Reflection on an AI-Assisted Software Testing Agent using the Model Context Protocol

Simran Kaur
Department of Computer Science
DePaul University
Chicago, USA
simikaur615@gmail.com

***Abstract*—This project explored the use of the Model Context Protocol (MCP) to build an AI-assisted software testing agent capable of generating JUnit tests, helping in code analysis, and automating Git operations. A custom MCP server was implemented to integrate large language model (LLM) reasoning with a Java codebase. This enables automated test generation workflows. Due to model quota constraints, full test improvement and code coverage wasn't implemented. However, the project successfully demonstrated the architecture, automation flow, and practical challenges of LLM-driven software engineering.**

***Index Terms*—MCP, software agents, automated testing, large language models, JUnit, AI-assisted development**

I. Introduction

AI-driven software engineering tools continue to transform development workflows by automating code generation, testing, and reasoning tasks. The goal of this project was to design and evaluate an MCP-connected agent that can generate Java unit tests, interact with the local environment, and incrementally improve coverage using LLM-based guidance.

The MCP gives a well-defined interface for using tools, accessing files, and command execution. This allows a tight loop between developer tools and model reasoning. This project investigates how well such an architecture supports automated test generation in a real Java project.

II. Methodology

The solution has three main parts: (1) a Python-based MCP server, (2) a Java Maven project acting as the testing target, and (3) a tester prompt file dedicated in guiding and ensuring systematic testing/reproducible measures.

A. MCP Server Implementation

The server was implemented using `fastmcp` and exposed tools for:

- Parsing Java source files to identify public methods
- Generating JUnit test classes programmatically
- Writing generated tests into `src/test/java/generated_tests`
- Performing Git operations such as add, commit, push, and pull requests

This architecture enabled natural-language commands in VS Code to trigger tool executions.

B. Agent Behavior and Prompting

A structured prompt (`tester.prompt.md`) specified:

- The agent's role as a test generator
- Expected test patterns (assertions, edge cases, negative tests)
- Requirements to summarize coverage results
- The iterative refinement loop planned for future work

This provided consistency across test generation attempts.

C. Test Generation Workflow

The intended workflow was:

1) Parse Java methods from source
2) Generate JUnit test classes
3) Run `mvn test` and collect coverage
4) Request improved tests based on missed branches or mutations

Only the first two steps were fully executed due to token limitations.

III. Results and Discussion

A. Test Generation Outcome

The MCP agent successfully produced JUnit test files with:

- Valid class definitions
- Test scaffolding for public methods
- Basic assertion logic

However, running `mvn test` revealed compilation errors, typically caused by:

- Incorrect constructor calls
- Missing imports or package paths
- Semantic misunderstandings of method behavior

These issues prevented obtaining meaningful JaCoCo coverage metrics.

B. Limitations from Model Quotas

A major constraint was the exhaustion of API quota during debugging iterations. Because the generated tests required multiple refinement passes, the lack of remaining queries halted the improvement cycle. As a result, the final system could not demonstrate coverage gains, though the infrastructure to support it was implemented.

C. Insights from AI-Assisted Development

Even without complete coverage results, several insights were observed:

- LLMs excel at structural code generation but frequently misinterpret object construction patterns without context.
- Prompt clarity significantly affects test validity.
- MCP provides a powerful mechanism to connect LLM reasoning with local development tools.
- AI-generated tests require human oversight to correct subtle logic errors.

D. Git Automation Success

The Git-related tools functioned reliably. The agent was able to:

- Stage all generated files
- Create meaningful commit messages
- Push changes to remote repositories

This demonstrates that MCP can effectively automate parts of the development workflow beyond test generation.

IV. Conclusion and Future Work

This project developed a functional MCP-based testing agent capable of generating Java unit tests and automating version control. While model quota and compilation issues prevented full coverage analysis, the system architecture is prepared for iterative improvement cycles.

Future enhancements may include:

- Automatic repair of failing tests
- Integration with JaCoCo or PIT for mutation-guided test generation
- Improved parsing and semantic understanding of Java types
- Multi-agent setups for combined static analysis and test synthesis

Overall, this project demonstrates both the potential and current limitations of AI-assisted software testing using MCP-based automation.

References

References

[1] Anthropic, "Model Context Protocol," 2024.
[2] EclEmma, "JaCoCo Java Code Coverage Library," 2023.
[3] PIT Mutation Testing, "Overview of Mutation Testing for Java," 2023.