**Development Document: QA Agent System**

**1. System Overview**

The QA Agent is a Streamlit web application that uses a Large Language Model (LLM) to automatically generate comprehensive QA documentation. It takes a user's request and produces a user story, functional/non-functional requirements, test cases, and a requirements traceability matrix. The generated data can be viewed in a user-friendly table format, downloaded as a JSON or Excel file, and saved to a local database for historical tracking and management.

The application leverages LangChain for seamless LLM integration and Pydantic for robust data validation and structured output.

**2. System Architecture**

The system follows a layered architecture with distinct components for the UI, business logic, LLM interaction, and data persistence.

* **Frontend (Streamlit):**
  + main.py is the core of the application. It handles the user interface, manages user input, and displays the generated output.
  + It uses Streamlit components (st.button, st.text\_area, st.dataframe) to create an interactive web app.
* **Backend/Business Logic:**
  + main.py also contains the business logic for handling user requests and orchestrating the data flow.
  + It selects the appropriate LLM (Gemini or Groq) and the correct LangChain prompt and output schema based on user selections.
* **LLM Integration (LangChain):**
  + LangChain is used to create a pipeline (or **chain**) that connects the prompt templates to the LLM.
  + **Structured Output:** The with\_structured\_output method is crucial. It forces the LLM to generate output that strictly adheres to the Pydantic schemas defined in schemas.py. This ensures the data is well-formed and can be reliably parsed.
* **Data Models (Pydantic):**
  + schemas.py defines the structure of all generated documents (UserStory, Requirement, TestCases, RequirementMatrix).
  + It also includes **wrapper classes** (RequirementsList, RequirementMatrixList) to handle lists of objects, which is a requirement for LangChain's structured output feature.
* **Data Persistence (SQLAlchemy):**
  + db.py manages all database interactions. It uses **SQLAlchemy** to connect to a local SQLite database (qa\_agent.db).
  + The PromptRun class acts as an **ORM (Object-Relational Mapping)** model, mapping Python objects to rows in the prompt\_runs table.
  + It provides functions to save, get, and delete records.

**3. Business Data Flow**

The data flow can be broken down into two main processes: document generation and history management.

**3.1 Document Generation Flow**

This process describes the journey of a user request from input to final output.

1. **User Input:** The user enters a request into the text area on the Streamlit UI and clicks "Generate."
2. **Request Processing:** The main.py application receives the request and, based on the user's selection, chooses the appropriate LLM and a specific LangChain chain (e.g., user\_story\_chain, req\_matrix\_chain).
3. **LLM Interaction:** The chosen chain invokes the LLM (Gemini or Groq) with the user's prompt. The with\_structured\_output method ensures the LLM's response is formatted according to the Pydantic schema (e.g., UserStory, RequirementsList, etc.).
4. **Data Parsing:** LangChain's with\_structured\_output automatically parses the LLM's JSON response into a Pydantic object.
5. **Data Storage:** The Pydantic object is converted into a Python dictionary (.model\_dump()) and then a JSON string. The original prompt and the JSON output string are passed to the save\_prompt\_run function in db.py to be saved as a new record in the prompt\_runs table.
6. **UI Display:** The data (as a Python dictionary) is stored in Streamlit's session state. The UI then retrieves this data to render it in either a JSON block or a series of pandas DataFrames displayed as tables.
7. **Data Download:** The user can click download buttons, which trigger helper functions (to\_excel or a simple JSON download) to create a downloadable file from the generated data.

**3.2 History Management Flow**

This process handles retrieving and manipulating past records.

1. **History Request:** The user navigates to the "Download & Manage History" tab.
2. **Data Retrieval:** The get\_all\_runs function in db.py is called to fetch all records from the prompt\_runs table.
3. **UI Display:** Each record is displayed with a checkbox. The user can select one or more records.
4. **Action (Download/Delete):**
   * **Download:** If the user clicks "Download Selected" or "Download All," the to\_excel\_history function is called. This function processes the JSON data from the selected records, converts it into a multi-sheet Excel file, and provides it as a downloadable file.
   * **Delete:** If the user clicks "Delete Selected," the IDs of the selected records are passed to the delete\_prompt\_runs function in db.py. This function executes a database query to remove the corresponding records from the table. The app then reruns to show the updated history.

**3.3 Data Flow Diagram**

This diagram provides a high-level visual representation of the business data flow.

The flow is cyclical: user input triggers generation, which is saved to the database and displayed. The user can then interact with the database history through the UI to retrieve and manage past data.