Introduction

Our system is an AI-assisted “coder” that automatically generates backend code and tests for a small business application called Customer Feedback Tracker. The target application collects, stores, and analyzes customer feedback submitted through different forms and channels, such as email, social media, or generic web links, and supports querying and simple statistics over this data.

Instead of writing the application logic by hand, our project uses a multi-agent system built on top of the Model Context Protocol (MCP). Given a high-level description of the Customer Feedback Tracker through a graphical user interface, the agents collaborate to produce Python modules, a minimal interface for running the code, a test suite with at least ten test cases, and a JSON report summarizing model usage (number of calls and total tokens). The goal of the system is to demonstrate how MCP-based multi-agent workflows can turn natural language requirements into runnable, testable code.

System design and workflow

The main input to the system is the Markdown spec spec/customer_feedback_spec.md. When the orchestrator starts, it reads this file and sends its contents to the Requirements Agent via MCP. The agent converts the text into a structured JSON “AppSpec” describing entities, operations, and expected files. This JSON is then passed to the CodeGen and TestGen agents, which generate the source files and generated/tests/test_feedback.py. In this workflow, the spec is the single source of truth that all agents follow.

Inside the orchestrator, the flow for the pipeline is flow in: RequirementsAgent → CodegenAgent → TestgenAgent → ReviewerAgent. For the RequirementsAgent, it reads the raw spec text, calls call_model, and asks the model to return JSON with information like name, entities, endpoints and non-functional requirements. For the CodegenAgent, it takes the JSON spec and calls the call_model again, and asks the model content in JSON format that contains code files like models.py, repository.py, service.py and cli.py. For the TestgenAgent, it uses spec and a small summary of the code, calls call_model and asks for JSON with at least one test file. For TestgenAgent, it does a quick local check on the files, and the orchestrator writes all these files into the generated folder.

(ui starts here) The UI communicates with the backend through an API endpoint. When the user clicks “Generate Code & Tests,” the UI packages the requirement text into a JSON request and sends it to the backend. The backend runs the orchestrator pipeline, which invokes the Requirements Agent, Code Generation Agent, Test Generation Agent, and Reviewer in sequence. When processing is complete, the backend returns a JSON response containing generated code files, test files, and model usage statistics. The UI parses this response and renders the generated files and usage data in clearly labeled sections so that users can easily review the system output.

Model roles and tools

Module 3 is in charge of managing model usage tracking to measure efficiency and output analysis. The main component of that is the function call_model() in the usage_tracker.py. This is a wrapper for all the calls made from agents. In our current version, call_model() returns different JSON strings depending on the prompt. For example, if the prompt looks like the Requirements Agent (“You are a senior software requirements analyst”), it returns a JSON app spec. If it looks like the CodeGen agent,

it returns a JSON object with file names and code content. If it looks like the TestGen agent, it returns a JSON object with test files. After generating this response, we do a token approximation by counting the number of words in the prompt which are prompt tokens, and in the response which are completion tokens. We then record these in a shared ModelUsageTracker. The tracker keep in track of totalPromptTokens, numApiCalls, and totalCompletionTokens across the run and get_usage_stats() expose it as a dictionary similar to JSON that orchestrator can attach to with the final output.

Module 4 is responsible for integrating the user interface with the backend multi-agent system. The UI is designed to serve as a bridge between the user and the backend, ensuring that it does not modify, fabricate, or override the output of any model or agent. The frontend submits the requirements to the backend in a standardized JSON format and depends on a fixed response structure to display outputs correctly.

Error handling

Module 3 helps with making the system more error and fault tolerant because every call goes through call_model(). Since it's only going through one place, it makes it easier for us to figure out any mistakes and catch errors and faults. If the model returns a bad JSON or some fields are missing, we can check that before letting other agents continue. We also use a try and except in the call_model() so if anything unexpected happens, it won't crash and return a response that still records usage. These features together make the program more stable. Even when the model fails, the UI can still show the results, and this overall is more organized and prone to errors.

Reflection

For module 1 (Yifan Wu), I was responsible for writing and maintaining the specification for the Customer Feedback Tracker in *spec/customer_feedback_spec.md*. The main difficulty was finding a balance between being detailed enough for the agents to follow and keeping the document clear and readable. Early versions were either too vague (some fields or operations were ignored in generation) or too long and repetitive. By iterating—splitting the spec into structured sections (data model, operations, module layout, integration notes) and stating exact file names, function signatures, and constraints—I was able to reduce ambiguity. Running the pipeline locally and reviewing the automated feedback helped me refine unclear parts, and showed me how much the quality of AI-generated code depends on having a precise, well-organized specification. Beyond writing the specification itself, I also helped coordinate the team. I drafted our initial team plan, clarified which files and responsibilities belong to each module, and updated it as the implementation evolved. I also helped debug cross-module issues by running the full pipeline on my machine – verifying that orchestrator.main generated the expected files, that run_tests.py executed the pytest suite successfully, and that the UI correctly displayed the generated code, tests, and usage statistics. This collaborative work made it easier for the other modules to integrate their parts and for the group to reach an end-to-end working system before the deadline.

For module 2 (Richard Zhang), I was responsible for writing code for the pipeline work. Some of the pipeline works okay and some do not., The good part for it is the structure is clear and simple, UI or CLI feeds text into the orchestrator, the agents each have a job, and everything end up as normal Python files plus tests in generated/ folder. I am treating each agent like a small MCP tool with clear input and output JSON, which makes it easier to think about the design. The hard part for it is using a real LLM model in a strict way. In practice, the model output is not predictable: it sometimes returns a

clean JSON, sometimes it adds code fences, sometimes it adds extra text, and sometimes it changes the schema even when I tried to set config like JSON mode and response type . My agents are very strict and expect specific JSON shapes , so small changes in the model output break the json.loads and make the agent fail, which makes the whole pipeline fail. Because of this, I won't be able to implement a real LLM for the program in the final version. I could not guarantee a LLM will always return the answer in the same format, so this part did not go well, and I had to accept that limitation for this project.

For module 3 (Jerry Lechen Zhang), I was responsible for writing code that tracks the model usage during the whole generation process. I made and implemented a ModelUsageTracker class to count the API calls and the approximate token usage for the responses and the prompts. I also made the call_model() wrapper, which makes sure that all model calls are in one place, which makes it easier to track problems and errors. The challenge I faced was integrating the usage tracking with the orchestrator, and making sure that it is successfully displayed in the UI. I took a long time trying to figure out how to integrate the two, but eventually, I found the problem to be in api.py. where the part integrating data usage was returning null. I also wrote 10 test cases so that we make sure the program call meets assignment requirements and passes all or at least 8 test cases.

For module 4 (Cheng Chen), I was responsible for creating a UI that aims to achieve end-to-end integration without requiring changes to backend logic. The UI reliably sends requests, handles long asynchronous operations, and displays generated output consistently. A major challenge was dealing with unpredictable output formats from the backend while maintaining layout stability. Generated files vary in length and complexity, which requires careful handling to prevent UI overflow and formatting issues. Debugging asynchronous API behavior and backend timing issues also required extensive logging and testing. The UI still has some limitations, including that the UI currently does not support file downloads, syntax highlighting, and interactive code editing. The lack of these functions limits usability beyond demonstration purposes.