Overall Flowchart of the Application:

1 graph TD

2 A[Start Application (main.py)] --> B{Load Environment Variables (.env)};

3 B --> C[Read User Story (from file)];

4 C --> D[Call get\_specification\_refinement(user\_story)];

5 D --> E{LangChain: Build QA Engineer Prompt};

6 E --> F{LangChain: Invoke Gemini Model (gemini-pro) with Structured Output

(SpecificationRefinement)};

7 F --> G[Receive Specification Refinement Output];

8 G --> H[Call get\_prioritization(user\_story)];

9 H --> I{LangChain: Build Product Owner Prompt};

10 I --> J{LangChain: Invoke Gemini Model (gemini-pro) with Structured Output

(Prioritization)};

11 J --> K[Receive Prioritization Output];

12 K --> L[Print Results];

13 L --> M[End];

Explanation of the Flow:

1. Start Application (`main.py`): The application begins execution from the main.py script.

2. Load Environment Variables: The load\_dotenv() function is called to load environment variables

(like GOOGLE\_API\_KEY) from the .env file. This ensures sensitive information is not

hardcoded.

3. Read User Story: The read\_file() utility function reads the content of the specified user

story file (e.g., user\_story.txt or a PDF).

4. Call `get\_specification\_refinement(user\_story)`: The main function then calls

get\_specification\_refinement from src/agent.py, passing the user story content.

5. LangChain: Build QA Engineer Prompt: Inside get\_specification\_refinement, LangChain's

ChatPromptTemplate is used to construct a prompt for the Gemini model. This prompt instructs

the model to act as a "meticulous QA Engineer" and identify ambiguities, contradictions,

missing edge cases, and non-functional requirements in the user story.

6. LangChain: Invoke Gemini Model (gemini-pro) with Structured Output: A ChatGoogleGenerativeAI

instance is created, specifying the gemini-pro model and passing the GOOGLE\_API\_KEY. The

with\_structured\_output(SpecificationRefinement) method is crucial here; it tells LangChain to

enforce the output of the Gemini model to conform to the SpecificationRefinement Pydantic

model, ensuring a consistent and parseable JSON-like structure. The chain.invoke() method

sends the prompt to the Gemini model.

7. Receive Specification Refinement Output: The structured output from the Gemini model, parsed

into a SpecificationRefinement object, is returned.

8. Call `get\_prioritization(user\_story)`: Similarly, the main function calls get\_prioritization,

also passing the original user story.

9. LangChain: Build Product Owner Prompt: Inside get\_prioritization, another ChatPromptTemplate

is used to create a prompt, instructing the Gemini model to act as an "experienced Product

Owner" and estimate the RICE score (Reach, Impact, Confidence, Effort) for the user story.

10. LangChain: Invoke Gemini Model (gemini-pro) with Structured Output: Another

ChatGoogleGenerativeAI instance is used, again with gemini-pro and the API key. This time,

with\_structured\_output(Prioritization) ensures the output conforms to the Prioritization

Pydantic model.

11. Receive Prioritization Output: The structured output for prioritization is returned.

12. Print Results: Finally, main.py prints both the specification refinement and prioritization

results to the console.

What about LangChain?

LangChain is a framework designed to simplify the development of applications powered by large

language models (LLMs). In this project, LangChain plays several key roles:

\* Prompt Management: It provides ChatPromptTemplate to easily define and manage the

conversational prompts sent to the LLM, separating the instructions from the user input.

\* LLM Integration: It offers a standardized interface (ChatGoogleGenerativeAI) to interact

with different LLMs (in this case, Google's Gemini models). This abstracts away the

complexities of direct API calls.

\* Structured Output: This is a powerful feature used here. By using

llm.with\_structured\_output(PydanticModel), LangChain ensures that the LLM's response is not

just free-form text but adheres to a predefined schema (the SpecificationRefinement and

Prioritization Pydantic models). This makes the LLM's output easily parsable and usable by

the rest of the application.

\* Chains: Although simple in this example (prompt | structured\_llm), LangChain's concept of

"chains" allows you to combine multiple components (like prompts, LLMs, and output parsers)

into a single, coherent workflow.

In essence, LangChain acts as an orchestration layer, making it easier to build applications

that leverage the power of LLMs for specific tasks like user story analysis, by providing

tools for prompt engineering, model interaction, and structured data extraction.