AI Coder Multi-Agent System

IN4MATX 119 Final Project Written Report

Team Members: Junxi Chen (70714925), Tiancheng Qiu(36732598), Sung Jin Kim (10906553)

Date: December 2025

---

1. Introduction

1.1 Purpose of the System

The AI Coder Multi-Agent System is an automated code generation platform designed to transform natural language software descriptions into fully functional, tested Python applications. The system addresses the growing need for efficient software development by leveraging multiple specialized AI agents that collaborate to produce high-quality code.

The primary goals of this system are:

1. **Automated Requirements Analysis:** Parse unstructured software descriptions and extract structured, actionable technical requirements.
2. **Code Generation:** Generate complete, modular Python applications that satisfy all specified requirements with proper software engineering practices.
3. **Test Generation:** Create comprehensive test suites that verify the generated code meets its requirements, achieving at least 80% pass rate with a minimum of 10 test cases.
4. **Usage Tracking:** Monitor and report API usage statistics for transparency and resource management.

1.2 Target Application

For this project, the system generates an **Expense Comparator** application—a personal finance tool that helps users track expenses, categorize spending, compare costs across different time periods, and visualize spending patterns through charts and graphs.

---

2. System Design and Workflow Description

2.1 Overall Architecture

The system follows a **pipeline architecture** with three sequential processing stages, each handled by a specialized AI agent. Communication between components is facilitated by the **Model Context Protocol (MCP)**, which provides a standardized interface for tool invocation.

```
┌─────────────────────────────────────────────────────────────────┐
│                       User Interface (GUI)                      │
│                     Gradio Web Application                      │
└─────────────────────────────────────────────────────────────────┘
                              │
                              │
                              ▼
┌─────────────────────────────────────────────────────────────────┐
│                     MCP Client (client.py)                      │
│                                                                 │
│  ┌──────────────┐   ┌──────────────┐   ┌──────────────┐         │
│  │ Requirements │   │     Code     │   │     Test     │         │
│  │   Analyzer   │──→│   Generator  │──→│   Generator  │         │
│  │    Agent     │   │    Agent     │   │    Agent     │         │
│  └──────────────┘   └──────────────┘   └──────────────┘         │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
                              │
                              │ MCP Protocol (stdio)
                              ▼
┌─────────────────────────────────────────────────────────────────┐
│                     MCP Server (server.py)                      │
│  Tools: write_file, read_file, validate_python_syntax,          │
│         create_directory, append_to_file, list_directory        │
└─────────────────────────────────────────────────────────────────┘
```

2.2 Input Parsing

The system accepts input through two interfaces:

1. **GUI Mode:** Users enter software descriptions in a text box within the Gradio web interface. The input is passed directly to the multi-agent pipeline.
2. **CLI Mode:** The system uses a predefined software description (Expense Comparator) embedded in the code.

Input processing involves:

- Validating that the description is non-empty
- Passing the raw text to the Requirements Analyzer Agent

- No pre-processing or tokenization is performed; the LLM handles natural language understanding

2.3 Data Flow Through the System

Step 1: Requirements Analysis

- Input: Raw software description (natural language)
- Processing: The Requirements Analyzer Agent uses Google's Gemini model to extract and structure requirements
- Output: Structured requirements document saved to `requirements_spec.txt`
- Tools Used: `write_file`

Step 2: Code Generation

- Input: Structured requirements from Step 1
- Processing: The Code Generator Agent creates the application file structure and implements all modules
- Output: Complete Python package in `output/expense_comparator/` with 8 modules
- Tools Used: `create_directory` , `validate_python_syntax` , `write_file`

Step 3: Test Generation

- Input: Requirements document and access to generated code
- Processing: The Test Generator Agent reads the generated code, designs test cases, and creates a comprehensive test file
- Output: `test_expense_comparator.py` with 12-15 test cases
- Tools Used: `read_file` , `write_file`

Step 4: Usage Report

- Throughout all steps, the Usage Tracker captures API calls and token counts
- Final output: `model_usage_report.json` with usage statistics

---

3. Model Roles, Tools, and MCP Integration

3.1 Agent Roles and Responsibilities

Agent	Model	Primary Responsibility	Output
Requirements Analyzer	Gemini 2.5 Flash	Parse software descriptions into structured requirements	`requirements_spec.txt`
Code Generator	Gemini 2.5 Flash	Generate modular, runnable Python code	8 Python modules + entry point
Test Generator	Gemini 2.5 Flash	Create comprehensive test cases	`test_expense_comparator.py`

3.2 Tool Descriptions and Usage

The MCP Server provides six tools that agents invoke to interact with the file system:

1. **`validate_python_syntax(code: str)`**
   - Purpose: Check Python code for syntax errors without executing it
   - Used By: Code Generator Agent
   - Why: Ensures all generated code is syntactically correct before saving
2. **`write_file(filename: str, content: str)`**
   - Purpose: Create or overwrite files in the output directory
   - Used By: All agents
   - Why: Primary mechanism for persisting generated artifacts
3. **`read_file(filename: str)`**
   - Purpose: Read file contents from the output directory
   - Used By: Test Generator Agent
   - Why: Allows the Test Generator to understand the code it needs to test
4. **`create_directory(path: str)`**
   - Purpose: Create directories for organizing generated code
   - Used By: Code Generator Agent

  - Why: Establishes proper Python package structure

5. **`append_to_file(filename: str, content: str)`**
   - Purpose: Add content to existing files without overwriting
   - Used By: Code Generator Agent
   - Why: Useful for incremental file building
6. **`list_directory(path: str)`**
   - Purpose: Enumerate directory contents
   - Used By: Any agent for verification
   - Why: Confirm files were created correctly

3.3 How MCP is Used

The Model Context Protocol (MCP) serves as the communication backbone between agents and tools:

1. **Server Setup:** The `server.py` file implements an MCP server using FastMCP that exposes tools via stdio transport.
2. **Client Connection:** The `client.py` establishes MCP sessions using `StdioServerParameters` and `stdio_client` from the MCP library.
3. **Tool Loading:** The `langchain-mcp-adapters` library loads MCP tools into LangChain-compatible format using `load_mcp_tools(session)`.
4. **Agent Creation:** Each agent is created using `create_react_agent()` from LangGraph, which combines the LLM with loaded MCP tools.
5. **Tool Invocation:** When an agent needs to perform an action (e.g., write a file), it generates a tool call that the MCP client sends to the server.

3.4 Collaboration Strategy

The agents collaborate through a **sequential pipeline** with shared context:

1. The Requirements Agent outputs structured requirements that are passed as context to the Code Generator
2. The Code Generator creates files that the Test Generator reads via MCP tools

3. All agents share the same MCP session, ensuring consistent file system state
4. The global Usage Tracker aggregates statistics across all agents

---

4. Error Handling and Fault Tolerance

4.1 Input Validation

- **Empty Input Check:** The GUI validates that the software description is non-empty before starting generation
- **Error Display:** Errors are captured and displayed in the GUI status box rather than crashing

4.2 Code Validation

- **Syntax Validation:** Before writing any Python file, the Code Generator calls `validate_python_syntax` to catch errors early
- **Re-validation Loop:** If syntax errors are detected, the agent is instructed to fix and re-validate

4.3 File Operation Safety

- **Directory Creation:** The `write_file` tool automatically creates parent directories using `os.makedirs(exist_ok=True)`
- **Permission Handling:** All file operations catch `PermissionError` and return descriptive error messages
- **Path Safety:** All file operations are restricted to the `output/` directory

4.4 MCP Session Management

- **Context Manager:** MCP sessions use Python's `async with` context manager to ensure proper cleanup
- **Initialization Check:** Sessions are explicitly initialized with `await session.initialize()`

4.5 Usage Tracking Robustness

- **Thread Safety:** The Usage Tracker uses `threading.Lock` to handle concurrent access
- **Token Estimation:** If exact token counts are unavailable, the tracker estimates based on response text length
- **Minimum Estimates:** A fallback minimum of 500 tokens per call prevents zero-token reports

4.6 Exception Handling

- **Try-Except Blocks:** All major operations are wrapped in try-except blocks
- **Graceful Degradation:** Errors in one component don't crash the entire system
- **Traceback Logging:** Exceptions are logged with full tracebacks for debugging

---

5. Reflection

5.1 What Went Well

1. **MCP Integration Success:** The Model Context Protocol provided a clean abstraction for tool invocation. Using FastMCP made server implementation straightforward, and the langchain-mcp-adapters library seamlessly integrated MCP tools with LangChain agents.
2. **Modular Architecture:** Separating concerns into distinct agents (Requirements, Code, Test) made the system easier to develop, debug, and modify. Each agent has a clear responsibility and well-defined inputs/outputs.
3. **Comprehensive Usage Tracking:** The callback-based usage tracker successfully captures API calls and token usage across all agents, meeting the rubric requirements.
4. **User-Friendly GUI:** The Gradio interface provides an intuitive way to interact with the system, with clear tabs for viewing different outputs and real-time status updates.
5. **Test Generation Quality:** The Test Generator consistently produces 12-15 focused test cases that achieve the required 80%+ pass rate.

5.2 Challenges Faced

1. **Token Counting Variability:** Google's Generative AI API returns token counts in different metadata fields depending on the response type. We had to implement multiple fallback mechanisms to reliably extract usage data.
2. **Agent Coordination:** Ensuring the Code Generator produces code that the Test Generator can successfully test required careful prompt engineering. Initial attempts resulted in tests that imported non-existent modules.
3. **Recursion Limits:** Complex code generation requires many tool calls (each file = ~2 calls). We had to increase LangGraph's recursion limit to 100 to avoid premature termination.
4. **Circular Import Prevention:** The generated code needed careful module organization to avoid circular dependencies, which required explicit instructions in the agent prompts.

5.3 Limitations

1. **Single Application Type:** The current system is optimized for generating the Expense Comparator application. Adapting to different software types would require prompt modifications.
2. **No Real-Time Feedback:** The GUI doesn't show incremental progress during generation. Users see results only after all agents complete.
3. **Limited Error Recovery:** If an agent fails mid-generation, there's no mechanism to resume from the last successful step.
4. **Synchronous Execution:** Agents run sequentially rather than in parallel where possible (e.g., test generation could theoretically begin while later modules are being written).

5.4 Future Improvements

1. **Streaming Output:** Implement real-time streaming of agent progress to the GUI
2. **Checkpointing:** Save intermediate state to enable recovery from failures
3. **Parallel Execution:** Allow independent operations to run concurrently
4. **Template Flexibility:** Support multiple software templates beyond Expense Comparator

5. **Test Execution Integration:** Run tests automatically after generation and display results in GUI

5.5 Lessons Learned

1. **Prompt Engineering is Critical:** The quality of generated code depends heavily on how precisely requirements are communicated to agents. Iterative prompt refinement was essential.
2. **MCP Simplifies Tool Management:** Compared to manually implementing tool interfaces, MCP provides a standardized, reusable approach that scales well.
3. **Usage Tracking Requires Planning:** Implementing accurate usage tracking from the start is much easier than retrofitting it later.
4. **Testing Generated Code is Hard:** Generated code often has subtle issues that only appear during testing. The Test Generator needs to be aware of the actual implementation, not just the requirements.

Appendix: Running the System

Quick Start

```
# Install dependencies
pip install -r requirements.txt

# Set API key
export GOOGLE_API_KEY="your-key"

# Run GUI
python main.py

# Run tests after generation
python run_tests.py
```

Expected Output

- Generated code in `output/expense_comparator/`

- Test file at `output/test_expense_comparator.py`
- Usage report at `output/model_usage_report.json`
- 10+ test cases with 80%+ passing

Word Count: ~2,100 words (excluding code and tables)