**1. Orchestrate Multi-Agent Interactions**

**Story:**

As a user, I want my query to be intelligently routed to the appropriate agents (Data Insights or Enterprise Search) so that I can get accurate answers from both structured and unstructured data sources.

**Acceptance Criteria:**

* The **Orchestrator** agent routes user queries to the **Data Insights (DI)** or **Enterprise Search (ES)** agent based on the type of query.
* The **Orchestrator** aggregates responses when both agents return data.
* Stateless routing is implemented; the Orchestrator does not require internal session storage.

**What Needs to Happen in the Code:**

1. Implement query-type detection for routing to **DI** or **ES**.
2. Enable **DI** and **ES** agents to run queries simultaneously.
3. Aggregate responses into a unified format when both agents return results.
4. Implement pass-through logic when only one agent provides a response.
5. Send the final response to the **Chat UI**.

**2. Structured Data Handling with Data Insights (DI)**

**Story:**

As a user, I want the system to handle queries on structured data (e.g., CSV files) and return SQL-query-based insights accurately.

**Acceptance Criteria:**

* The **Data Insights (DI)** agent executes SQL queries against structured data files.
* SQL results are processed and formatted for clarity.
* **DI** agent connects to the shared SQL database to check session history.

**What Needs to Happen in the Code:**

1. Generate SQL queries dynamically based on user prompts.
2. Retrieve relevant chat history from the SQL database before generating a query.
3. Ensure SQL results are converted into readable text.
4. Return results to the **Orchestrator** agent.

**3. Unstructured Data Handling with Enterprise Search (ES)**

**Story:**

As a user, I want the system to search through unstructured data and return insights, including information extracted from images.

**Acceptance Criteria:**

* The **Enterprise Search (ES)** agent processes user queries on unstructured data.
* **ES** handles image text extraction.
* **ES** connects to the shared SQL database for session history retrieval.
* Embeddings are fetched from the **Vector Database** for semantic search.

**What Needs to Happen in the Code:**

1. Implement logic for unstructured data searches.
2. Support OCR or image text extraction mechanisms.
3. Use embeddings for similarity-based querying.
4. Validate context from shared chat history before processing.
5. Return results to the **Orchestrator** agent.

**4. Unified Chat History Management**

**Story:**

As a user, I want my chat history to be consistently stored and retrieved so that follow-up queries can maintain accurate context.

**Acceptance Criteria:**

* A **Chat History Manager** maintains a unified SQL database for all agent responses and user prompts.
* Chat history includes **session\_id**, **chat\_id**, **user\_query**, **agent\_response**, **source\_agent**, and **aggregated\_response**.
* Both **DI** and **ES** agents can access the same chat history.

**What Needs to Happen in the Code:**

1. Design SQL tables to include **session\_id**, **chat\_id**, **user\_query**, **response**, **source\_agent**, and **timestamp**.
2. Store intermediate and final responses into the SQL database.
3. Enable efficient queries for retrieving the latest **n** interactions.
4. Allow both **DI** and **ES** to pull from the same session history.

**5. Follow-Up Query Handling Across Agents**

**Story:**

As a user, I want follow-up queries to route correctly to the **Data Insights** or **Enterprise Search** agent based on the context of the original query.

**Acceptance Criteria:**

* Follow-up queries are routed to the most relevant agent (**DI** or **ES**).
* The **Orchestrator** determines follow-up routing based on the original context.
* Both **DI** and **ES** agents validate context against chat history before responding.

**What Needs to Happen in the Code:**

1. Implement context-aware routing in the **Orchestrator** agent.
2. Ensure each agent cross-references chat history before responding.
3. Gracefully handle mismatched context (e.g., **DI** receiving **ES**-related follow-ups).
4. Clearly annotate the source of the follow-up response.

**6. Report Generation Agent**

**Story:**

As a user, I want aggregated insights from both **DI** and **ES** agents to be summarized into a meaningful report.

**Acceptance Criteria:**

* The **Report Generation Agent** combines responses from **DI** and **ES** agents.
* The final aggregated response is stored in the shared chat history.

**What Needs to Happen in the Code:**

1. Implement response-merging logic.
2. Standardize report formatting (e.g., bullet points, tables).
3. Store the aggregated response with metadata.

**7. Stateless Orchestrator Workflow**

**Story:**

As a user, I want the **Orchestrator** agent to route and aggregate data without maintaining internal state.

**Acceptance Criteria:**

* **Orchestrator** relies on shared SQL history for context.
* Stateless routing ensures scalability and modularity.
* Follow-up queries are re-evaluated each time without local dependencies.

**What Needs to Happen in the Code:**

1. Avoid session persistence in **Orchestrator**.
2. Retrieve session data from SQL history on each interaction.
3. Ensure **Orchestrator** dynamically routes every query.

**8. Reset Context Command**

**Story:**

As a user, I want to clear my session's chat history and context to start fresh.

**Acceptance Criteria:**

* A **Reset Context Command** clears the session context via the **Shared API**.
* Chat history is archived or deleted for the session.

**What Needs to Happen in the Code:**

1. Implement an API endpoint for session resets.
2. Remove session-specific chat history.
3. Add a "Reset Context" button in the frontend.

**9. Monitoring and Logging**

**Story:**

As a developer, I want to monitor all agent activities, including data retrieval, routing, and follow-up query handling.

**Acceptance Criteria:**

* Logs include **session\_id**, **query\_type**, **agent**, and **response\_time**.
* Errors are logged with detailed metadata.
* Monitoring metrics include query latency and system load.

**What Needs to Happen in the Code:**

1. Store logs in a structured format.
2. Implement error-handling mechanisms.
3. Use tools like **Azure Monitor** or **Prometheus** to track system health.