Final Project on Software Agents

Using LLMs for software debugging and testing.

1st Tyler Enczur
School of Computing & Digital Media, DePaul University
Chicago, United States of America
tenczur@depaul.edu

***Abstract*—This project explores the design and implementation of tools using the Model Context Protocol (MCP) to improve the utility and practicality of LLMs in software testing. These tools target Java Maven projects using JaCoCo test reports, Git version control, and VS Code integration to automatically generate, execute, and refine JUnit test cases for real-world Java codebases. In conjunction with these tools, an LLM is theoretically capable of analyzing source code, identifying coverage gaps, and iteratively improving test quality through an autonomous feedback loop. Furthermore, the project incorporates Git-based workflow tools that enable automatic commits, pushes, and pull request creation based on coverage improvements.**

I. Introduction

Ever since the explosion in popularity and funding for the development of Large Language Models (LLMs), their applications to improvements in productivity in a variety of sectors have been practically unending. In the case of software development, LLMS have the potential to significantly improve productivity though the automation of software testing and debugging. The Model Context Protocol (MCP) offers a language-agnostic communication layer that provides LLMs access to pre-written routines that allow for theoretically complete autonomy in a variety of tasks. This project report documents the design, implementation, and evaluation of an MCP-based testing agent that:

- Generates JUnit tests from Java source code,
- Executes those tests against a Maven project,
- Parses JaCoCo coverage reports to identify gaps,
- Iteratively refines test cases, and
- Automates Git operations (commit, push, PR) based on coverage metrics.

The goal is to demonstrate that an LLM can handle the workflow loop of code analysis, test generation, and version-control automation.

II. Methodology

Overall, the development of these tools was split into three main categories: test coverage optimization with debugging, git version control, and code reviewing for general style and semantics. Given this project primarily targeted Java Maven projects, test coverages were measured with JaCoCo and code reviews were performed with PMD for general semantic issues, and Checkstyle for general style guidelines. Furthermore, the git version control tools targeted GitHub repositories. These tools were used in conjunction with a Github Agent prompt: `tester.prompt.md`. This file denotes the list of tools available for access, namely the VSCode file operation tools and the custom MCP tools implemented as previously outlines. It also outlines the procedures and boundary conditions for the Agent to act, such as when to call certain tools, how to handle different output types, git operation procedures, etc. This prompt file largely determines the overall behavior and use of the provided MCP tools by the testing agent.

A. Test Coverage Optimization & Debugging

For the optimization of test coverage, I implemented the tools `runJacocoTestReport()` and `getCoverageForClass()`. The `runJacocoTestReport()` method runs the jacoco report and stores a parsed dictionary of the line and branch coverages for each class. The parsing is performed by an externally defined Java application. The `getCoverageForClass()` method simply retrieves the stored coverage for a given class.

For debugging, I implemented another Java parsing application that reads surefire-reports to parse error messages for failing maven tests. The tool `runMavenTests()` runs the maven tests and returns the parsed error messages and the methods they occur in.

B. Git Version Control

In accordance with the project specification, the following tools were implemented for git version control: `git_add_all`, `git_commit`, `git_push`, `git_pull_request`, and `git_status`. Each of these tools made use of `GitPython` and/or `PythonGithub` for git commands or GitHub interaction.

C. Code Reviewing Tools

For the project extension, I implemented another external Java application that runs PMD and Checkstyle reports on a source code directory, and parses the results into a list of issues that are then returned to the caller. These issues can then be reviewed and fixed by the Agent.

III. Results

After using these tools in conjunction with a curated agent prompt, LLMs such as GPT-5 Mini were able to relatively consistently produce test classes and bug fixes in the provided Java codebase. These improvements were made near

completely autonomously, only with occasional ”continue” or ”proceed” prompts during pauses for user input. Therefore. the results confirm that an MCP-powered LLM can semi-autonomously improve test coverage in a real Java codebase. My main observations are as follows:

- Although LLMs initially typically have near complete coherence in responses and actions, as the context grows, they tend to become less productive and require more frequent intervention.
- Git automation greatly reduced the technical debt of LLM usage. In cases of significant hallucination or misapplied solutions, having a commit history to revert changes greatly reduced manual intervention work.
- Initial coverage improvements are incredibly rapid (from 0% to 80%); however, reaching near complete coverage takes significantly more time and increased the frequency of manual intervention.

IV. Discussion

Overall, I believe this project has affirmed by initial stance upon LLM usage in a software development context. Although already incredibly useful in general information retrieval and simple debugging, integration with MCP tools and agent prompts significantly improve predictability, accuracy, and consistentcy in output over wider contexts and in more complex environments. However, my results to indicate that they are not yet able to act completely autonomously without oversight or correction. As a tool, like any other it has its limits. It frequently hallucinates or miscalculates solutions to problems that may otherwise be simple for a human debugger to recognize and solve. Although it works with great speed, it can often hang at larger contexts and loses coherence without manual intervention or affirmation of its purpose.