## Architecture Overview: A Two-Phase Approach

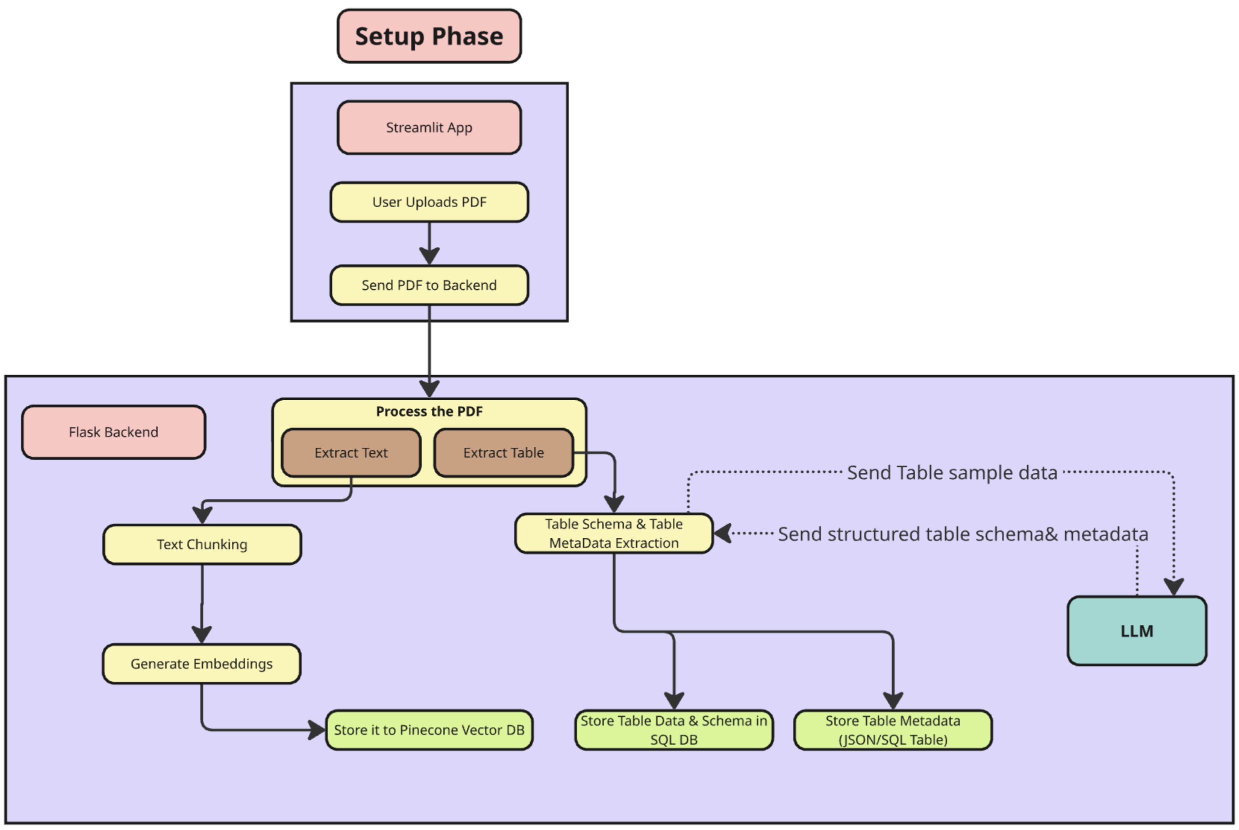
To build a robust and scalable multi-agent RAG system, we will divide the architecture into two distinct phases:

1. **The Setup Phase (Ingestion & Processing):** This phase handles the initial ingestion, processing, and storage of the uploaded PDF documents. Its primary goal is to prepare the data in an optimal format for efficient retrieval later.
2. **The Runtime Phase (Query & Response):** This phase is activated when a user submits a query. It involves a sophisticated multi-agent system that orchestrates the retrieval of relevant information from the processed documents and generates a comprehensive answer.

This separation of concerns ensures that the computationally intensive pre-processing happens only once per document, allowing for a swift and efficient user experience during the query-response loop.

# 1. Setup Phase Architecture

This phase is the foundation of our system. It is responsible for taking a raw PDF, understanding its contents (both text and tables), and storing them in a structured way that our AI agents can later query.


Setup Phase
Streamlit App
User Uploads PDF
Send PDF to Backend
Flask Backend
Process the PDF
Extract Text
Extract Table
Send Table sample data
Text Chunking
Table Schema & Table MetaData Extraction
Send structured table schema& metadata
LLM
Generate Embeddings
Store it to Pinecone Vector DB
Store Table Data & Schema in SQL DB
Store Table Metadata (JSON/SQL Table)

| Component | Description | Technologies |
|---|---|---|
| Streamlit Frontend | A user-friendly web interface where users can upload PDF files and interact with the system. | Streamlit |
| Flask Backend | The central nervous system of the setup phase. It provides a RESTful API endpoint for file uploads, communicates with all other backend components, and manages the overall processing pipeline. | Flask, Python |
| Temporary File Storage | A transient storage location for the uploaded PDF file while it is being processed. This can be a local directory on the server or a cloud-based object store. | Local File System, Amazon S3, Google Cloud Storage |
| Central ServiceRoom Agent | While more active in the runtime phase, a lightweight version of this agent can be used here to orchestrate the setup process, triggering the PDF Extractor Agent and monitoring its progress. | Custom Python Class/Module |
| PDF Extractor Agent | This specialized agent is responsible for parsing the raw PDF file. It identifies and separates the textual content from the tabular data. | PyMuPDF, pdfplumber |
| Text Chunker & Vector DB Manager | This component takes the extracted text, splits it into smaller, semantically meaningful chunks, generates vector embeddings for each chunk using a pre-trained language model, and then stores these embeddings in a vector database. | LangChain, Sentence-Transformers, Pinecone |

| | | |
|---|---|---|
| Table Extractor & Schema Inference Agent | This agent extracts tables from the PDF. It then infers the schema of each table (column names and data types) and converts the table into a structured format like a CSV or a database table. | Camelot, Tabula-py, Pandas |
| Table Enhancement Agent (Optional) | For tables lacking context (e.g., a table of financial figures without a year), this agent can infer missing metadata by analyzing the surrounding text in the original PDF. | Custom NLP logic, LLM calls |
| Pinecone Vector DB | A managed vector database used to store the embeddings of the text chunks. It enables efficient similarity-based retrieval. | Pinecone |
| SQL Database (e.g., PostgreSQL, SQLite) | A relational database used to store the extracted and structured tabular data. This allows for powerful and precise data retrieval using SQL queries. | PostgreSQL, SQLite, MySQL |
| Redis Cache(optional) | An in-memory data store used for caching frequently accessed data, such as document processing status or extracted metadata, to speed up repeated operations. | Redis |

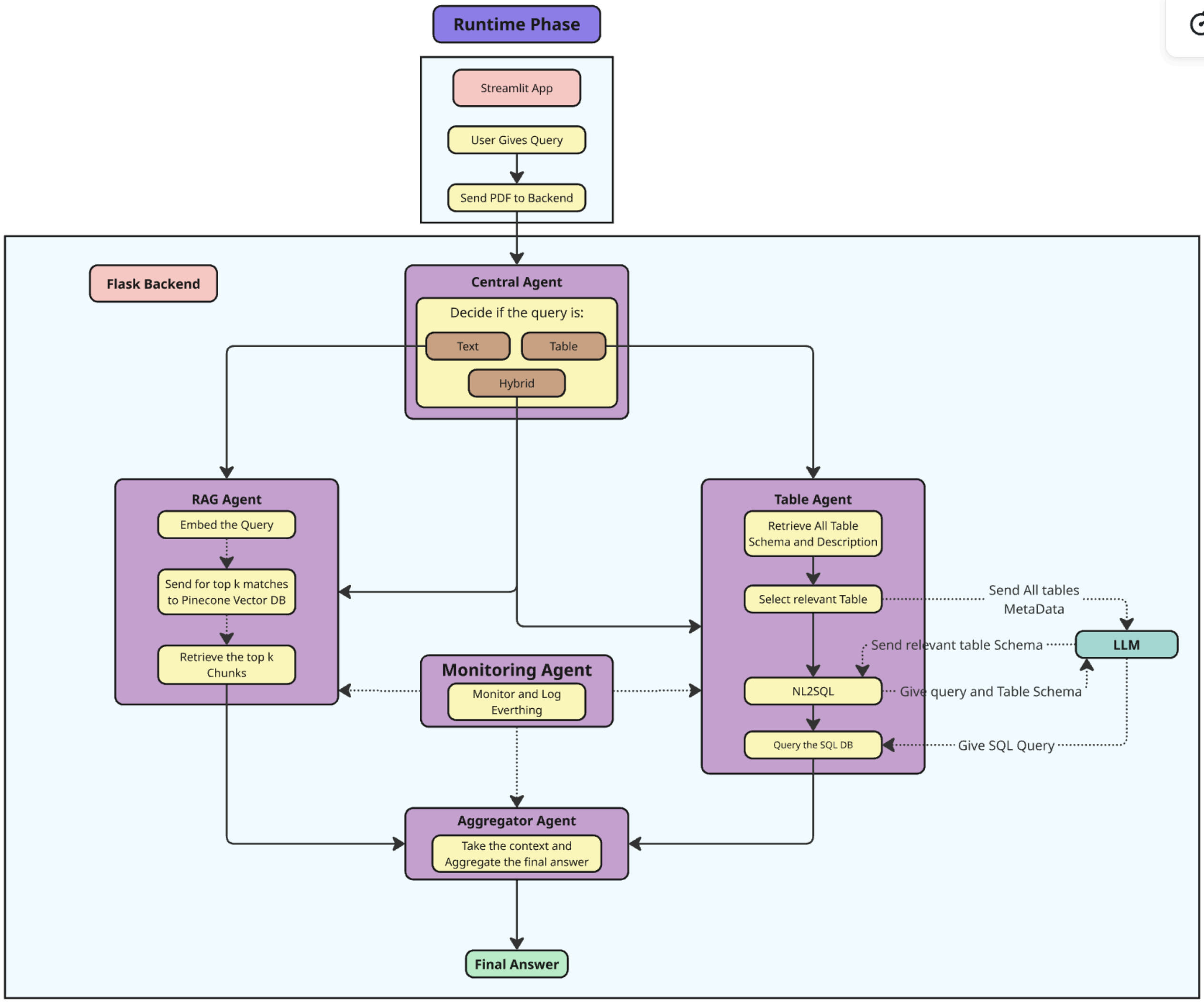
## Setup Phase: Workflow

1. **PDF Upload:** The user uploads a PDF file through the Streamlit frontend.
2. **API Trigger:** The frontend sends the file to the Flask backend via a POST request to a dedicated `/upload` endpoint.
3. **Temporary Storage:** The Flask application saves the uploaded PDF to a temporary storage location.
4. **Orchestration Kick-off:** The backend's Central ServiceRoom Agent is notified of the new file and initiates the processing pipeline.
5. **PDF Decomposition:** The Central ServiceRoom Agent invokes the **PDF Extractor Agent**. This agent scans the PDF and intelligently separates the unstructured text from the structured tables.
6. **Text Processing Pathway:**
   - The extracted text is passed to the **Text Chunker & Vector DB**.
   - The text is segmented into smaller chunks.
   - Vector embeddings are generated for each chunk.
   - The chunks and their corresponding embeddings are stored in the **Pinecone Vector DB**.
7. **Table Processing Pathway:**
   - The identified tables are passed to the **Table Extractor & Schema Inference Agent**.
   - This agent extracts the tables, cleans them, infers their schemas, and converts them into a structured format (e.g., a Pandas DataFrame).

    - *(Optional)* The **Table Enhancement Agent** can be invoked to add missing contextual information (like dates) to the table's metadata by analyzing the text surrounding the table in the original document.
    - The processed tables are then stored in a dedicated **SQL Database**. Each table from the PDF becomes a table in this database.
8. **Status Update & Caching:** Redis is widely used as a fast, in-memory data store for caching and quick status tracking. It’s ideal for storing lightweight, rapidly-changing state like job or document statuses (processing, completed, failed).It allows your frontend to instantly retrieve the status without hitting a slower backend/database.
9. **Completion:** Once both pathways are complete, the system marks the document as "ready for querying."

## 2. Runtime Phase Architecture

This is where the magic happens. When a user asks a question, this multi-agent system collaborates to understand the query, retrieve the most relevant information (from text, tables, or both), and synthesize a precise and accurate answer.



| Component | Description | Technologies |
|---|---|---|
| Streamlit Frontend | The user interface for submitting queries and displaying the generated answers. | Streamlit |
| Flask Backend | Provides the /query API endpoint and hosts the multi-agent system. | Flask, Python |
| Central ServiceRoom Agent (Orchestrator) | The mastermind of the runtime phase. It receives the user query, delegates tasks to specialized agents, and orchestrates the overall workflow to generate the final answer. | LangChain Agents, LlamaIndex Query Engines |
| Text Retriever Agent | This agent specializes in finding relevant text passages. It takes the user query, converts it into a vector embedding, and queries the | Pinecone, Sentence-Transformers |

|  |  |  |
|---|---|---|
|  | Pinecone Vector DB to find the most similar text chunks. |  |
| Table Retriever Agent | This agent identifies which tables in the SQL database are relevant to the user's query. It can do this by comparing the query to the table associated metadata. | Custom logic, LLM-based routing |
| SQL Executor (NL-to-SQL Agent) | A powerful agent that translates a natural language question into a precise SQL query. It then executes this query on the SQL database to fetch specific data points from the relevant tables. | LangChain SQL Agent, LlamaIndex NL-to-SQL |
| Aggregator Agent | The final agent in the chain. It receives the retrieved text snippets and the data from the SQL queries. Its job is to synthesize all this information into a coherent, human-readable answer, citing the sources of the information. | Large Language Model (e.g., GPT-4, Claude 3, Gemini) |
| Monitoring & Logging Agent | A crucial but often overlooked agent. It runs in the background, logging all agent interactions, query performance, and any errors. This is vital for debugging, evaluating, and improving the system over time. | ELK Stack (Elasticsearch, Logstash, Kibana), Prometheus, Grafana |

## Runtime Phase: Workflow

1. **Query Submission:** The user types a question into the Streamlit interface and hits "submit."
2. **API Call:** The frontend sends the query to the Flask backend's `/query` endpoint.
3. **Orchestration Begins:** The **Central ServiceRoom Agent** receives the query and initiates the runtime workflow.
4. **Hybrid Retrieval Strategy:** The Central ServiceRoom Agent analyzes the query to decide the best retrieval strategy.
    - **Conditional Triggering:** The agent first determines if the query is more likely to be answered by text, tables, or a combination. This can be done using a smaller, faster language model to classify the query's intent.
        - For a query like "What were the company's net profits in 2023?", the agent would prioritize the Table RAG path.
        - For a query like "What is the company's mission statement?", the Text RAG path would be prioritized.

        - For a complex query like "Summarize the key financial highlights and the CEO's comments on them," both pathways would be triggered in parallel.
    - **Joint Triggering (Default for High Accuracy):** For maximum accuracy, the system can be configured to always trigger both the Text and Table Retriever Agents simultaneously.
5. **Text Retrieval Path:**
    - The Central ServiceRoom Agent passes the query to the **Text Retriever Agent**.
    - This agent vectorizes the query and retrieves the top-k most relevant text chunks from the **Pinecone Vector DB**.
6. **Table Retrieval Path:**
    - The Central ServiceRoom Agent passes the query to the **Table Retriever Agent**.
    - This agent identifies the most relevant table(s) in the **SQL Database**.
    - The query and the schema of the selected table(s) are then passed to the **SQL Executor (NL-to-SQL Agent)**.
    - The NL-to-SQL Agent converts the natural language query into a SQL query (e.g., `SELECT "Net Profit" FROM financial_summary WHERE Year = 2023;`).
    - The SQL query is executed on the database, and the resulting rows of data are returned.
7. **Information Aggregation:**
    - The retrieved text chunks and the data from the SQL query are passed to the **Aggregator Agent**.
8. **Answer Generation and Synthesis:**
    - The **Aggregator Agent** (powered by a powerful LLM like Gemini or GPT-4 or Claude 3) synthesizes the information from both sources.
    - **Maximizing Accuracy and Avoiding Redundancy:**
        - The agent is instructed to prioritize the structured data from the tables for factual, numerical answers.
        - The textual context is used to supplement and explain the tabular data.
        - The agent performs a final cross-reference to ensure consistency and removes any redundant information before generating the final answer.
9. **Response Delivery:** The final, synthesized answer is sent back to the Flask backend, which then relays it to the Streamlit frontend to be displayed to the user.
10. **Continuous Monitoring:** Throughout this entire process, the **Monitoring & Logging Agent** is capturing data on agent performance, retrieval times, and the quality of the final output for ongoing system improvement.

## Key Design Decision: Should table data be included in text chunks?

**No, table data should NOT be duplicated in the text chunks.** Here's why this separation is critical for a high-performance system:

- **Avoiding Redundancy:** Storing the same information in two places is inefficient and increases the risk of inconsistencies.
- **Leveraging a "Right Tool for the Job" Approach:**
  - **Vector search (Pinecone)** is excellent for semantic, similarity-based searches on unstructured text. It's great for finding concepts, summaries, and descriptive passages.
  - **SQL databases** are purpose-built for precise, structured data retrieval. They can perform exact matches, filtering, sorting, and aggregations (like `SUM`, `AVG`, `COUNT`) with perfect accuracy, something vector search struggles with.
- **Clarity for the Agents:** By keeping text and tables separate, we provide a clean and unambiguous data landscape for our agents. The `NL-to-SQL Agent` knows it only needs to deal with the structured database, and the `Text Retriever Agent` knows it's searching over prose. This specialization improves the reliability of each agent.
- **Scalability and Maintenance:** A separate SQL database for tables is easier to manage, update, and scale independently of the vector database.

By architecting the system in this manner, we create a production-ready, research-level RAG system that is not only powerful and accurate but also scalable, maintainable, and easy to understand.

TEAM ROLES & TASK BREAKDOWN (Updated)

Overall Leads

| Name | Role | Responsibility |
|---|---|---|
| **Vijendir Sir** | Project Sponsor / Director | Final approvals, stakeholder alignment, architectural guidance |
| **Krishna Kaushik** | AI Systems Architect / Technical Lead | End-to-end architecture, technical design decisions, agent orchestration lead |

Core Team Roles

| Role | Responsibilities |
|---|---|
| **Frontend Engineer** | Streamlit UI: PDF upload, query bar, answer rendering, traceable answer breakdowns |
| **Backend Engineer** | Flask endpoints, multi-agent orchestration, PDF parsing, async handlers |
| **Prompt Engineer** | Write, tune, and test all prompt templates for schema extraction, SQL generation, summarization |
| **Data/Infra Engineer** | Set up Pinecone, DuckDB, in-memory tables, schema store, and all caching layers |
| **LLM Integration Engineer** | Call Gemini/GPT via API for prompts, handle retries, latency control, API failover |
| **DevOps Engineer** | Dockerization, Gunicorn, Nginx, CI/CD setup, cloud deploy to AWS/GCP |
| **QA Engineer** | Unit & integration testing: agents, prompt outputs, SQL exec results, latency thresholds |

## Development Pipeline

1. Development

  - Branching per module: feature/pdf-extract, feature/sql-prompt, etc.

- Linters (black, flake8), type-checkers (mypy), formatters

2. Version Control & Code Review

 - PRs via GitHub → reviewed by Krishna/Vijendir sir

3. CI/CD Pipeline

 - Gunicorn + Nginx

 - Optional: - GitHub Actions: build, test, Dockerize → deploy to staging

4. Logging & Monitoring

 - Logs: `logging`, `Sentry`

 - Metrics: `Prometheus` hooks per agent

 - Dashboards: `Grafana` (latency, LLM call time, retrieval errors)

5. Deployment Guidelines

 - `Dockerfile` + `docker-compose.yaml` for local & staging

 - Use `.env` for all secrets, API keys, and config toggles

## FULL CODE OUTLINE (Updated)

### 📁 Modular Project Structure

```
pdf_rag_backend/
├── app.py                  # ✅ All Flask routes in one file
├── agents/                 # Core functional logic lives here
│   ├── pdf_extractor.py
│   ├── text_chunker.py
│   ├── embedder.py
│   ├── retriever.py
│   ├── table_extractor.py
│   ├── schema_agent.py
│   ├── sql_prompt_agent.py     # LLM-prompted SQL generation
│   ├── sql_executor.py
│   ├── aggregator.py
│   └── answer_formatter.py
├── prompts/
│   └── prompt_templates.py     # All system + user prompts here
├── utils/
│   ├── logger.py
│   └── observability.py       # Prometheus decorators
├──.env
├── requirements.txt
├── Dockerfile
└── docker-compose.yaml
```

## 🧰 Suggested Libraries

## USE LangGraph framework for building intelligent workflows using graph-based logic, especially for LLMs and tool-based agents.

| Purpose | Library |
|---|---|
| Backend framework | Flask, Flask-CORS |

| | |
|---|---|
| PDF parsing | PyMuPDF, pdfplumber, camelot |
| Chunking & retrieval | langchain, pinecone-client |
| LLMs | Gemini Pro, GPT-4 |
| SQL | MYSQL, DuckDB, pandasql |
| Observability | Sentry, Prometheus, Grafana |
| DevOps | Docker, Gunicorn, GitHub Actions |

## Suggested Evaluation Metrics

| Metric | Description |
|---|---|
| **Text Answer Accuracy** | Human-rated relevance or BLEU-style similarity to ground truth |
| **Table Query Accuracy** | SQL match correctness against reference answer |
| **Latency** | Time from PDF upload → final response |

| | |
|---|---|
| **Join Reasoning Score** | Correct fusion of table + text contexts |
| **Failure Mode Audit** | % of questions where hallucination / blank / crash occurs |