# AI Testing & Security Agent: Project Reflection

## Abstract

I built a fully autonomous AI software engineering agent using the Model Context Protocol (MCP) that can iteratively test, analyze, and repair a Java Maven codebase. Beyond just generating basic tests and fixing bugs, I extended the system with some pretty cool features: specification-based test generators, a static security scanner, taint analysis, and a CVE-based dependency checker. The agent actually improved test coverage, caught configuration and runtime bugs, and found several security issues. This reflection walks through the engineering process, what improved, the security insights I gained, and what I learned about AI-assisted development.

## 1. Introduction

Autonomous software engineering agents are becoming real, practical tools for CI/CD, testing, and quality improvement. I used MCP as the foundation to build a multi-stage intelligent agent that handles test generation, bug detection, security auditing, and development automation. The goal was to see how well an AI agent could actually maintain and improve a real Java Maven project through repeated refinement cycles.

What makes this different from traditional automated testing tools is that I added advanced extensions like boundary-value test generation, equivalence-class reasoning, decision-table analysis, and static security pattern detection. Together, these components turned the agent into a pretty comprehensive quality-assurance toolchain.

## 2. How I Built It

The agent's workflow has four main phases: running tests, analyzing coverage, automatically enhancing tests, and detecting bugs and security issues. I implemented this through a set of MCP tools.

### 2.1. Automated Build and Coverage Analysis

I created tools like `run_maven_test`, `get_quality_dashboard`, `get_test_failures`, and `get_missing_cove`
that execute full Maven builds, collect Surefire and JaCoCo reports, and pinpoint exactly which lines aren't tested or which methods are failing.

### 2.2. Specification-Based Test Generation

I implemented three advanced test generators:

- **Boundary Value Analysis Generator:** Extracts numerical parameters from method signatures and creates tests using boundary values like {-1, 0, 1, 100}
- **Equivalence Class Generator:** Produces representative tests for valid, invalid, and edge-case categories
- **Decision Table Generator:** Parses conditional logic (like boolean combinations in if statements) and generates tests to cover all possible truth-table combinations

These tools really expanded test depth and made sure that input partitions and branching logic got thoroughly covered.

## 3. Security Analysis Extensions

I added two major security features:

### 3.1. Static Security Analyzer

This custom tool scans Java files for common vulnerabilities:

- SQL injection patterns through string concatenation
- Command injection via `Runtime.exec`
- Weak cryptography like MD5 and SHA-1
- Insecure randomness (using `Random` instead of `SecureRandom`)
- Path traversal risks with user-controlled input
- Hard-coded secrets like passwords and API keys

I also built in lightweight taint analysis to track when `request.getParameter()` inputs flow into dangerous sinks.

### 3.2. CVE Dependency Scanner

This tool analyzes project dependencies using Maven's JSON dependency tree and cross-references them against a CVE database. It reports:

- Identified CVEs
- CVSS severity scores
- Recommended fixed versions
- Overall project risk score

## 4. Git Automation

The agent handles all version control operations automatically—staging, committing, pushing, and creating pull requests. This lets the entire refinement loop run without me having to touch anything.

## 5. What Happened

### 5.1. Coverage Improvements

Coverage increased steadily with each iteration. Initially, only basic arithmetic methods were covered. After I integrated the specification-based generators, previously untested logic—especially branches in `max()`, `min()`, `divide()`, and `factorial()`—started getting meaningful coverage. The decision-table testing tool was particularly effective for comprehensive condition coverage.

The agent's ability to read JaCoCo XML reports and generate targeted tests turned out to be a really reliable way to incrementally improve things.

### 5.2. Bug Detection and Automated Fixes

During development, the system caught and fixed several issues:

- A NullPointerException in Java version parsing logic
- A reflection-access error that needed `--add-opens` flags in Surefire
- Multiple failing tests caused by incorrect configuration between JaCoCo and Surefire

The automated repair mechanism let the agent apply small, targeted changes to the codebase, which really showed the value of autonomous debugging loops.

### 5.3. Security Findings

The security tools uncovered multiple risks:

- Potential SQL injection vulnerabilities from unsafe string concatenation
- Use of MD5 hashing (flagged as weak cryptography)
- Hard-coded secrets in sample classes
- Tainted input flowing into file constructors

On top of that, dependency scanning flagged a Log4j version with a known critical CVE and suggested upgrading to a safe version.

These results showed that integrating security analysis into the testing loop really broadened what the agent could do beyond just checking if the code works.

## 6. What I Learned About AI-Assisted Development

A few things stood out:

- AI excels at repetitive, structured tasks like test generation and coverage scanning. It doesn't get bored or make careless mistakes.
- Combining static patterns with taint analysis gives you powerful security insights without much computational overhead.
- Automated configuration debugging (like fixing InaccessibleObjectException) has real practical value in actual builds.
- Security-focused reasoning makes the agent way more useful for enterprise-grade workflows.

Overall, AI assistance sped up development by cutting down on manual debugging time and making everything more consistent.

## 7. Ideas for Future Improvements

Here's what I'd add if I kept working on this:

- Integrate symbolic execution to reach deeper semantic paths
- Expand CVE data ingestion using live NVD feeds
- Apply machine learning to prioritize generated tests based on which ones catch the most bugs
- Add automated patch verification to make sure fixes don't break existing behavior

## 8. Conclusion

This project proved that an MCP-powered AI agent can autonomously analyze, refine, and secure a real Java codebase. Through iterative testing, bug fixing, security scanning, and dependency auditing, the system delivered measurable quality improvements. The combination of functional and security-focused extensions worked especially well, showing that AI-driven engineering assistants can genuinely enhance modern software development pipelines.