Reflection on an MCP-Based Testing Agent for Apache Commons Lang

Taha Shad
DePaul University
SE 333: Software Testing
Email: tshad@depaul.edu

***Abstract*—This reflection report describes my experience building an intelligent testing agent using the Model Context Protocol (MCP) for the SE333 final project. The agent was designed to integrate with a real Java codebase (Apache Commons Lang), run Maven tests, generate and summarize JaCoCo coverage reports, and propose directions for new JUnit tests. I discuss the overall architecture, how I combined Python tooling with Maven and JaCoCo, and what I learned about AI-assisted software development. The results section reflects on coverage patterns in the Lang project, the strengths and limitations of using an AI agent inside Visual Studio Code, and concrete ideas for future extensions of the toolchain.**

I. Introduction

The goal of the final project was to explore how AI-assisted tooling can automate parts of the testing workflow for a non-trivial Java project. Rather than simply writing more tests by hand, the assignment asked us to build an MCP server that exposes tools to the model: running Maven tests, generating coverage data, analyzing JaCoCo reports, and interacting with Git.

I chose the Apache Commons Lang3 project as the target system. It is mature, well-tested, and structured as a standard Maven project, which made it a good candidate for experimenting with test-automation workflows. The main research question I kept in mind was: *How far can an AI agent, armed with the right tools, go in supporting coverage-driven testing on a real codebase?*

II. Methodology

A. MCP Server and Tool Design

I implemented the MCP server in Python using the `fastmcp` framework. The server exposes several tools, each corresponding to a concrete development action:

- `run_maven_tests`: runs `mvn test` in the Lang3 project and returns the console output.
- `summarize_coverage`: parses the `jacoco.xml` file and produces a human-readable summary of coverage by counter type (instruction, branch, line, method, class).
- `suggest_junit_tests_for_class`: given a fully qualified class name, generates skeleton JUnit tests based on method signatures.
- `suggest_boundary_tests`: produces boundary-value and equivalence-class ideas for a textual API description.
- `git_status`, `git_add_all`, `git_commit`, and `git_push`: lightweight Git automation tools to keep the repository synchronized with GitHub.

These tools were exposed to the VS Code Chat view via an MCP configuration file. With auto-approve enabled, the agent could call them directly while I interacted through natural language prompts.

B. Maven and JaCoCo Integration

On the Java side, I configured the Commons Lang3 `pom.xml` to include the JaCoCo Maven plugin. The plugin is set up to attach a Java agent, run during the `test` phase, and generate an XML report under `target/site/jacoco/jacoco.xml`. I also added a small compatibility configuration for newer Java versions to avoid reflection-related test failures.

To analyze coverage, the MCP tool `summarize_coverage` reads the XML file, iterates through the global `counter` elements, and prints percentages for each counter type. For more detailed analysis, I used a small script and a PowerShell pipeline to list the top source files by missed lines, which helped identify realistic targets for improvement.

C. Workflow

My typical workflow for an iteration was:

1) Ask the agent to run `run_maven_tests`. Inspect failures, if any.
2) Call `summarize_coverage` to get updated global coverage statistics and the list of most-missed packages and classes.
3) Pick a candidate class (for example, a range or reflection utility) and request `suggest_junit_tests_for_class`.
4) Use `suggest_boundary_tests` to refine ideas around edge cases, such as empty ranges or null inputs.
5) Manually refine and implement a subset of the suggested JUnit tests, then re-run Maven and coverage.
6) When a meaningful change was made, use the Git tools to stage, commit, and push the updates to GitHub.

This pipeline deliberately kept a human in the loop while offloading repetitive tasks (running commands, parsing XML, drafting boilerplate tests) to the agent.

III. Results and Discussion

A. Coverage Patterns

The JaCoCo report for Commons Lang showed that the project already has strong overall coverage. Line coverage sits in the low 90% range for the `org.apache.commons.lang3` package, with some sub-packages such as `reflect`, `builder`, `time`, and `text` contributing most of the remaining missed lines.

The agent's coverage summary tool highlighted these hotspots clearly, which was an improvement over manually navigating the HTML report. It also reinforced an important lesson: on mature projects, coverage improvements tend to be incremental rather than dramatic. Most remaining gaps were in complex branches, unusual error paths, or rarely used configuration options.

Where I did add or refine tests (for example, around numeric ranges and some utility methods), the changes produced small but visible improvements in per-class coverage and, more importantly, surfaced subtle behaviors that were not obvious from the API documentation alone. Even when coverage numbers moved only slightly, the structured view from the agent made it easier to reason about where effort was paying off.

B. Lessons Learned About AI-Assisted Development

This project made several aspects of AI-assisted development very concrete:

1) Tools matter as much as prompts: The same model felt much more useful once it had access to specialized tools. Without MCP, the agent could only talk about testing in the abstract. With tools like `run_maven_tests` and `summarize_coverage`, it could operate on the actual project state and provide grounded advice.

2) Human oversight is still essential: The JUnit skeletons and boundary ideas generated by the agent were helpful as starting points, but they often needed adjustment. Some suggested tests were redundant with existing ones, and others targeted branches that were not meaningful edge cases. Deciding which tests were worth implementing remained a human judgement call.

3) Error handling is a first-class feature: Integrating with Maven, JaCoCo, and Git revealed how fragile fully automated workflows can be: version mismatches, failing legacy tests, or authentication issues can all block progress. Designing tools that fail gracefully and return informative error messages to the agent was just as important as the "happy path" functionality.

4) AI as a collaborator, not a replacement: The most productive interactions felt like pair programming with a very fast assistant. I relied on the agent to summarize logs, point me at suspicious branches, and draft boilerplate code, but I stayed responsible for design decisions, refactoring, and verifying that new tests truly reflected intended behavior.

C. Future Enhancement Recommendations

If I had more time to extend this project, I would focus on three directions:

- **Tighter feedback loop.** Right now, I manually decide which suggested tests to implement. A more advanced agent could automatically create candidate tests in a sandbox branch, run them, and only propose those that increase coverage without introducing flaky behavior.
- **Richer coverage analytics.** The current summary operates at package and file level. It would be useful to add per-method statistics, visualizations of branch coverage, and correlation between coverage and historical test failures or bug fixes.
- **Deeper Git integration.** The Git tools could be extended to open pull requests, attach coverage summaries to commit messages, and enforce simple policies (for example, rejecting commits that reduce coverage below a threshold).

These enhancements would push the agent closer to an autonomous CI assistant rather than a local scripting helper.

IV. Conclusion

Building an MCP-based testing agent for Apache Commons Lang gave me hands-on experience with combining traditional testing tools and AI-driven automation. The project showed that large language models become far more useful when connected to concrete tools such as Maven, JaCoCo, and Git, and when their role is framed as an interactive partner instead of a fully autonomous developer.

From a testing perspective, the agent made it easier to see where coverage was lacking and to brainstorm new test ideas, even if human judgement was still required to decide which tests were valuable. From a broader software engineering perspective, the project illustrated how important it is to design reliable, well-scoped tools and clear workflows before bringing AI into the loop.

Overall, the experience confirmed the course message that AI-assisted development is less about replacing engineers and more about augmenting them. With the right toolchain and a thoughtful workflow, intelligent agents can help turn coverage analysis and test generation from a tedious chore into a more systematic, data-driven process.

References

[1] SE333 Final Project Handout: "Final Project on Software Agents," DePaul University, 2025.