INF 119 Final Project Report

Group 17

i. Introduction: Explain the purpose of the system.

For our final project, we implemented a system that converts plain English requirements into a functional Python verb conjugation program, corresponding automated tests, and a web UI. The core purpose is to demonstrate how a coordinated multi-agent architecture using a structured message-passing protocol can leverage a large language model to automate major stages of software creation.

The system is designed to automate the entire path from natural-language requirements to design, code generation, and testing; to offer clear metrics on model usage and performance; and to demonstrate how modular agents can be inspected, executed, and refined.

ii. System design and workflow description: How does the system parse the input? How does data flow through your system, taking which steps?

The architecture is composed of three layers: a user interface, a set of specialized agents, and the Model Context Protocol (MCP).

The user submits the requirement in text through a Gradio UI. The UI sends the text to the Parser Agent via MCP. The Parser performs extraction, identifying requirements, constraints, and missing details. It generates a structured object. Rule based parsing is used first; the LLM is only invoked when clarification or interpretation is needed.

The system moves through a sequential pipeline. The user inputs requirements through the UI, which forwards them to the Parser Agent via MCP. The Parser produces a RequirementSpec, passed to the Design Agent to generate a DesignSpec outlining modules and file structure. The CodeGen Agent uses this design to generate Python source files, which are then sent to the Test Agent to produce pytest tests. Throughout the process, the Tracking Agent records all LLM calls and outputs a usage_report.json. Final

code and tests are saved with download links returned through the UI. All agents communicate exclusively through MCP.

iii. Model roles and tools: Explain the roles of each model, how responsibilities are delegated, and what tools they are using and why. How do you use MCP?

Each agent in the system has a defined responsibility that contributes to the overall pipeline. The Parser Agent interprets the user's natural language requirements and extracts structured information using rule-based logic, only relying on the LLM when clarification or disambiguation is needed. Once the requirements are formalized, the Design Agent transforms them into a high level "DesignSpec" that outlines the system's modules, interfaces, and file layout. This design document serves as the blueprint for downstream stages.

The CodeGen Agent is the main agent of the LLM. It converts the DesignSpec into Python source files by composing targeted prompts that request specific modules, logic implementations, and any necessary glue code. After the code generation, the Test Agent uses the LLM to create pytest test cases that evaluate correctness, edge behavior, and input validation. Alongside these agents, the Tracking Agent monitors every LLM call recording model names, token counts, latencies, and failures to produce a usage report. It does not generate code itself but ensures that the system remains measurable and auditable.

Several tools support these roles. Google's Gemini model provides the natural language reasoning and code generation capabilities needed for parsing, design, and test creation. Gradio offers an accessible UI for submitting requirements and inspecting results. Python and pytest serve as the foundation for the produced application and its tests. MCP manages typed messages, agent-specific queues, and delivery order, ensuring that each agent receives exactly the information it needs without tight coupling. This approach allows any agent to be modified, replaced, or extended without disrupting the rest of the system.

iv. Error handling: What did you do to make sure your system is fault-tolerant?

The system incorporates multiple layers of fault tolerance to prevent invalid outputs and ensure the pipeline remains stable even when upstream stages fail. Each agent validates the structure and completeness of its inputs by ensuring that required fields exist in both the RequirementSpec and DesignSpec and halts execution if essential information is missing. The CodeGen and Test Agents also perform basic syntax checks on all LLM generated code, attempting to parse or compile it before writing files to disk. When errors are detected, the agents retry with improved prompts or fall back to template-based code to preserve usability. All LLM calls include timeout and retry logic to account for network failures, while the UI provides progress updates to avoid situations where long operations appear frozen. Intermediate artifacts are written to disk so that the system can recover partial progress rather than restarting the entire pipeline. MCP's message logging further supports debugging by preserving a trace of all inter-agent communication. If external models become unavailable or API keys are missing, the system turns off peacefully by alerting the user and generating minimal stub outputs rather than crashing. Throughout the process, the Tracking Agent records failures, latencies, and token usage to provide visibility into model behavior and inform future tuning efforts.

v. Reflection: Discuss limitations, challenges, and what went well/didn't go well.

Developing this system highlighted both the strengths and weaknesses of an LLM-driven, multi-agent software workflow. The modular agent structure and MCP messaging made the pipeline easy to reason about, maintain, and debug. The LLM proved capable of generating workable application code, corresponding tests, and design documentation, demonstrating the capability of automating large portions of the software development lifecycle. The Gradio UI made interaction straightforward, and the usage-tracking infrastructure offered valuable insight into model performance and cost.

However, several challenges emerged. The most persistent difficulty was ensuring that LLM generated code was consistent and the syntax was valid, as even small prompt variations could lead to drift or incomplete outputs. API errors also required careful timeout handling and user-facing messaging. More broadly, the system's outputs remain prototypes rather than

production-ready artifacts; they lack comprehensive error handling, performance considerations, and security safeguards. The MCP implementation itself is intentionally simple and lacks distributed features and persistence across restarts which is something that could be an improvement.