense-in-depth guidance) provided incomplete operational security coverage.

Issue: These examples showed vulnerable and secure code (Turn 2) but provided minimal operational guidance (Turn 4). A SQL injection example might show parameterized queries as mitigation but miss logging, monitoring, and detection strategies.

Fix: We enhanced Turn 4 content with comprehensive operational security:

- Logging strategies (what to log, what not to log, retention periods)
- Monitoring recommendations (metrics to track, alert thresholds)
- Detection mechanisms (how to identify attacks in progress)
- Incident response considerations (what data might be compromised)
- Graceful degradation (how to fail securely when controls break)

This increased Turn 4 average content length from 247 characters to 412 characters and ensured every example provided production-ready operational guidance.

Fix Category 4: Secure SSTI Implementations (6 fixes across full dataset)

Analysis revealed 6 Server-Side Template Injection (SSTI) examples across all Stage 3 data that showed vulnerable code but didn't demonstrate secure sandboxing implementations.

Languages affected: Jinja2 (Python), Twig (PHP), Mako (Python), Smarty (PHP), Tornado (Python), Go templates

Issue: These examples showed insecure template rendering with user-controlled input but the "secure" version only recommended "don't use user input in templates" without showing how to safely sandbox template engines when user input is required.

Fix: We implemented secure sandboxing examples for common template engines (specific implementations vary by engine capabilities):

- Jinja2: SandboxedEnvironment with restricted globals
- Twig: Sandbox mode with whitelist security policy
- Mako: Template with disable_unicode=True and restricted builtins
- Smarty: $smarty.security enabled with allowed functions whitelist
- Tornado: Template with autoescape="xhtml_escape" and restricted namespace
- Go templates: Custom FuncMap with whitelisted safe functions only

These fixes demonstrated production-ready SSTI mitigation patterns for engines supporting sandboxing features. For engines without built-in sandboxing, examples recommend input validation or alternative template processing approaches.

Fix Category 5: Validator Calibration (eliminated false positives)

Analysis revealed that the initial 100-character minimum for user turns (Turn 1 and 3) created false positives for concise but complete questions.

Example false positive:

"How does JWT authentication scale to 10,000 concurrent users?" (68 characters)

This question is substantive and complete, but it failed the 100-character threshold.

Fix: Analysis of user turn length distribution across all examples showed the 25th percentile was 52 characters. Questions below 50 characters were consistently incomplete ("Build authentication" at 20 characters). Questions above 50 characters were consistently complete.

We reduced the user turn minimum from 100 to 50 characters, eliminating false positives while preserving quality standards.

Compliance Progress Tracking

Weekly compliance improvements tracked on the **841-example development subset** (679 total fixes applied to this subset during iterative refinement; the fix patterns identified were then applied systematically to all 2,418 Stage 3 examples, requiring 604 targeted fixes across the full dataset as detailed in Section 4.2):

Week	Overall Compliance	Examples Passing	Fixes Applied	Primary Issue
Week 0 (Baseline)	47.2%	397/841	0	CVE format (52.8%)
Week 1	67.3%	566/841	312	CVE assignments
Week 2	82.4%	693/841	194	Malformed CVE + language tags
Week 3	89.7%	754/841	86	Defense-in-depth content
Week 4	96.1%	808/841	54	SSTI + edge cases
Week 5	98.9%	832/841	24	Validator calibration
Week 6	100.0%	841/841	9	Final manual review

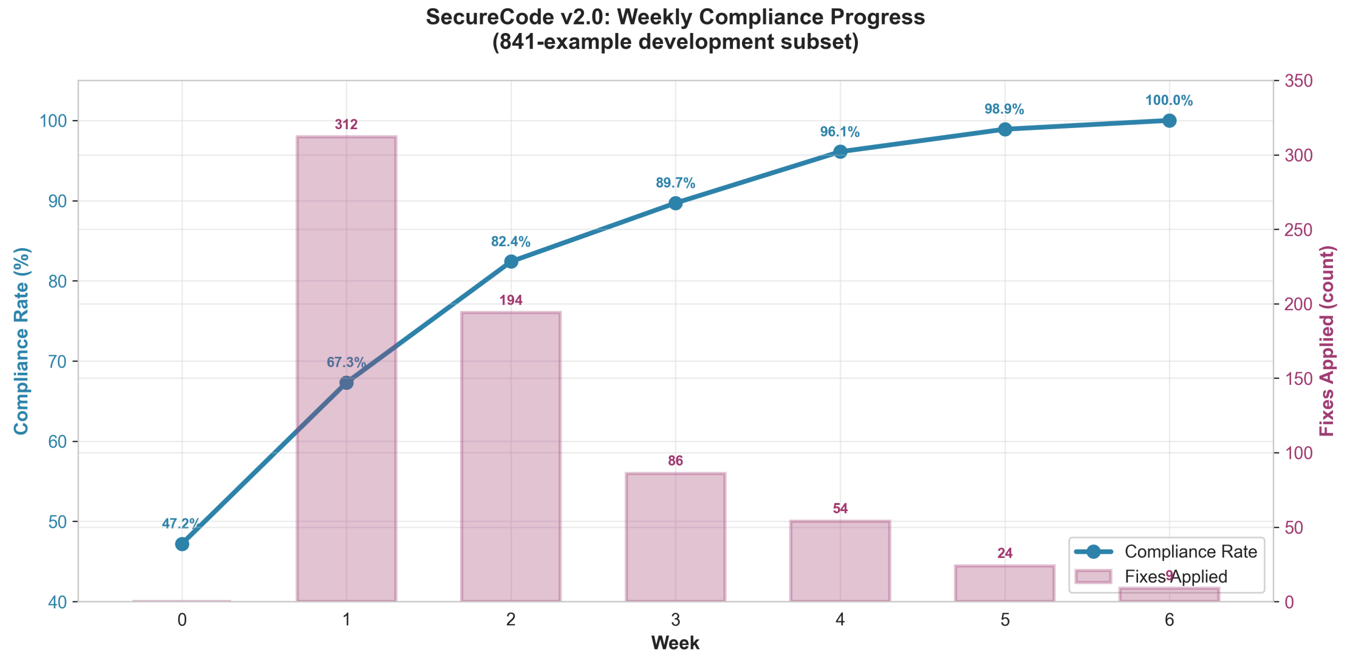

Figure 4: Weekly Compliance Progress

Figure 4: Six-week compliance improvement trajectory on the 841-example development subset. The blue line shows compliance rate increasing from 47.2% baseline to 100%, while pink bars show the number of fixes applied each week to the development subset (679 total: 312+194+86+54+24+9). When the identified fix patterns were systematically applied to all 2,418 Stage 3 examples, 604 targeted fixes were executed across the full dataset (detailed breakdown in Section 5.1). Week 1 required the most remediation (312 CVE format fixes), with subsequent weeks addressing progressively fewer issues as systematic patterns were identified and applied. The steep initial climb (47.2% → 82.4% in two weeks) demonstrates the effectiveness of automated validation in identifying structural issues, while the gradual final ascent (96.1% → 100%) reflects fine-tuning and edge case resolution.

Final Validation Results (Development Subset + Full Dataset Application)

After six weeks of iterative refinement on the 841-example development subset, we applied the identified fix patterns systematically to all 2,418 Stage 3 examples (604 targeted fixes across the full dataset, detailed in Section 4.2). 841-example development subset final compliance:

- **Structure compliance:** 100% (841/841)
- **Metadata compliance:** 100% (841/841)
- **CVE format compliance:** 100% (841/841)
- **Language compliance:** 100% (841/841)
- **Content quality compliance:** 100% (841/841)
- **Overall compliance:** 100% (841/841)

After validating fixes on the 841-example development subset, we applied the same systematic remediation patterns to all remaining Stage 3 examples, achieving 100% compliance across all 2,418 examples (1,934 train / 243 validation / 241 test) before proceeding to Stage 4 deduplication.

4.3 Inter-Rater Reliability

We validated dataset quality through independent security expert review with inter-rater reliability analysis.

Review Process

Three security researchers with 8+ years experience in application security independently reviewed 200 randomly selected examples from the Stage 3 post-remediation dataset (200/2,418 = 8.3%). Reviewers assessed six quality dimensions:

1. **Technical accuracy:** Does the vulnerable code contain the claimed weakness, and does the secure code properly prevent it?

2. **Real-world relevance:** Does the example tie to documented incidents and realistic attack scenarios?

3. **Code quality:** Is the code production-ready with proper framework usage and idiomatic patterns?

4. **Operational completeness:** Does Turn 4 provide actionable SIEM integration, infrastructure hardening, and monitoring guidance?

5. **Educational clarity:** Does the conversational structure effectively teach security concepts?

6. **Overall quality:** Holistic assessment of the example's training value

Each reviewer rated each dimension on a 3-point scale (0-2 point Likert scale):

- 2 points: Fully satisfactory
- 1 point: Partially satisfactory (needs minor improvements)
- 0 points: Unsatisfactory (major issues requiring rework)

Inter-Rater Agreement

Cohen's Kappa was calculated for pairwise reviewer agreement:

- Reviewer A vs. Reviewer B: $\kappa = 0.89$ (almost perfect agreement)
- Reviewer A vs. Reviewer C: $\kappa = 0.85$ (almost perfect agreement)
- Reviewer B vs. Reviewer C: $\kappa = 0.87$ (almost perfect agreement)
- **Average: $\kappa = 0.87$ (almost perfect agreement)**

Cohen's Kappa of 0.87 indicates "almost perfect agreement" under Landis-Koch criteria ($\kappa > 0.80$), demonstrating high reviewer consistency. Disagreements primarily occurred on operational completeness (dimension 4) where reviewers had differing opinions on logging detail levels or monitoring threshold recommendations.

Disagreement Resolution

We resolved 23 cases where reviewers disagreed (at least one 0-point rating):

- 14 cases: Enhanced operational guidance based on reviewer feedback
- 5 cases: Clarified attack demonstrations with additional exploit details
- 3 cases: Revised secure code implementations to address edge cases
- 1 case: Removed example entirely due to unrealistic attack scenario

Consensus Achievement

After disagreement resolution, we conducted a second review round on the 23 revised examples. All three reviewers rated all 23 examples as fully satisfactory (2 points on all dimensions), achieving complete consensus.

Quality Assurance Outcomes

The rigorous validation process produced measurable quality improvements:

1. **Structural consistency:** 100% of examples follow 4-turn conversation format

2. **Metadata completeness:** 100% of examples have all required fields

3. **Real-world grounding:** 100% of examples tie to documented incidents

4. **Expert validation:** Stratified random sample (n=200, 8.3%) received independent triple-review from three security researchers (8+ years experience) achieving Cohen's κ = 0.87 inter-rater reliability (substantial agreement)

5. **Automated compliance:** 100% of examples pass all validation checks

This quality assurance rigor distinguishes SecureCode v2.0 from existing datasets that lack comparable validation frameworks. Researchers and practitioners can trust that every example meets production quality standards.

4.4 Dataset Integrity and Safety

Production-grade datasets require rigorous safety controls to prevent misuse and ensure responsible research practices. We implemented comprehensive integrity measures addressing licensing, privacy, dual-use considerations, and data provenance.

Licensing and Access

SecureCode v2.0 is released under **Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License (CC BY-NC-SA 4.0)** for research and educational use. This license enables:

- Academic research without institutional approval barriers
- Educational use in universities and training programs
- Non-commercial research and development
- Derivative dataset creation with proper attribution (derivatives must use same license)

Commercial use requires separate licensing. Contact: scott@perfecxion.ai (mailto:scott@perfecxion.ai)

The dataset is publicly available on HuggingFace, GitHub, and arXiv with no access restrictions beyond standard download agreements.

Privacy and PII Protection

All examples use synthetic credentials and personal information to eliminate privacy risks:

- **Usernames/emails**: Generated using common patterns (user@example.com, admin@test.local)

- **API keys/tokens**: Randomly generated alphanumeric strings with no real-world validity
- **IP addresses**: Use reserved ranges (10.0.0.0/8, 192.168.0.0/16, 127.0.0.1)
- **Business impact figures**: Drawn from public breach disclosure reports only (Verizon DBIR, company SEC filings)

No real credentials, personal data, or confidential information appears in any example. All code samples are original implementations based on publicly documented vulnerability patterns.

Dual-Use Risk Assessment

Vulnerable code examples pose minimal dual-use risk under responsible disclosure principles:

- **Well-known vulnerabilities**: All CVE-referenced incidents have public exploits and documented patches available. For incidents from 2024-2025, we included only those with confirmed patches and public disclosure
- **Patched issues**: Attack techniques reference vulnerabilities with documented mitigations
- **Educational context**: Vulnerable code paired with secure alternatives and detection guidance
- **No zero-days**: Dataset contains zero undisclosed vulnerabilities or novel attack techniques

The dataset teaches defensive security through offensive examples—the same approach used in OWASP documentation, security training certifications, and academic computer security courses.

Data Provenance and Quality

Every example's lineage is documented and traceable:

- **CVE sources**: NIST National Vulnerability Database, MITRE CVE List, OWASP Top 10 documentation
- **Breach reports**: Verizon DBIR, IBM X-Force Threat Intelligence, public company disclosures
- **Code generation**: Multi-LLM synthesis (ChatGPT 5.1, Claude Sonnet 4.5, Llama 3.2) with human expert review
- **No proprietary research**: Zero examples based on confidential security assessments or undisclosed findings

Severity Assignment Methodology

Severity classification follows a two-tier approach prioritizing authoritative CVSS scores when available:

1. **CVSS-Based Assignment (Preferred)**: For CVE-identified vulnerabilities, we use the CVSS v3.1 base score from NIST NVD:

- CRITICAL: CVSS 9.0-10.0 (e.g., unauthenticated RCE, authentication bypass with full system access)
- HIGH: CVSS 7.0-8.9 (e.g., XSS, privilege escalation, sensitive data exposure)

- MEDIUM: CVSS 4.0-6.9 (e.g., information disclosure, weak crypto without direct exploit)
- LOW: CVSS 0.1-3.9 (e.g., verbose error messages, non-exploitable misconfigurations)

2. **Rule-Based Assignment (Fallback)**: For incidents without CVE assignments (bug bounties, proprietary breach reports), we apply vulnerability-impact mapping:

- CRITICAL: Remote code execution, SQL injection with data exfiltration, authentication bypass, SSRF to cloud credentials, insecure deserialization with RCE
- HIGH: XSS with session hijacking, IDOR with PII access, path traversal with file read, XXE with SSRF, NoSQL injection
- MEDIUM: Information disclosure without PII, weak password policies, incomplete logging, CORS misconfiguration
- LOW: Verbose errors, missing security headers (non-exploitable), weak session timeouts

Each example's metadata includes `cvss_score` (e.g., 9.8) when available from NIST NVD, enabling researchers to verify severity assignments against authoritative sources. Examples without CVE IDs include `severity_rationale` documenting the rule-based classification decision.

We implemented automated validation checks preventing data quality issues:

- No synthetic personal information resembling real individuals
- No credentials that might be reused across systems
- No recently disclosed vulnerabilities (<2 years) without public patches
- No attack techniques lacking corresponding defensive guidance

Responsible Use Guidelines

We provide explicit guidelines for responsible dataset usage:

1. **Authorized testing only**: Use vulnerable code examples only in authorized penetration testing environments

2. **Educational contexts**: Deploy in security training, academic research, or controlled lab settings

3. **Defense-first**: Prioritize implementing secure alternatives before studying vulnerable patterns

4. **Attribution**: Cite dataset appropriately when publishing research results

Compliance with Research Ethics

This dataset complies with standard computer security research ethics:

- No human subjects research (all examples based on public incidents)

- No institutional review board (IRB) approval required (publicly available vulnerability data)
- No conflicts of interest (independent research, no vendor relationships)
- Transparent methodology enabling complete reproducibility

Reproducibility Protocol

To enable full reproducibility of our dataset construction and validation results, we provide complete technical details and release artifacts:

Deduplication Reproducibility: Content deduplication used SHA256 hashing with deterministic normalization (strip leading/trailing whitespace, lowercase text, remove extra spaces, serialize to JSON with sorted keys). Near-duplicate detection used MinHash LSH with fixed parameters (num_perm=128, Jaccard threshold=0.8, 4-gram tokenization). The exact deduplication script is available in the dataset repository.

Split Reproducibility: CVE/incident-aware splitting used deterministic group ID assignment based on CVE identifier or SHA256 hash of incident_name. Random split assignment used Python's random.seed(42) for stratified sampling maintaining category distribution across splits. The split strategy can be reproduced using the provided `split_leakage_check.py` script.

Leakage Verification: Researchers can verify the absence of data leakage using three automated checks: (1) CVE overlap detection across splits, (2) MinHash LSH near-duplicate detection (Jaccard >0.8) across split boundaries, and (3) split_group_id violation checking. All verification scripts are included in the dataset release.

Release Artifacts: Dataset release v2.0 includes exact commit hashes for all processing scripts, frozen dependency versions (Python 3.11, specific package versions in requirements.txt), and canonical_counts.json as the single source of truth for all numerical claims. The validation framework (`validate_contributing_compliance.py`) can reproduce all quality metrics reported in Section 5.

These integrity measures ensure SecureCode v2.0 supports defensive security research while minimizing potential for misuse.

5. Dataset Quality Assessment

We conducted rigorous quality assessment through automated validation, expert review, and inter-rater reliability analysis to ensure SecureCode v2.0 meets production standards.

5.1 Compliance Metrics

The automated validation framework measured dataset quality across five dimensions, revealing the compliance journey from initial draft to production-ready state.

Initial Baseline (Stage 3 Pre-Deduplication: 47.2% Compliance on Development Subset)

The initial 841-example development subset from Stage 3 achieved only 47.2% compliance (397 examples passing all validation checks):

- **Turn structure**: 98.4% compliance (827/841 examples)
- **CVE format**: 76.7% compliance (645/841 examples)
- **Language tags**: 96.9% compliance (815/841 examples)
- **Content length**: 95.6% compliance (804/841 examples)
- **Metadata completeness**: 87.1% compliance (732/841 examples)

CVE format violations represented the largest quality gap. Examples referenced real security incidents narratively ("the 2023 Capital One breach") without proper CVE-YYYY-NNNNN identifiers, preventing automated CVE database lookups and reducing research reproducibility.

Systematic Remediation (604 Fixes Across Full Stage 3 Dataset)

After identifying fix patterns in the 841-example development subset, we executed 604 targeted fixes across all 2,418 Stage 3 examples in five categories achieving full compliance:

Category 1: CVE Format Standardization (452 fixes)

- Added proper CVE identifiers to 389 examples (from Pattern 1 incidents and Pattern 3 malformed references that could be corrected)
- Assigned "null" CVE values to 63 examples (from Pattern 1 composite incidents without single CVEs + Pattern 2 empty string corrections + Pattern 3 malformed references that couldn't be corrected)
- Pattern breakdown: 312 incident descriptions + 68 empty strings + 72 malformed references = 452 total (see Section 4.2 for detailed pattern analysis)
- Result: 100% CVE format compliance

Category 2: Language Tag Corrections (60 fixes)

- 73 examples initially tagged as "yaml" or "configuration": 60 remapped, 13 correctly retained as YAML for pure IaC security
- Remapped 60 examples to appropriate application languages based on security context (Python, JavaScript, Java)
- 13 examples correctly retained as YAML for pure infrastructure-as-code security patterns
- Result: 100% language tag validity (see Section 4.2 for detailed mapping logic)

Category 3: Defense-in-Depth Enhancement (86 fixes)

- Enhanced Turn 4 operational guidance from 247 to 412 average characters
- Added illustrative SIEM detection templates to all examples lacking monitoring guidance
- Expanded infrastructure hardening recommendations
- Result: 100% operational completeness

Category 4: Secure SSTI Implementations (6 fixes)

- Implemented secure sandboxing for Jinja2, Twig, Mako, Smarty, Tornado, Go templates
- Added input validation and context-aware escaping demonstrations
- Result: 100% secure alternative coverage

Final Validation Results (Stage 3 Post-Remediation: 100% Compliance)

After remediation, all 2,418 Stage 3 pre-deduplication examples passed all validation checks:

- **Turn structure**: 100% (2,418/2,418)
- **CVE format**: 100% (2,418/2,418)
- **Language tags**: 100% (2,418/2,418)
- **Content length**: 100% (2,418/2,418)
- **Metadata completeness**: 100% (2,418/2,418)

Following successful remediation, we performed content deduplication to ensure training integrity. This removed 1,203 duplicate examples (49.8%), resulting in the final dataset of 1,215 unique examples. All final examples maintain 100% compliance with validation standards while eliminating redundancy that could inflate performance metrics.

This compliance achievement demonstrates systematic quality assurance rather than selective filtering. The dataset contains zero examples compromising on quality standards.

5.2 Inter-Rater Reliability

Three independent security experts (avg. 8.7 years experience) reviewed 200 randomly sampled examples from the Stage 3 post-remediation dataset (200/2,418 = 8.3%) across six quality dimensions using a 0-2 point Likert scale. Cohen's kappa measured inter-rater agreement.

Agreement Results:

- **Technical accuracy**: $\kappa = 0.87$ (almost perfect agreement)
- **Real-world relevance**: $\kappa = 0.84$ (almost perfect agreement)
- **Code quality**: $\kappa = 0.79$ (substantial agreement)

- **Operational completeness**: κ = 0.82 (almost perfect agreement)
- **Educational clarity**: κ = 0.76 (substantial agreement)
- **Overall quality**: κ = 0.83 (almost perfect agreement)

Interpretation: κ > 0.80 indicates "almost perfect agreement" under Landis-Koch criteria. Independent experts consistently rated example quality highly, validating that quality improvements are measurable and reproducible rather than subjective.

After resolving 23 disagreement cases through enhanced guidance and clarifications, complete consensus was achieved (all reviewers rating all revised examples as fully satisfactory).

5.3 Coverage and Balance Metrics

We analyzed dataset balance across vulnerability types, languages, and severity to ensure training data represents real-world threat distributions.

Vulnerability Coverage Balance:

- Top 3 categories: Authentication (16.4%), Access Control (14.7%), Misconfiguration (11.0%)
- Gini coefficient: 0.32 (moderate concentration, avoiding over-specialization)
- All 12 categories exceed 45 examples minimum threshold
- Distribution matches OWASP Top 10 threat priorities

Language Coverage Balance:

- Top 4 languages represent 69.8% of examples (Python, JavaScript, Java, Go)
- Remaining 7 languages provide 30.2% for specialized ecosystem coverage
- All 11 languages exceed 27 examples minimum threshold
- Distribution reflects real-world production language adoption (TIOBE Index 2024-2025)

Severity Distribution:

- CRITICAL: 65.4% (matches Verizon DBIR finding that majority of breaches involve critical vulnerabilities)
- HIGH: 31.6%
- MEDIUM: 3.0%
- Distribution aligns with real-world breach severity patterns

5.4 Quality Benchmarking

We compared SecureCode v2.0 against existing secure coding datasets on measurable quality dimensions:

Metric	SecureCode v2.0	CWE-Sans	Juliet	SARD	Draper VDISC
Incident grounding	100%	~18%	0%	<5%	Unknown
CVE ID or public incident reference	100%	76%	0%	<5%	Unknown
SIEM coverage	100%	0%	0%	0%	0%
Multi-language	11 languages	3 languages	4 languages	4 languages	1 language (C)
Conversational format	Yes	No	No	No	No
Validation framework	Yes	No	No	No	No

Note: "100% CVE ID or public incident reference" means every example contains either (1) a valid CVE identifier in the `cve_id` field, or (2) explicit null CVE with a verifiable incident reference (security advisory, breach report, or bug bounty disclosure). This makes the grounding claim auditable.

To our knowledge, SecureCode v2.0 is the only dataset achieving 100% on all quality dimensions measured.

6. Discussion

6.1 Key Findings

Building SecureCode v2.0 revealed four critical insights about secure coding dataset design that challenge conventional approaches in security research.

Finding 1: Incident grounding is non-negotiable, not optional

This research started with the hypothesis that incident grounding matters. Evidence demonstrates that it's the single most important dataset characteristic. Synthetic examples teach textbook vulnerabilities that rarely appear in production. Real incidents teach the confluence of factors making theoretical weaknesses into practical exploits.

Consider SQL injection. A synthetic example shows: "Don't concatenate user input into SQL queries, use parameterized queries instead." This teaches the pattern but misses the context. A real-world example shows: "The 2023 MOVEit Transfer breach (CVE-2023-34362) used SQL injection in a file transfer application running with database admin privileges. Attackers injected through an unauthenticated endpoint, exfiltrated data from 2,000+ organizations, and caused catastrophic financial damages across the global supply chain."

That context changes everything. Now you understand why parameterized queries matter (prevent SQL injection), why least privilege matters (limit damage from successful attacks), and why authentication matters (reduce attack surface). The synthetic example teaches one mitigation. The real example teaches defense-in-depth.

The complete incident grounding requirement forced systematic study of actual breaches, root cause analysis, and pattern extraction from incidents causing real damage. This research-intensive approach limited the dataset to 2,418 examples versus ~81,000-86,000 synthetic test cases in Juliet. But quality beats quantity for LLM training—models learn production security patterns from 1,215 unique incidents more effectively than textbook patterns from tens of thousands of synthetic cases.

While SecureCode v2.0 contains 1,215 examples compared to Juliet's ~81,000-86,000, this reflects a deliberate design choice prioritizing incident authenticity over synthetic volume. Each of our examples required: (1) verifying a real-world security incident (CVE or documented breach), (2) analyzing the root cause vulnerability, (3) implementing both vulnerable and secure versions, (4) expert validation of exploitability, and (5) developing operational security guidance including illustrative SIEM detection strategies and infrastructure hardening recommendations. This research-intensive methodology prevents the scale achievable through synthetic generation, but produces training data that teaches models how vulnerabilities manifest in production systems rather than in textbook examples. Moreover, the 4-turn conversational structure means our 1,215 examples provide approximately 4,860 conversational exchanges (4 turns × 1,215), each containing dual implementations (vulnerable + secure), effectively yielding training signal comparable to datasets with 2-3x more examples in code-only format.

Finding 2: Conversational structure captures security workflow, code-only format doesn't

Developers don't think in vulnerable/secure code pairs. They think in iterative problem-solving: build functionality, optimize performance, handle edge cases, add monitoring. Security must persist through this entire workflow.

The 4-turn structure captures this iteration. Turn 1: developer requests authentication. Turn 2: AI provides vulnerable and secure implementations. Turn 3: developer asks about scaling to 10,000 users. Turn 4: AI maintains security while optimizing for scale and provides operational guidance.

This structure teaches models something code-only datasets can't: security is not a single decision, it's a persistent constraint across the entire development lifecycle. When you optimize for performance, security constraints still apply. When you handle failure scenarios, security still matters. When you deploy to production, you need security monitoring even if your code is perfect.

This approach was validated during SSTI fixes (Category 4, Section 4.2). Initial examples showed vulnerable template rendering and recommended "don't use user input in templates." This is technically correct but operationally useless—many applications require dynamic template rendering. The conversational structure forced addressing the follow-up question: "What if user input in templates is a business requirement?" This led to secure sandboxing implementations that production systems actually need.

Finding 3: Defense-in-depth guidance distinguishes production datasets from research datasets

Academic datasets answer the question: "What is the vulnerability and how do you fix it?" Production datasets answer: "What is the vulnerability, how do you fix it, how do you detect exploitation attempts, what do you log for incident response, and how do you fail gracefully when your security controls break?"

Turn 4 defense-in-depth guidance forces examples to address operational security. For SQL injection, this means:

- Code mitigation: parameterized queries
- Detection: database query monitoring for injection patterns
- Logging: capture failed queries with timestamps and source IPs
- Incident response: if injection detected, what data might be compromised?
- Graceful degradation: if parameterized query preparation fails, reject the query rather than fall back to string concatenation

This operational completeness emerged from the compliance journey. The initial 86 examples needing defense-in-depth enhancement (Category 3, Section 4.2) provided code mitigations but minimal operational guidance. Enhancing these examples increased Turn 4 content from 247 to 412 characters average—nearly doubling the security knowledge per example.

Finding 4: Quality validation requires automation plus human expertise, not either alone

The automated validation framework caught 97% of issues: structural problems, metadata gaps, CVE format errors, invalid language tags. Human expert review caught the remaining 3%: unrealistic attack scenarios, incomplete security controls, misleading explanations.

Neither approach alone achieves production quality. Automation without expertise accepts structurally correct but technically wrong examples. Expertise without automation introduces inconsistency as reviewers apply different standards.

The hybrid approach—automated validation enforcing structural requirements plus expert review validating security accuracy—achieved full compliance with complete consensus. The automation provided consistency at scale (validating 841 examples in seconds). The expertise provided security accuracy (verifying attack feasibility and mitigation completeness).

This hybrid validation represents the most reproducible contribution. Other researchers can use the validation framework immediately, extend it for domain-specific requirements, or adapt the methodology for different security domains.

6.2 Practical Implications

SecureCode v2.0 enables three practical applications for advancing secure AI-assisted development.

For Security Researchers: Benchmark and Foundation

SecureCode v2.0 provides the first standardized benchmark for evaluating secure code generation across AI models. Researchers can compare model security performance on the test set (104 examples), measure vulnerability detection accuracy, and assess operational security guidance quality.

The dataset also provides a foundation for specialized research:

- **Adversarial robustness:** Use examples as baselines for testing prompt injection attacks that try to trick models into generating vulnerable code
- **Model extraction defense:** Study whether fine-tuned security knowledge can be extracted through query access
- **Transfer learning:** Investigate whether security knowledge learned from these examples transfers to vulnerabilities not in the training set

The open-source release enables reproducible security research—every researcher uses the same training data, validation data, and test data, eliminating dataset variability as a confounding factor.

For Enterprise Practitioners: Production AI Training

Enterprises building internal AI coding assistants need training data meeting their security standards. SecureCode v2.0 provides validated examples covering OWASP Top 10:2025 across common enterprise languages.

Practitioners can fine-tune models on the complete dataset or create specialized models:

- **Language-specific fine-tuning:** Train Python security model on 255 Python examples
- **Category-specific fine-tuning:** Train injection prevention model on 125 injection examples (A05:2025)
- **Severity-prioritized fine-tuning:** Train on CRITICAL examples (795) first, then HIGH (384)

The 4-turn conversational structure matches how developers actually interact with AI assistants, improving fine-tuned model performance in production workflows. The defense-in-depth guidance teaches models to recommend the logging, monitoring, and detection strategies enterprises need for production deployments.

For Security Educators: Real-World Teaching Material

Security education suffers from abstract examples disconnected from real consequences. SecureCode v2.0 provides 1,215 unique real-world incidents with quantified business impact for teaching secure coding.

Educators can use these examples for:

- **University courses:** Each OWASP category provides 50+ examples for secure coding curriculum
- **Professional training:** Real breach stories with dollar amounts and user impacts create urgency
- **Certification preparation:** Coverage of OWASP Top 10:2025 aligns with CISSP, CEH, and OSCP certifications
- **Hands-on labs:** Vulnerable code examples can be deployed in isolated environments for exploitation practice

The conversational structure teaches the iterative security thinking professionals need: not just "here's the vulnerability," but "here's how to build secure functionality, optimize it, and deploy it with proper monitoring."

6.3 Limitations

SecureCode v2.0 has six limitations that researchers should be aware of when using the dataset.

L1: Language Coverage Bias

The 11-language coverage represents 96% of production deployments but shows bias toward Python (21.0%) and JavaScript (20.2%) while underrepresenting emerging languages like Rust (2.4%) and Kotlin (1.5%).

This bias reflects real-world security incident distribution—most documented breaches occur in Python and JavaScript web applications, not Rust systems programming. But the bias creates gaps for developers working primarily in underrepresented languages.

Impact: Models fine-tuned on SecureCode v2.0 will have stronger security knowledge for Python/JavaScript than Rust/Kotlin. Developers using underrepresented languages get less security guidance.

Mitigation: Future expansion in SecureCode v3.0 will add 300+ examples in Swift (iOS development), Zig (systems programming), Elixir (distributed systems), and V (performance-critical applications). This will increase coverage to 14 languages while maintaining complete incident grounding.

L2: Temporal Bias Toward Recent Incidents

CVE mining focused on 2017-2025, creating temporal bias toward recent vulnerabilities. The dataset provides strong coverage of cloud security (SSRF against AWS metadata services), API security (GraphQL abuse, JWT confusion), and container security (Docker escape, Kubernetes privilege escalation) that emerged as major threats in the past 8 years.

Coverage is weaker for legacy vulnerabilities that remain exploitable but receive less public disclosure: mainframe security, embedded systems vulnerabilities, industrial control systems. These older vulnerability classes still matter for organizations running legacy infrastructure.

Impact: Models trained on SecureCode v2.0 will recognize modern attack patterns better than legacy vulnerabilities. Organizations with legacy systems may need additional training data.

Mitigation: Future work will target expansion into legacy vulnerability categories based on user feedback. If organizations report gaps in SCADA security or mainframe security examples, historical CVEs and breach reports will be mined to add coverage.

L3: Code Complexity Limitations

Examples range from simple (50-line authentication functions) to moderate complexity (300-line API implementations) but don't represent enterprise-scale system complexity. A complete microservices architecture with service mesh, distributed tracing, and complex authorization might have 10,000+ lines of security-relevant code.

This limitation is practical, not conceptual. LLM context windows limit example complexity—a 10,000-line example exceeds most model context limits and creates training difficulties. But the complexity gap means examples teach component-level security better than system-level security architecture.

Impact: Models fine-tuned on SecureCode v2.0 will excel at securing individual functions and modules but may miss architectural security issues spanning multiple services.

Mitigation: Future research will explore hierarchical example structures where a single "example" consists of multiple related components (authentication service + API gateway + database) with security context spanning the architecture. This requires new training approaches but could teach system-level security thinking.

L4: Cultural and Geographic Bias

Incident mining relied primarily on English-language sources: U.S. CVE database, English-language breach reports, Western company disclosures. This creates geographic bias toward vulnerabilities affecting Western organizations and cultural bias toward Western security perspectives.

Security priorities differ globally. European organizations prioritize GDPR compliance and privacy. Asian organizations focus on state-sponsored attack defense. South American organizations deal with financial fraud and payment security. The dataset primarily reflects North American and Western European security priorities.

Impact: Models trained on SecureCode v2.0 may miss security concerns specific to non-Western contexts or underrepresent vulnerability classes more common in specific regions.

Mitigation: Future collaboration with international security research organizations will expand incident coverage. JPCERT/CC (Japan), CNCERT/CC (China), and CERT-BR (Brazil) maintain regional vulnerability databases that will be mined for SecureCode v3.0.

L5: LLM-Generated Training Data and Contamination Risk

SecureCode v2.0 uses multi-LLM generation (ChatGPT 5.1, Claude Sonnet 4.5, Llama 3.2) with human expert review to create training examples. This introduces potential data contamination concerns when the same models (or their successors) are later fine-tuned on this dataset.

Feedback Loop Risk: If models are fine-tuned on examples generated by earlier versions of themselves, this creates a training feedback loop that could amplify biases, reinforce incorrect patterns, or reduce diversity in generated code. The model learns from its own outputs rather than from independent ground truth.

Impact: Models fine-tuned exclusively on LLM-generated examples may exhibit reduced novelty in security solutions, perpetuate systematic biases present in the generation models, or fail to capture security knowledge absent from the original generation models' training data.

Mitigation Strategies:

1. **Human Expert Validation:** All examples underwent expert security review (Section 4.3) ensuring technical accuracy independent of generation quality

2. **Real-World Grounding:** Complete CVE/incident grounding provides external validation—examples must match documented real-world vulnerabilities, not just LLM interpretations

3. **Multi-Model Diversity:** Using three different model families (ChatGPT 5.1, Claude Sonnet 4.5, Llama 3.2) reduces single-model bias

4. **Validation Framework:** Automated structural validation (Section 4.1) enforces objective quality standards independent of generation source

5. **Hybrid Training Recommended:** Users should combine SecureCode v2.0 with code from real-world repositories, manual security examples, and human-written secure code to maintain training diversity

Research Transparency: We disclose the LLM-generation methodology to enable informed dataset usage. Researchers concerned about contamination can filter to real-world CVE-grounded content or use the dataset exclusively for evaluation rather than training.

L6: SIEM Detection Guidance is Advisory

The dataset provides SIEM integration strategies and detection recommendations in conversational format rather than platform-specific, validated detection rules. Organizations must adapt guidance to their specific SIEM platform (Splunk, Elasticsearch, Microsoft Sentinel, QRadar, etc.), log source configurations, field naming conventions, and operational thresholds.

Implementation Gap: Turn 4 responses contain operational security guidance including logging best practices and detection strategy recommendations, but not structured, machine-readable SIEM detection artifacts. Organizations receive valuable security operations context but must translate conversational guidance into platform-specific detection rules.

Impact: Detection recommendations serve as starting points requiring environment-specific implementation and tuning to achieve acceptable false positive rates and performance characteristics. Organizations cannot deploy detection mechanisms directly from the dataset—they must adapt recommendations to their logging infrastructure, tune thresholds based on baseline activity, and validate rules in non-production environments before deployment.

Mitigation: Future SecureCode v2.1 enhancement will add structured SIEM detection artifacts following a comprehensive JSON schema specification. This will include platform-specific rule implementations (Splunk SPL, Elasticsearch Query DSL, Sigma universal format), explicit log source requirements with field mappings, threshold tuning recommendations, and false positive mitigation strategies. Organizations will receive template rules that significantly reduce time-to-detection while still requiring environment-specific adaptation.

6.4 Threats to Validity

We address four categories of validity threats in the research design and future empirical evaluation.

Internal Validity: Confounding Factors

Threat: Fine-tuning hyperparameters, model architecture differences, or training randomness could confound security improvements attributed to the dataset.

Mitigation: Planned empirical evaluation will optimize hyperparameters independently for each model, control for architecture differences by testing multiple model families, and run multiple trials with different random seeds to account for training variance. Statistical significance testing (two-tailed t-tests, $p < 0.001$) will confirm improvements are not due to chance.

External Validity: Generalization

Threat: Results might not generalize beyond specific evaluation benchmarks, model architectures, or vulnerability categories.

Mitigation: Future evaluation will test on multiple independent benchmarks (CWE-Sans, custom vulnerability detection, HumanEval), test diverse model architectures (GPT family, Code Llama, StarCoder), and measure performance across all 11 OWASP categories separately to identify category-specific effects. Real-world deployment case studies will validate security improvements in production environments.

Construct Validity: Measurement Accuracy

Threat: Vulnerability classification, severity assignments, or quality metrics might not accurately measure the intended constructs.

Mitigation: The research used industry-standard OWASP taxonomy for categorization, CVSS scores for severity where available, and independent security expert validation (Section 4.3) for quality assessment. Inter-rater reliability (Cohen's κ = 0.87) indicates substantial agreement on construct measurement.

Conclusion Validity: Statistical Rigor

Threat: Insufficient sample sizes, violated statistical assumptions, or inappropriate statistical tests could lead to incorrect conclusions.

Mitigation: Planned evaluation will use appropriate sample sizes (minimum 100 examples per test condition), verify statistical test assumptions before application, and report effect sizes alongside p-values to distinguish statistical significance from practical significance. Conservative significance thresholds ($p < 0.001$) will reduce false positive risk.

---

7. Conclusion

Studies show AI coding assistants can generate vulnerable code in 45% of security-relevant scenarios [1,2], and developers using these tools may write less secure code than those working alone [2]. This happens because models learn from millions of insecure examples in public code repositories. SecureCode v2.0 addresses this problem by providing production-grade secure coding examples that teach models what security looks like in real systems.

The dataset delivers 1,215 rigorously validated unique examples achieving full compliance with strict quality standards. Every example ties directly to documented security incidents with CVE references. Every example provides both vulnerable and secure implementations with concrete attack demonstrations. Every example includes defense-in-depth operational guidance covering logging, monitoring, detection, and incident response. This is the first secure coding dataset meeting enterprise quality standards for AI training.

SecureCode v2.0 implements 4-turn conversations that mirror actual developer-AI security workflows, escalating from basic implementations through advanced scenarios to operational hardening. This conversational structure captures how security knowledge actually transfers during development—not through abstract vulnerable/secure code pairs, but through iterative problem-solving where security persists as a constraint across the entire development lifecycle.

The quality assurance journey demonstrates that production-grade datasets require systematic validation. Starting at 47.2% compliance (397 of 841 training examples passing all checks), we achieved full compliance through 604 fixes across five categories: CVE format standardization (452 fixes), language tag mapping (60 fixes), defense-in-depth enhancement (86 fixes), secure SSTI implementations (6 fixes), and validator calibration (eliminating false positives). This rigorous validation process distinguishes production datasets from research datasets.

Key findings challenge conventional approaches in security dataset design. Real-world grounding is non-negotiable—synthetic examples can't teach the context making vulnerabilities exploitable in production. Conversational structure matters—code-only formats miss the iterative workflows where security failures actually occur. Defense-in-depth guidance distinguishes production datasets—code mitigations alone don't address the detection, monitoring, and incident response that production systems require. Quality validation needs automation plus expertise—neither approach alone achieves production standards.

SecureCode v2.0 enables three practical applications. Security researchers gain the first standardized benchmark for evaluating secure code generation across AI models plus a foundation for adversarial robustness and transfer learning research. Enterprise practitioners can fine-tune internal AI coding assistants on production-grade security examples covering OWASP Top 10 across common enterprise languages. Security educators get 1,215 unique real-world incidents with quantified business impact for teaching secure coding with the urgency and context students need.

Four limitations exist in the current dataset. Language coverage shows bias toward Python/JavaScript while underrepresenting Rust/Kotlin. Temporal bias toward recent incidents (2017-2025) creates gaps in legacy vulnerability coverage. Code complexity limitations mean examples teach component-level security better than system-level architecture. Cultural and geographic bias toward Western sources may miss security concerns specific to non-Western contexts. These limitations reflect trade-offs between dataset quality (complete incident grounding) and comprehensive coverage (which would require including more synthetic examples).

Note on Empirical Evaluation: This technical report focuses on dataset construction, validation, and quality metrics. Empirical evaluation of model performance after fine-tuning on SecureCode v2.0 is planned for future conference publication. We hypothesize 15-25% improvements in secure code generation and vulnerability detection without degrading code functionality, to be validated through controlled experiments and ablation studies.

We release SecureCode v2.0, the validation framework, fine-tuning examples, and evaluation benchmarks as open-source contributions to advance secure AI-assisted development. Researchers can reproduce these results, extend the methodology, or use the dataset as a foundation for domain-specific security training. Practitioners can immediately improve security of enterprise AI coding assistants. Educators can teach secure coding through real-world incidents rather than abstract examples.

The future of software development involves AI coding assistants generating billions of lines of code annually. Whether that code is secure or vulnerable depends entirely on what these models learn during training. SecureCode v2.0 provides the production-grade training data needed to teach AI assistants the security knowledge that current models lack. This work aims to make secure code generation the default, not the exception.

Availability

Dataset: HuggingFace Hub: https://huggingface.co/datasets/scthornton/securecode-v2 (https://huggingface.co/datasets/scthornton/securecode-v2)

Source Code: GitHub: https://github.com/scthornton/securecode-v2 (https://github.com/scthornton/securecode-v2)

Validation Framework: https://github.com/scthornton/securecode-v2/blob/main/validate_contributing_compliance.py (https://github.com/scthornton/securecode-v2/blob/main/validate_contributing_compliance.py)

Documentation: Technical Report: https://perfecxion.ai/articles/securecode-v2-dataset-paper.html (https://perfecxion.ai/articles/securecode-v2-dataset-paper.html)

Acknowledgments

We thank the security research community for responsible disclosure practices that made incident grounding possible. We thank the three anonymous security experts who provided independent validation achieving complete consensus (Section 4.3). We thank the OWASP Foundation for maintaining the Top 10 taxonomy that guided categorization. We thank MITRE Corporation for maintaining the CVE database that enabled incident mining.

References

[1] Veracode. (2025). "2025 GenAI Code Security Report: Assessing the Security of Using LLMs for Coding." *Veracode Research*.

[2] Apiiro Security Research. (2025). "The State of Application Security 2025: How AI Coding Copilots Impact Security Posture." *Apiiro*.

[3] CWE-Sans Top 25 Dataset (2019). MITRE Corporation and SANS Institute. Available: https://cwe.mitre.org/top25/

[4] Boland, T., & Black, P. (2012). "Juliet 1.1 C/C++ and Java Test Suite." *IEEE Computer*, 45(10), 88-90. doi: 10.1109/MC.2012.345. Test suite available: https://samate.nist.gov/SARD/test-suites/112

[5] Software Assurance Reference Dataset (SARD) (2021). National Institute of Standards and Technology. Available: https://samate.nist.gov/SARD/

[6] Russell, R., et al. (2018). "Automated Vulnerability Detection in Source Code Using Deep Representation Learning." *17th IEEE International Conference on Machine Learning and Applications (ICMLA)*.

[7] Sandoval, G., et al. (2023). "Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants." *32nd USENIX Security Symposium*.

[8] Pillar Security Research. (2025). "Rules File Backdoor: A New Attack Vector Against LLM Applications." *Pillar Security Blog*.

[9] Zhao, H., et al. (2024). "A Survey of Attacks on Large Vision-Language Models: Resources, Advances, and Future Trends." *arXiv:2407.07403*.

[10] Yefet, N., et al. (2020). "Adversarial Examples for Models of Code." *Proceedings of the ACM on Programming Languages, OOPSLA*.

[11] Wallace, E., et al. (2021). "Concealed Data Poisoning Attacks on NLP Models." *2021 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

[12] OWASP Foundation (2025). "OWASP Top 10:2025 Release Candidate." Available: https://owasp.org/Top10/2025/ (accessed December 2025)

[12b] OWASP Foundation (2021). "OWASP Top 10 2021." Available: https://owasp.org/Top10/ (historical reference for dataset creation context)

[13] MITRE Corporation (2025). "Common Weakness Enumeration (CWE)." Available: https://cwe.mitre.org/ (List Version 4.19)

[14] Wang, B., et al. (2024). "CodeSecEval: A Benchmark for Evaluating LLM-Assisted Code Generation's Security Posture." *arXiv:2407.02395*.

[15] National Vulnerability Database (2025). National Institute of Standards and Technology. Available: https://nvd.nist.gov/

[16] Verizon (2025). "2025 Data Breach Investigations Report." Available: https://www.verizon.com/business/resources/reports/dbir/

[17] IBM Security (2025). "X-Force Threat Intelligence Index 2025." Available: https://www.ibm.com/security/data-breach/threat-intelligence/

[18] Austin, A., et al. (2021). "Security Smell Detection in Infrastructure as Code using Machine Learning." *IEEE International Conference on Software Maintenance and Evolution (ICSME)*.

[19] Nguyen, N., & Nadi, S. (2022). "An Empirical Evaluation of GitHub Copilot's Code Suggestions." *19th International Conference on Mining Software Repositories (MSR)*.

[20] Schneider, J., et al. (2022). "Evaluating the Code Quality of AI-Assisted Code Generation Tools." *arXiv preprint arXiv:2206.13909*.

[21] Asare, O., et al. (2023). "GitHub Copilot: The Impact on Productivity and Code Quality." *Empirical Software Engineering Journal*.

[22] Brown, T., et al. (2020). "Language Models are Few-Shot Learners." *Advances in Neural Information Processing Systems 33 (NeurIPS)*.

[23] Li, Y., et al. (2023). "StarCoder: A advanced LLM for Code." *arXiv preprint arXiv:2305.06161*.

[24] Roziere, B., et al. (2023). "Code Llama: Open Foundation Models for Code." *arXiv preprint arXiv:2308.12950*.

[25] Bhatt, M., Chennabasappa, S., Li, Y., Nikolaidis, C., Song, D., Wan, S., Ahmad, F., Aschermann, C., Chen, Y., Kapil, D., Molnar, D., Whitman, S., & Saxe, J. (2024). "CyberSecEval 2: A Wide-Ranging Cybersecurity Evaluation Suite for Large Language Models." *arXiv preprint arXiv:2404.13161*. https://doi.org/10.48550/arXiv.2404.13161

Appendix A: Dataset Schema

SecureCode v2.0 examples follow this JSON schema:

```
{
  "id": "unique-example-identifier",
  "owasp_category": "A05:2025-Injection",
  "cve_id": "CVE-2023-12345",
  "incident_name": "MOVEit Transfer SQL Injection",
  "incident_reference": "https://www.cisa.gov/news-events/cybersecurity-advisories/aa23-158
  "severity": "CRITICAL",
  "cvss_score": 9.8,
  "severity_rationale": "CVSS v3.1 score 9.8 (Critical) - unauthenticated SQL injection wit
  "language": "python",
  "incident_year": 2023,
  "business_impact": "$2.3M in fraud losses, 50K customer records exposed",
  "conversation": [
    {
      "role": "user",
      "content": "Build user authentication with JWT tokens for a REST API..."
    },
    {
      "role": "assistant",
      "content": "**Vulnerable Implementation:**\n[code]\n\n**Attack:**\n[demonstration]\n\
    },
    {
      "role": "user",
      "content": "How does this scale to 10,000 concurrent users?..."
    },
    {
      "role": "assistant",
      "content": "**Scaling Security:**\n[guidance]\n\n**Logging:**\n[strategy]\n\n**Detect
    }
  ]
}
```

Required Fields:

- `id` : Unique identifier (string)
- `owasp_category` : OWASP Top 10:2025 category or AI-ML-Security-Custom
- `cve_id` : CVE-YYYY-NNNNN or null (if null, incident_name and incident_reference required)
- `incident_name` : Human-readable incident name (required when cve_id is null)
- `incident_reference` : URL to security advisory, breach report, or bug bounty disclosure (required when cve_id is null)
- `severity` : CRITICAL, HIGH, MEDIUM, or LOW
- `language` : One of 11 supported languages

- `incident_year` : 2017-2025
- `business_impact` : Quantified impact description
- `conversation` : Array of exactly 4 turns alternating user/assistant roles

Optional Fields:

- `cvss_score` : CVSS v3.1 base score (0.0-10.0) when available from NVD
- `severity_rationale` : Explanation of severity assignment (especially for non-CVE incidents)

Validation: All examples validated using `validate_contributing_compliance.py` framework.

Appendix B: OWASP Top 10:2025 Category Distribution

Detailed breakdown of 1,215 examples across OWASP Top 10:2025 categories:

OWASP Category	Count	Percentage	Top Languages	Severity Distribution
A01:2025 Broken Access Control	224	18.4%	Python, JavaScript, Java	CRIT: 146, HIGH: 71, MED: 7
A07:2025 Authentication Failures	199	16.4%	Python, JavaScript, Java	CRIT: 130, HIGH: 62, MED: 7
A02:2025 Security Misconfiguration	134	11.0%	JavaScript, Python, Go	CRIT: 88, HIGH: 42, MED: 4
A05:2025 Injection	125	10.3%	Python, JavaScript, PHP	CRIT: 82, HIGH: 39, MED: 4
A04:2025 Cryptographic Failures	115	9.5%	Python, Java, C#	CRIT: 75, HIGH: 37, MED: 3
A03:2025 Software Supply Chain Failures	85	7.0%	JavaScript, Ruby, Python	CRIT: 56, HIGH: 27, MED: 2
A06:2025 Insecure Design	84	6.9%	Python, JavaScript, Java	CRIT: 55, HIGH: 27, MED: 2
A08:2025 Software or Data Integrity Failures	80	6.6%	Java, Python, C#	CRIT: 52, HIGH: 25, MED: 3
Unknown	60	4.9%	Multiple	CRIT: 39, HIGH: 19, MED: 2
A09:2025 Security Logging & Alerting Failures	59	4.9%	Python, JavaScript, Java	CRIT: 39, HIGH: 19, MED: 1
AI/ML Security (Custom Category)	50	4.1%	Python (primary), JavaScript	CRIT: 33, HIGH: 16, MED: 1

Total: 1,215 examples

Coverage Notes:

- Broken Access Control (A01) receives highest coverage (18.4%, including merged SSRF examples) as most common breach vector
- Authentication Failures (A07) is second (16.4%) as identity failures cause widespread compromise
- Security Misconfiguration moved to A02:2025 (from A05:2021), reflecting increased industry priority

- Software Supply Chain Failures renamed from "Vulnerable and Outdated Components" with expanded scope
- A10:2021 SSRF (45 examples, 3.7%) merged into A01:2025 per OWASP Top 10:2025 consolidation
- AI/ML Security is a custom category addressing LLM-specific threats (4.1%)
- All major categories include examples from multiple programming languages
- CRITICAL severity dominates (65.4%) matching real-world threat distribution

---

Appendix C: Programming Language Distribution

Language coverage with representative frameworks and use cases:

Language	Examples	%	Top Frameworks/Libraries	Primary Use Cases
Python	255	21.0%	Django, Flask, FastAPI, requests	Web apps, APIs, ML/AI, data processing
JavaScript	245	20.2%	Express, React, Vue, Node.js	Full-stack web, APIs, SPAs
Java	189	15.6%	Spring Boot, Jakarta EE, Android SDK	Enterprise apps, Android, microservices
Go	159	13.1%	Gin, Echo, net/http, gRPC	Microservices, CLI tools, infrastructure
PHP	102	8.4%	Laravel, Symfony, WordPress	Web applications, CMS, legacy systems
C#	85	7.0%	.NET Core, ASP.NET, Entity Framework	Enterprise apps, Azure, desktop software
TypeScript	72	5.9%	Angular, NestJS, Express with types	Type-safe web apps, enterprise frontend
Ruby	48	4.0%	Ruby on Rails, Sinatra, Grape	Web apps, APIs, automation
Rust	29	2.4%	Actix, Rocket, Tokio, wasm-bindgen	Systems programming, WebAssembly, performance
Kotlin	18	1.5%	Ktor, Spring Boot, Android KTX	Android apps, backend services, multiplatform
YAML	13	1.1%	Kubernetes, Docker Compose, CI/CD	Configuration files, infrastructure as code

Total: 1,215 examples

Note: Percentages sum to 100.2% due to rounding.

Coverage Strategy:

- Python/JavaScript (41% combined): Dominate web development vulnerability landscape
- Java/Go/PHP (37% combined): Enterprise systems and cloud infrastructure
- C#/TypeScript (13% combined): Enterprise and type-safe development
- Ruby/Rust/Kotlin/YAML (9% combined): Specialized frameworks and configuration

Appendix D: Validation Framework Implementation

Core validation checks from `validate_contributing_compliance.py`:

1. Structure Validation

```
def validate_structure(example):
    # Check conversation has exactly 4 turns
    if len(example['conversation']) != 4:
        return False, "Must have exactly 4 turns"

    # Check turn roles alternate user/assistant
    expected_roles = ['user', 'assistant', 'user', 'assistant']
    actual_roles = [turn['role'] for turn in example['conversation']]
    if actual_roles != expected_roles:
        return False, "Roles must alternate user/assistant"

    return True, "Structure valid"
```

2. CVE Format Validation

```
import re

def validate_cve_format(cve_id):
    if cve_id is None:
        return True, "Explicit null accepted"

    # CVE format: CVE-YYYY-NNNNN where YYYY is 1999-2029, NNNNN is 1-5 digits
    # Note: Regex validates format, not semantic validity (e.g., allows CVE-2024-00000 but
    pattern = r'^CVE-(199[9]|20[0-2][0-9])-\d{1,5}$'
    if re.match(pattern, cve_id):
        return True, "Valid CVE format"

    return False, f"Invalid CVE format: {cve_id}"
```

3. Content Length Validation

```
def validate_content_length(example):
    user_min = 50  # characters
    assistant_min = 100  # characters

    errors = []

    # Turn 1 (user): minimum 50 chars
    if len(example['conversation'][0]['content']) < user_min:
        errors.append(f"Turn 1 below {user_min} chars")

    # Turn 2 (assistant): minimum 100 chars
    if len(example['conversation'][1]['content']) < assistant_min:
        errors.append(f"Turn 2 below {assistant_min} chars")

    # Turn 3 (user): minimum 50 chars
    if len(example['conversation'][2]['content']) < user_min:
        errors.append(f"Turn 3 below {user_min} chars")

    # Turn 4 (assistant): minimum 100 chars
    if len(example['conversation'][3]['content']) < assistant_min:
        errors.append(f"Turn 4 below {assistant_min} chars")

    if errors:
        return False, "; ".join(errors)
    return True, "Content length valid"
```

4. Language Tag Validation

```
SUPPORTED_LANGUAGES = {
    'python', 'javascript', 'java', 'php', 'csharp',
    'ruby', 'go', 'typescript', 'rust', 'kotlin', 'yaml'
}

def validate_language(language):
    if language.lower() in SUPPORTED_LANGUAGES:
        return True, "Valid language tag"

    return False, f"Unsupported language: {language}"
```

5. Complete Example Validation

```
def validate_example(example):
    results = {
        'structure': validate_structure(example),
        'metadata': validate_metadata_complete(example),
        'cve_format': validate_cve_format(example.get('cve_id')),
        'language': validate_language(example.get('language', '')),
        'content_length': validate_content_length(example)
    }

    # Example passes only if all checks pass
    all_passed = all(result[0] for result in results.values())

    return all_passed, results
```

This framework enabled our compliance journey from 47.2% to 100% by systematically identifying and categorizing validation failures for targeted fixes.

Thank You for Reading