Final Project Report: AI Coder System

i. Introduction

The purpose of this project was to learn how to use AI to automate the software development process. The goal was to transform natural language requirements into working Python applications. The "AI Coder" acts as an intelligent coding assistant that not only generates code but also plans the architecture, writes comprehensive test suites, and iteratively fixes its own errors.

The project utilizes a multi-agent architecture (MCP) that mimics a real-world development team where distinct roles (Architect, Developer, QA Engineer) collaborate to produce high-quality software. The goal is to streamline the coding workflow, reduce manual effort in setting up boilerplate code and tests.

ii. System Design and Workflow Description

The system is designed as a pipeline of specialized agents orchestrated by a central controller. The workflow follows a linear progression with a feedback loop for quality assurance.

Input Parsing

The entry point is a Gradio-based web interface (app.py). The system accepts raw text input describing the software requirements. This input is validated to ensure it is not empty before being passed to the orchestrator.

Data Flow Steps

1. **Initialization**: The orchestrator.py initializes the MCPClient (for model communication) and the UsageTracker. It configures the three specialist agents: Planner, Coder, and Tester.
2. **Planning Phase**: The PlannerAgent receives the raw requirements. It analyzes them to identify main modules, responsibilities, and edge cases, producing a structured development plan.
3. **Coding Phase**: The CoderAgent takes both the original requirements and the structured plan. It generates the implementation code (Python), ensuring it includes docstrings and follows the plan.
    a. Internal Loop: The generated code undergoes syntax validation. If syntax errors are found, the agent automatically prompts the model to fix them with up to 3 retries.
4. **Testing Phase**: The TesterAgent receives the requirements and the generated code. It produces a test suite (using pytest) covering at least 10 test cases.
    a. Internal Loop: Similar to the Coder, the Tester validates the syntax of the generated tests and retries if necessary.
5. **Artifact Persistence**: The generated application code and test files are saved to the generated/ directory.
6. **Self-Healing Loop**: The orchestrator executes the generated tests against the generated code.
    a. If tests pass: The pipeline completes successfully.
    b. If tests fail: The system enters a "self-healing" loop. The error output is captured and sent back to the CoderAgent along with the code and

requirements. The agent attempts to fix the code to resolve the errors. This process repeats up to 3 times.

7. **Output**: The final code, tests, usage report, and execution instructions are returned to the UI for the user.

iii. Model Roles and Tools

The system utilizes a "Specialist Architecture" where different LLM models are assigned roles based on their strengths.

Roles and Responsibilities

1. PlannerAgent (Architect)
   - **Model**: gemini-2.0-flash-thinking-exp
   - **Role**: Responsible for high-level reasoning and architectural design. It breaks down complex requirements into manageable components and identifies critical test scenarios.
   - **Delegation**: It does not write code; it provides the "blueprint" for the Coder.
2. CoderAgent (Developer)
   - **Model**: gemini-2.5-pro
   - **Role**: Responsible for high-quality code generation. It translates the plan into executable Python code. It is also responsible for fixing bugs identified during testing.
   - **Delegation**: It focuses solely on implementation logic, relying on the Planner for structure and the Tester for verification.
3. TesterAgent (QA Engineer)
   - **Model**: gemini-2.0-flash
   - **Role**: Responsible for high-volume test generation. It creates unit tests to verify the code's correctness.
   - **Delegation**: It ensures the implementation meets the requirements without modifying the application code itself.

Tools and MCP Usage

The agents interact with the models via a custom Model Context Protocol (MCP) Client (mcp_client.py).

MCP: The MCPClient acts as a standardized abstraction layer. It handles authentication, message formatting (converting system/user/assistant roles to the model's expected format), and API communication. This allows the agents to be independent of the underlying API details.

Internal Tools: Agents also use utility functions like validate_python_syntax to check their output before passing it downstream, ensuring that syntax errors are caught early.

iv. Error Handling

The system implements multiple layers of error handling to ensure fault tolerance:

1. **Syntax Validation Loops**: Both CoderAgent and TesterAgent verify that their generated Python code is syntactically valid. If ast.parse() fails, the agent catches the error and prompts the model to correct the syntax, retrying up to 3 times.
2. **Self-Healing Mechanism**: The orchestrator runs the generated tests in a sandbox. If they fail, the system doesn't just crash or return broken code. Instead, it captures the

failure output and triggers a "fix" cycle, allowing the CoderAgent to iteratively repair the code.

3. **API Resilience**: The MCPClient includes robust handling for API errors, specifically ResourceExhausted (429) errors. It implements an exponential backoff strategy, waiting and retrying requests when rate limits are hit.
4. **Graceful Failure**: The entire pipeline is wrapped in try-except blocks. If a critical failure occurs (e.g., max retries exceeded, API down), the system logs the full stack trace and returns an error message to the UI instead of crashing the application.
5. **Logging**: Comprehensive logging is implemented throughout (logging_config.py), tracking every step of the pipeline, which aids in debugging and monitoring system health.

v. Reflection

Limitations

- **Context Window**: While the models have large context windows, extremely large or complex requirements might still lead to loss of detail or hallucinations.
- **Single-File Output**: Currently, the system generates a single Python file for the application. It does not support generating complex multi-file project structures.
- **Dependency Management**: The system assumes standard libraries or pre-installed packages. It does not dynamically generate a requirements.txt or install new packages for the generated code.
- **Limited User Control**: It is difficult to guide the code generation process or provide iterative feedback on the requirements once the pipeline starts. The user is essentially stuck with whatever the agent generates, as there is no mechanism to intervene or refine the output without restarting from scratch.

Challenges

- **Prompt Engineering**: Crafting prompts that consistently yield valid code and tests across different models required significant iteration. Ensuring the "Planner" output was useful to the "Coder" was a key challenge.
- **Flaky Tests**: Sometimes the TesterAgent generates tests that are incorrect or too strict, causing the self-healing loop to try and "fix" correct code.
- **API Rate Limits**: Managing the quota for the Gemini API was difficult during heavy testing, necessitating the implementation of the exponential backoff logic.

What Went Well

- **Specialist Architecture**: Assigning specific models to specific roles proved very effective. Using a "thinking" model for planning and a "pro" model for coding resulted in better structure and logic than using a single model for everything.
- **Self-Healing**: The ability of the system to read its own error messages and fix the code is a powerful feature. It significantly increased the success rate of generating runnable code.
- **Modular Design**: The separation of concerns (Orchestrator, Agents, MCP Client) makes the codebase easy to understand, extend, and debug.