Agentic RAG Chatbot

- Agentic RAG Chatbot: System Architecture*A multi-agent system for document-based Question Answering with a custom communication protocol
- A Retrieval-Augmented Generation (RAG) chatbot that answers questions based on uploaded documents (PDF, PPTX, DOCX, CSV, TXT).
- **Core Design :** Built on a multi-agent architecture where specialized agents collaborate to process, retrieve, and generate answers.

Project Overview & Key Features

Core Design: Built on a multi-agent architecture where specialized agents collaborate to process, retrieve, and generate answers.

Key Features:

- 📄 Multi-format document parsing.
- 🤖 Specialized agents for Ingestion, Retrieval, and Response Generation.
- 💬 Custom communication protocol (MCP) for structured agent interaction.
- 🔍 High-speed semantic search using FAISS and Sentence Transformers.
- 🗣️ Support for multi-turn conversations and follow-up questions.
- 📱 Clean, interactive UI built with Streamlit.

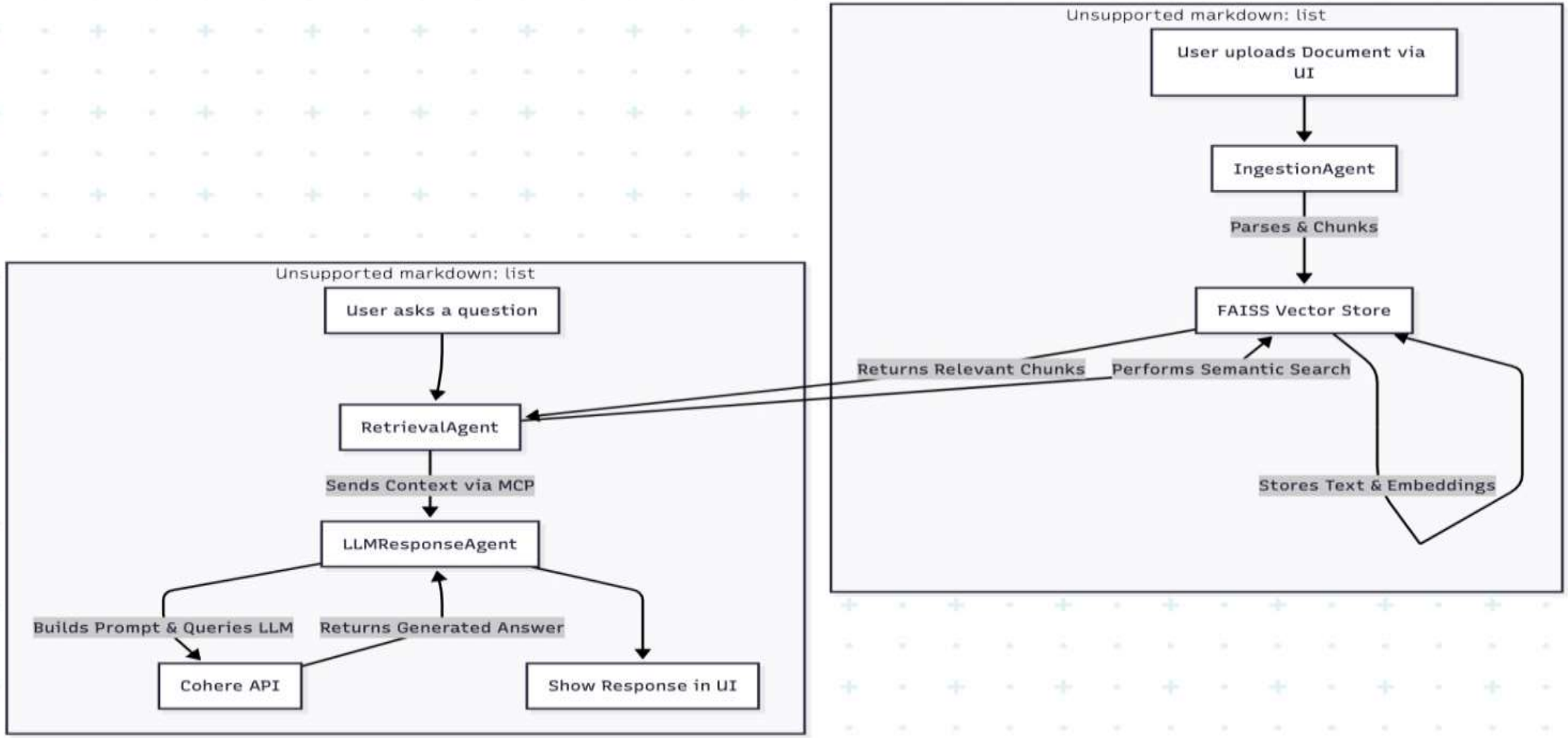
System Architecture: Code & Components

- The system is organized into modular components, each with a specific responsibility.
- agents/: The core logic units of the system.
- ingestion_agent.py: Parses and chunks documents.
- retrieval_agent.py: Creates embeddings and handles search.
- llm_response_agent.py: Generates the final answer.
- parsers/: Contains specialized functions to read different file types.
- vector_store/: Manages the FAISS index for semantic search.
- embeddings/: Handles the conversion of text to vectors.
- mcp/: Implements the Model Context Protocol for agent communication.
- ui/streamlit_app.py: The user-facing application.
- main.py: The entry point that orchestrates the agents.

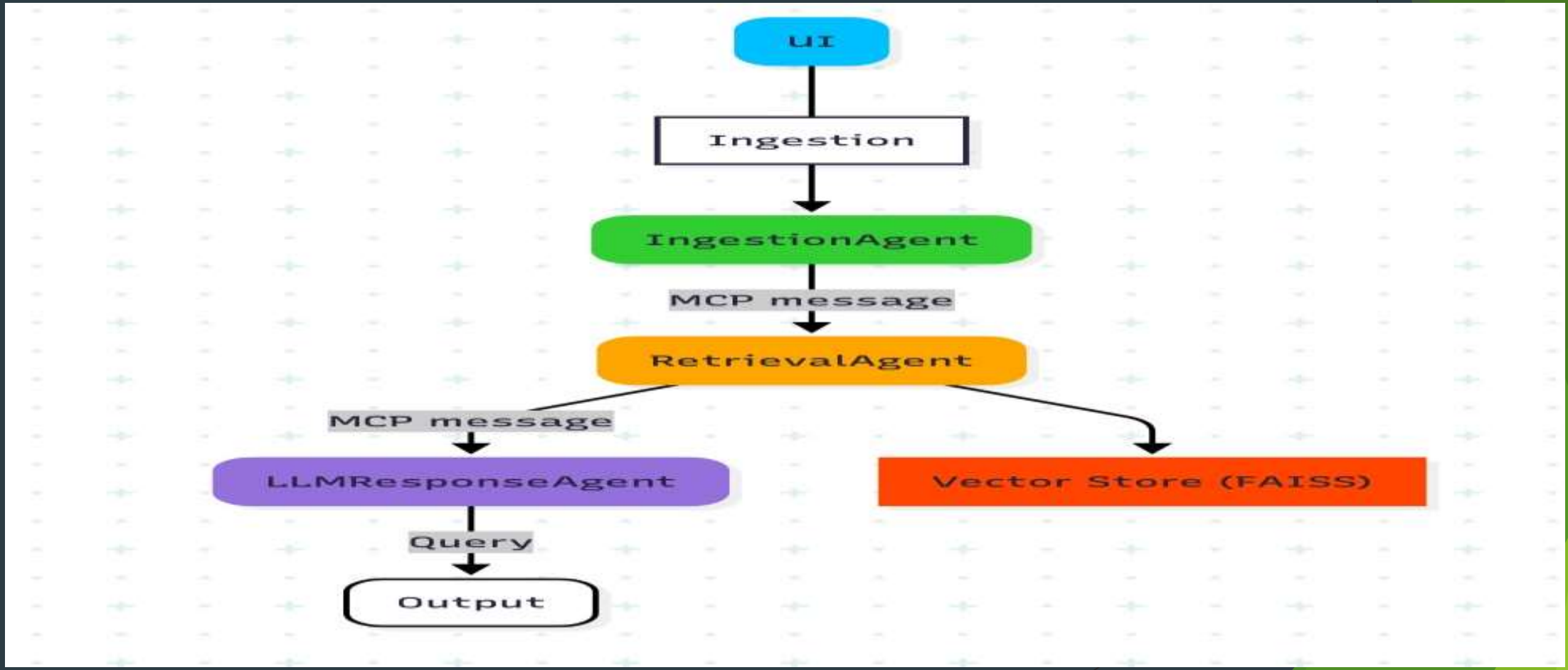
System Flow & Architectural Diagram

- The system follows a clear, orchestrated flow from document upload to answer generation.
- 1. A user uploads a document, which the Ingestion Agent processes and stores in the FAISS Vector Store.
- 2. When a question is asked, the Retrieval Agent finds relevant context from the store.
- 3. The LLM ResponseAgent uses this context to get a precise answer from the Cohere LLM.
- 4. The final response is displayed to the user.

System Flow


Unsupported markdown: list
User uploads Document via UI
IngestionAgent
Parses & Chunks
FAISS Vector Store
Stores Text & Embeddings
Performs Semantic Search
Returns Relevant Chunks
Unsupported markdown: list
User asks a question
RetrievalAgent
Sends Context via MCP
LLMResponseAgent
Builds Prompt & Queries LLM
Returns Generated Answer
Cohere API
Show Response in UI

Architectural Diagram



The Model Context Protocol (MCP)

MCP is a custom JSON-based message format that enables structured communication and traceability between agents.

Modularity: Decouples agents, allowing them to be developed and tested independently.

Traceability: Each message has a trace_id, making it easy to debug the flow of a request.

Clarity: Provides a clear, explicit structure for data exchange.

Example MCP Message:

```
json    {      "sender": "RetrievalAgent",
 "receiver": "LLMResponseAgent",
"type": "RETRIEVAL_RESULT",
 "trace_id": "rag-457",
 "payload": {        "retrieved_context": ["context chunk 1", ...],
  "query": "What KPIs were tracked in Q1?"      }    }
```

Technology Stack

The project is built on a modern, open-source Python stack.

Backend & Logic : Python 3.11+

User Interface: Streamlit

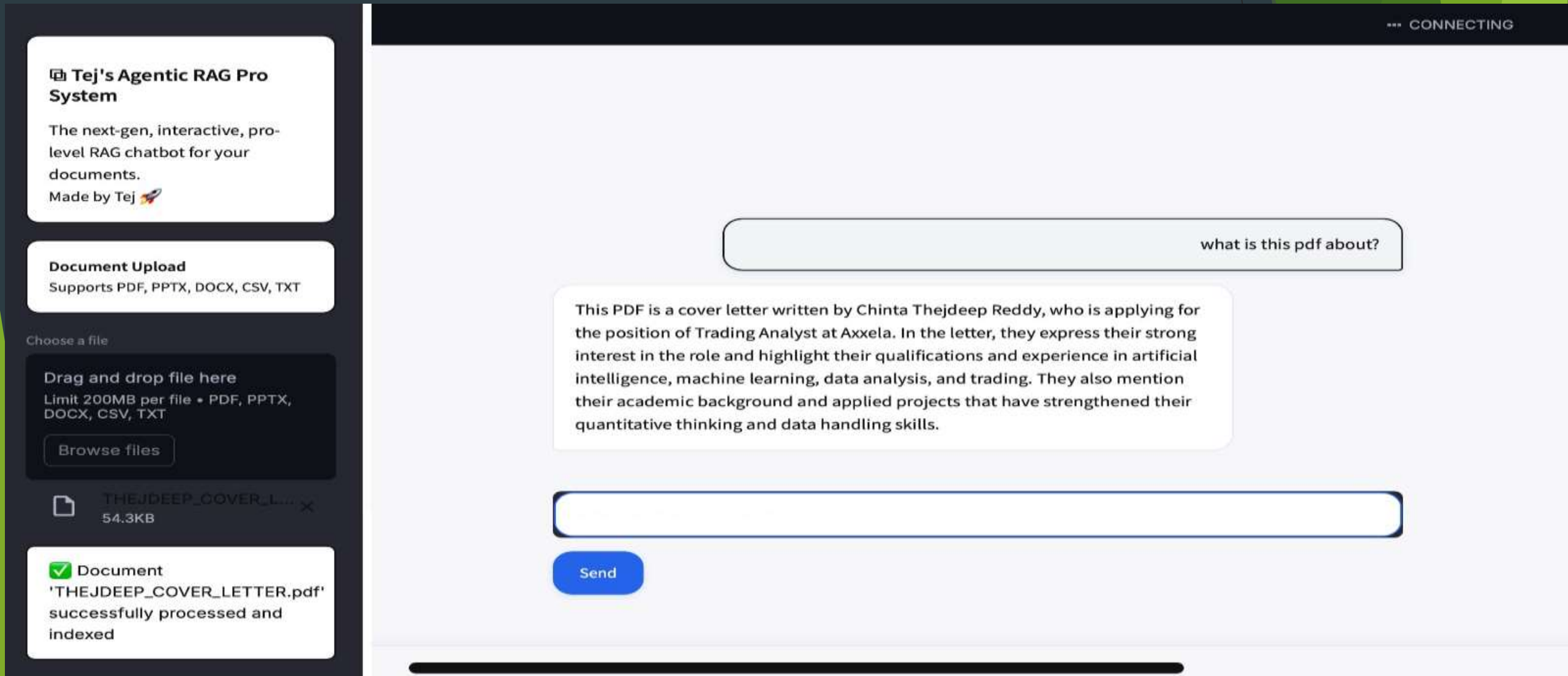
LLM: Cohere (command-r-plus)

Vector Search: FAISS (Facebook AI Similarity Search)

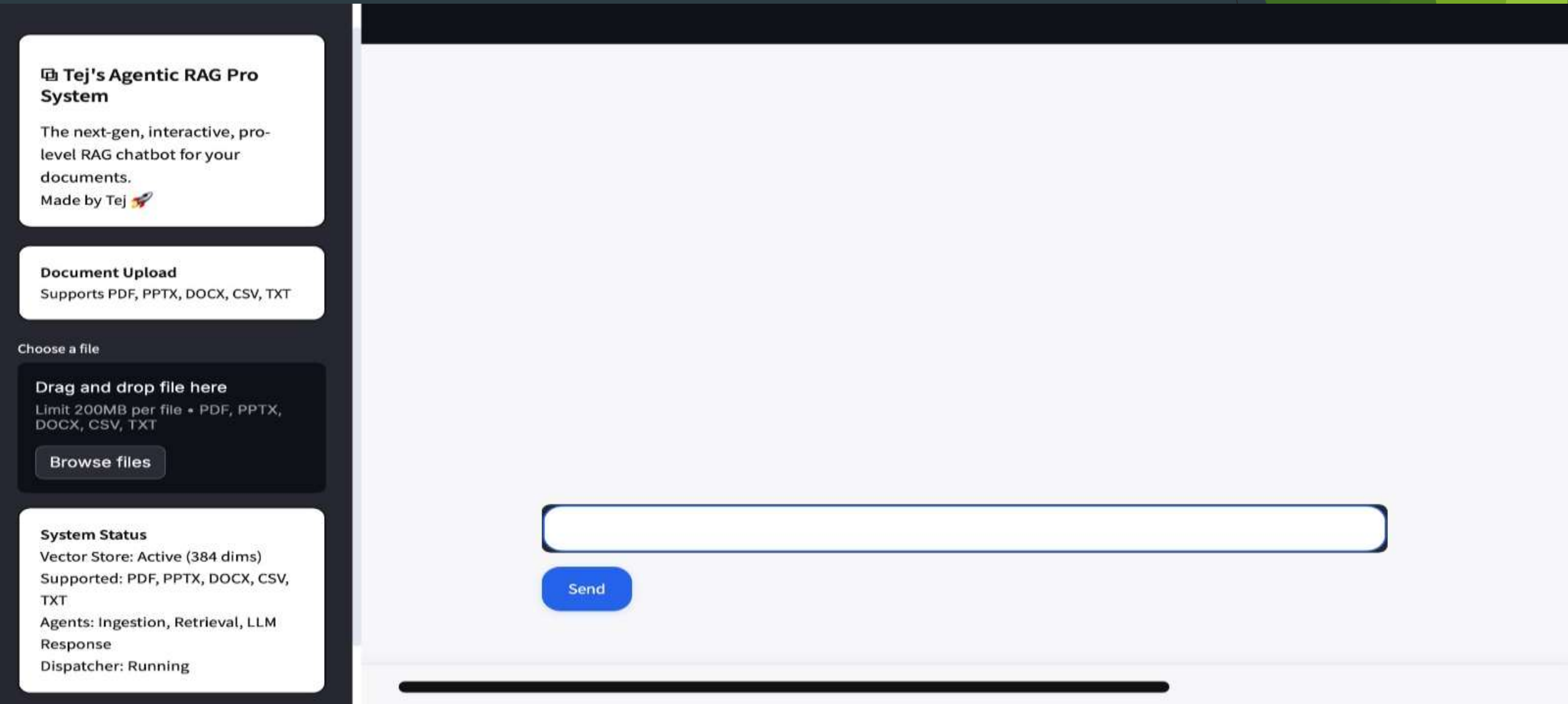
Embeddings: Sentence Transformers (all-MiniLM-L6-v2)

Document Parsing: PyMuPDF, python-pptx, python-docx, pandas

UI Screenshots of Working App



UI Screenshots of Working App



Challenges & Future Scope

Challenges Faced: Handling inconsistent parsing across document formats.

Ensuring a clear and robust MCP message flow.

Optimizing the embedding of large documents. Maintaining context in multi-turn conversations.

Future Roadmap: Deploy with FastAPI and Docker for production.

Integrate more LLMs (OpenAI, Mistral, Claude).

Use a message broker (Kafka/RabbitMQ) for asynchronous tasks.

Add user session management and chat history. Highlight source chunks in the UI for better explainability.

Conclusion

This agent-based architecture creates a robust, modular, and scalable RAG system. By separating concerns, it simplifies development and improves traceability.

Presented By:
Chintha Thejdeep Reddy
tejdeep2502@gmail.com