Reflection of Automating Test Development with AI Agents and MCP Tools

Jose Sandoval
November 16, 2025
College of Computing and Digital Media
DePaul University
Chicago, IL, USA
jsando47@depaul.edu

***Abstract*—In this project's reflection report, I provide an examination of the process behind automating test development through the use of AI-assisted software principles, tools, and prompt engineering techniques. In part, the project involves the implementation of a Mode Context Protocol (MCP) for Visual Studio's Copilot to use for testing and development guidance that led to a 97% total coverage and 93% branch coverage of the provided Java codebase. This evolved in phases, starting from a basic manual setup for JaCoCo primarily to create reports, as well as other dependencies and a python environment for the fastMCP development. Through an iterative prompt engineering process, tools were developed for bug testing, code review, and even Git version control automation as development evolved and the tested coverage increased. This report also covers the analysis of coverage that is being improved through iterations, lessons learned throughout the process, and some recommendations for future enhancements to be implemented.**

***Index Terms*—AI-assisted, automation, development, testing, test coverage, MCP, Model Context Protocol, AI Agent, JaCoCo, Git, version control**

I. Introduction

AI assistance has widely disrupted the traditional software development process, and in particular, the manual tasks requiring repetitive actions such as generating tests for the codebase. The primary application was GitHub Copilot in Visual Studio Code (VSCode) and Agent mode with 'YOLO' option enabled for full autonomy permissions. In this state, the AI Agent was to complete a fully automated testing framework for the Java code provided. The prompt engineering and system were designed to provide instructions for the Agent to iteratively develop, conduct tests, analyze the resulting reports, and refactor to continuously improve coverage closer to 100%.

The provided codebase consisted of about 108 different Java classes in varying code length and implementations. The project resulted in 19 different MCP tools over the span of development with some initially generated by the Agent as it recognized patterns that could be more efficiently conducted. This was by design a part of the prompt to ensure it could also self-heal and improve its own methodology to preserve token usage, time, and overall resources. The baseline total coverage of the agent's test generation was a respectable 92% and after several iterations, it slowly crawled its way to achieving wider coverage with more focus on branch coverage. The resulting total coverage reached 97% with 1,417 of 54,972 missed and 93% branch coverage with 508 of 8,122 missed.

II. Methodology

This project took a phase approach when developing:

Phase 1: Environment Setup - Manual setup stage that involved installing and running VSCode's current version with AI integration, Node.js 18+, Git, GitHub Account, Java 11+, and Maven 3.6+. Python was installed with the uv package manager to initialize a virtual environment for this project's dependencies which primarily consisted of fastMCP. VSCode's was configured with 'YOLO' mode to provide the AI Agent auto-approve permissions.

Phase 2: Core Testing Agent Development - This phase continued manual setup with focus on the Core Testing Agent Development which included integration of JaCoCo for coverage recording and reporting, the MCP tools developed to aid the Agent in repetitive tasks, specific scenarios, and help it analyze, fix, and repeat processes.

Phase 3: Git Automation Tools - Additional MCP tools were created to provide direct ability for the Agent to automatically and independently complete version control actions including the commits and pushing to the main branch on GitHub.

Phase 4: Intelligent Test Iteration - Two additional tools were created to find and detect bugs when test failures occurred as well as providing a metrics generation. With these tools, the Agent had ways to identify code bugs and be able to track the quality of the test being generated as an indicator of iteration effectiveness.

Phase 5: Creative Extensions - Designed and implemented two additional MCP tools to extend the capabilities of the MCP. One was the specification-based test generator with value analysis, and also, an AI code review for the agent to conduct static analysis to include code issues and common vulnerabilities.

III. Results

A. Coverage Improvement Patterns

This project moved quickly through iterations and it consistently showed improved coverage in the systematic generation approach. Immediately at first iteration, the report analysis

discovered that the TypeUtils class had the lowest coverage at only 35% and the ToStringBuilder class was at 66% while the Conversion class was much greater at 94%. While the instructions mentioned achieving full coverage at 100% as the goal, it didn't lead the Agent to focus on the lowest coverage classes first. Instead, the Agent focused on achieving 100% on those that were much closer in the 90's until specifically instructed to focus on the least covered classes.

Coverage metrics showed steady improvement with ending numbers:

- Branch coverage: 93% (7614/8122)
- Instruction coverage: 97% (53,555/54,972)

The effectiveness of batch processing was also evident when it generated tests for five classes while running test suites, analyzing the resulting coverage statistics, and using that information as guidance to further iterate.

Having Git automation and integrated through the MCP tools effectively aided in showcasing the progress while provided points in time where things could be rolled back or used as guidance as iterations proceeded. The audit trail for progress was clear and closely tracked smaller increments as it got closer to 100%.

B. Lessons Learned About AI-Assisted Development

There were many insights learned and observed while working with AI as an assistant for this development:

The initial guidance provided through the prompt file resulted in mixed results that led to refining the instructions to obtain results that were closer in-line with what was intended. The AI Agent struggled at times follow strictly what it was informed to do, so specific details had to be clearer for it to eventually achieve a happy-medium of independent actions while following the guard rails in the prompt. This proved to be useful as a means for oversight on the quality it was returning as it may to have been obvious if it didn't immediately defy some of the prompting.

Context is Critical - When the prompting struggled to keep the Agent in full check, the MCP tools became a significant source of improvement. The MCP tools forced it further to fall in-line with the developed constraints and exact direction to follow. The results became more consistent and relevant with each iteration as context increased as well.

Automation Enables Scale - To be able to scale manual testing at this level with large codebases would be impractical at best and nearly impossible at worst. With time and resources constraints, it us not feasible to even achieve half of what was provided in minutes to an hour worth of work by the AI Agent. This automated and systematic approach is proved effective as a repeatable workflow that can be further implemented.

Integration Complexity - All wasn't automatic or easy for that matter, it was more time consuming to get the correct and compatible environment for the multiple tools to function cohesively. Reading and understanding the codebase was important to ensure compatibility with tools, versioning, and the approach of integration and testing. The upfront time and planning paid off when the AI agent can focus on test generation and not get hung up on inconsistent tool responses.

C. Future Enhancement Recommendations

Based on this experience, some areas do present opportunities for enhancement:

Historical Coverage Tracking - It would useful to provide a storage solution for these iterations and at minimum for the analysis reports, coverage reports, and to track or trace progress beyond Git commit history.

Integration with External Tools - As other tools begin to develop further, implementing and connecting external tools to aid in analyzing code for more detailed code analysis could help in providing the Agent with approaches to solving coverage issues while relieving some load from its already demanding processes.

Automated Code Refactoring - With oversight, an Agent that is given permissions and guidance to correct source code and re-test for functionality would tie in the rest of the development life cycle.

Multi-Language Support - This setup was very much tailored to a codebase, but extending to other environments, languages, and versions of dependencies would make this scalable and functional.

IV. Conclusion

In the end, the project was a successful demonstration of an effective AI assisted test generation option. With the use of MCP tools over several phases of development, and many more iterations of prompting, the comprehensive testing framework achieved near 100% coverage reaching between 93% and 97% coverage across the codebase. Some key achievements were the integration of Git versioning control and Continuous Integration practices, effective analysis tools developed, and seamless testing integration.

The lessons learned highlight both the promise and limitations of current AI-assisted development tools. While AI significantly accelerates development and enables systematic approaches to large-scale tasks, human oversight and iterative refinement remain essential for quality outcomes.

Future work should focus on enhancing the intelligence of automated tools, expanding integration capabilities, and developing more sophisticated analysis techniques. The foundation established in this project provides a solid base for such enhancements.

The project did also have lessons of limitations, highlighting the need for precise guard rails, direction, and understanding the capabilities of current tools. All in all, this is foundation built to provide a strong base for further enhancements that can be made.

References

[1] FastMCP Documentation, GitHub. Available: https://github.com/jlowin/fastmcp

[2] JaCoCo Documentation. Available: https://www.jacoco.org/jacoco/trunk/doc/

[3] Model Context Protocol Specification. Available: https://modelcontextprotocol.io