SE333 — Final Project on Software Agents (10%)

Background:

This assignment represents the cutting edge of AI-assisted software development. Embrace the learning process and document your learning process. Everyone should be able to get 10/10 on this project.

- Implement and configure Model Context Protocol (MCP) servers for development automation
- Integrate AI-powered development agents with modern toolchains
- Develop custom automation tools for software development workflows
- Apply software engineering principles to real-world development scenarios

Project Overview:

You will build an intelligent agent using the Model Context Protocol (MCP) that automatically generates, executes, and iterates on test cases to achieve maximum code coverage. This project combines modern AI-assisted development with traditional software engineering practices.

- **Deadline:** November 11th, 2025
- **Presentation (demo)**: November 12th, 2025
- **Team Size**: Individual or up to **3 people**
- **Programming Languages**: Python, Java (Maven projects)
- **Tools**: VS Code, Node.js, Git, Maven, JaCoCo

Phase 1: Environment Setup & Basic MCP Integration

Prerequisites

- **VS Code** (latest version) with Chat view

- **Node.js 18+** (LTS recommended)
- **Git** and active GitHub account
- **Java 11+** and **Maven 3.6+**

Deliverables

1. **Development Environment Configuration**
   - Install and configure `uv` package manager
   - Set up Python virtual environment with MCP dependencies
   - Configure VS Code with MCP server integration
2. **Basic MCP Server Implementation**

```
# Setup commands
uv init
uv venv
source .venv/bin/activate  # On Windows: .venv\Scripts\activate
uv add mcp[cli] httpx fastmcp
```

3. **First MCP Tool (calculator MCP)**
   a. Visit fastmcp on GitHub.
   b. Create a new `server.py` file (or update your existing one).
   c. Add the following snippet at the bottom of the file:

```
if __name__ == "__main__":
    mcp.run(transport="sse")
```

   d. Starter your server: In your terminal:

```
python server.py
```

   e. Connect the MCP Server to VS Code:
      - Press `CTRL+SHIFT+P` → search for `MCP: Add Server`.

- Paste your MCP server URL (`http://...`).
- Give it a name.
- Press **Enter.**
- Confirm the new tool is listed in your VS Code Chat view.

f. Test the tool

- In the Chat prompt, type:

```
what is 1+2 // The tool should return the correct result.
```

g. Enable YOLO Mode (Auto-Approve)

- Press **CTRL+SHIFT+P** → search for **Chat: Settings**.
- Enable **Auto-Approve**.
- Make sure all tools are now highlighted for use.

h. If `mvn` does not run automatically, you can fix it:

- Press **CTRL+ALT+P**.
- Open **Auto-Approve Settings**.
- Add the entry:

```
mvn test
```

h. Add prompt for your agent:

- Inside your root folder in your repo, create a folder `.github\prompts`
- Add a new file called `tester.prompt.md`. The file is basically your prompt file. It has the following markdown format below:

```
---
mode: "agent"
tools: [`tool1`]
description: "description of the tool`
model: 'Gpt-5 mini'
```

```
---

## Follow instruction below: ##
1. Do X
2. Do Y
```

- Write your custom agent prompt inside this file, following the provided format.

Phase 2: Core Testing Agent Development

Objectives

Develop the foundation of your testing agent with Maven integration and basic test generation capabilities.

Deliverables

1. **Maven Project Integration**
   - Given an example project in Maven project
     - Configure JaCoCo plugin for coverage reporting
     - Set up automated test execution pipeline
2. **Test Generation MCP Tool**
   - Create MCP tool that analyzes Java source code
   - Generate JUnit test cases based on method signatures
   - Implement basic test execution and result parsing
3. **Coverage Analysis Tool**
   - Parse JaCoCo XML reports
   - Identify uncovered code segments
   - Generate coverage improvement recommendations

Phase 3: Git Automation Tools

Objectives

Develop MCP tools for automated version control workflows integrated with your testing agent.

Required Git MCP Tools

1. `git_status()`
    - Return clean status, staged changes, conflicts
2. `git_add_all()`
    - Stage all changes with intelligent filtering
    - Exclude build artifacts and temporary files
    - Confirm staging success
3. `git_commit(message)`
    - Automated commit with standardized messages
    - Include coverage statistics in commit messages
4. `git_push(remote="origin")`
    - Push to remote with upstream configuration
    - Handle authentication through existing credential helpers
5. `git_pull_request(base="main", title, body)`
    - Create a pull request against the specified base branch (default: `main`).
    - Use standardized templates for the **title** and **body**.
    - Automatically include metadata such as test coverage improvements or bug fixes.
    - Return the pull request URL for tracking.

Integration Requirements

- Git tools should work seamlessly with testing workflow
- Automatic commits when coverage thresholds are met
- Branch protection for main/master branches
- Integration with CI/CD pipeline concepts

Phase 4: Intelligent Test Iteration

Objectives

Implement the iterative improvement cycle that automatically enhances test coverage through multiple generations.

Requirement

- You will test your tool on a real-life project with test cases, hidden bug and even test failures
- Go to D2L → Final Project → codebase
  - This is the codebase in which you will test your code
- Note, everytime you make improvements, you must commit your code to Github.

Deliverables

1. **Agent Prompt Engineering**
   - Create `.github/prompts/tester.prompt.md`
   - Define agent behavior for test generation and iteration
   - Implement feedback loop based on coverage results
2. **Automated Test Improvement**
   - Implement automatic test enhancement based on coverage gaps
   - Handle test failures with debugging and fix generation
   - Track coverage improvement over iterations
3. **Fix the bug**
   - The code has a bug, if your test exposes the bug, you must use agent to fix it
4. **Quality Metrics Dashboard**
   - Generate comprehensive coverage reports that improves over time
   - Track test quality metrics (assertions per test, edge case coverage, bug fixes) over time.
     - For example, for each improvement in coverage or bug fixes, you need to commit onto Github autonomously using agent.

Phase 5: Creative Extensions

Objectives

Design and implement **two more** MCP tools that extend the capabilities of your testing agent.

Requirements

Each extension must:

- Address a real software development challenge
- Integrate with your existing MCP ecosystem
- Include comprehensive documentation and examples
- Demonstrate measurable value to development workflow

Suggested Extensions

1. **Specification-Based Testing Generator**
    - Implement boundary value analysis
    - Generate equivalence class test cases
2. **AI Code Review Agent**
    - Static analysis integration (SpotBugs, PMD)
    - Code smell detection and refactoring suggestions
    - Security vulnerability scanning (CodeQL?)
    - Style guide enforcement with auto-fix

Alternative Ideas

Students may propose their own extensions with instructor approval.

- Problem statement and motivation
- Technical approach and implementation plan
- Success metrics and evaluation criteria

Final Deliverables & Presentation

Documentation Requirements

1. **Technical Documentation in README.md**
    - Complete MCP tool API documentation
    - Installation and configuration guide
    - Troubleshooting and FAQ section
2. **Reflection Report** (2 pages max)
    - Research writing:
        - Should have abstract, introduction, methodology, result
        - Result should discuss:
            - Analysis of coverage improvement patterns
            - Lessons learned about AI-assisted development
            - Future enhancement recommendations
    - Your report should be written in **latex**:
        - Refer to this latex link hosted on overleaf (link).
3. **Demonstration Video** (5 minutes)
    - Project overview and objectives:
    - Live demonstration of complete workflow
    - Coverage improvement showcase
    - Demo of most impressive features
4. Final demo (5 minutes)
    - For synchronous students: Show video live in class, followed by Q&A.
    - For asynchronous students: Submit a recorded demo, which will be played for the class. If feasible, async students may participate in Q&A asynchronously (e.g., via discussion board).

Final Demo (10 minutes - in front everyone)

- I will be playing the demo in front everyone
- Followed by Q&A session (if feasible for Async)

Resources and Support

Getting Help

- Weekly sessions for technical troubleshooting and discussion

---

Submission Guidelines

Weekly Check-ins

Submit brief progress reports (1-2 paragraphs) via course management system documenting:

- We will probably have some time in class to work on this every class
- Current week's accomplishments
- Challenges encountered and solutions
- Plans for following week
- Any blockers requiring instructor assistance

Final Submission

All deliverables must be submitted via GitHub repository with:

- Complete source code with clear organization
- Comprehensive README with setup instructions
- Documentation in markdown format
- Video demonstration uploaded to course platform
- Reflection report as PDF

Late Policy: NO late policy