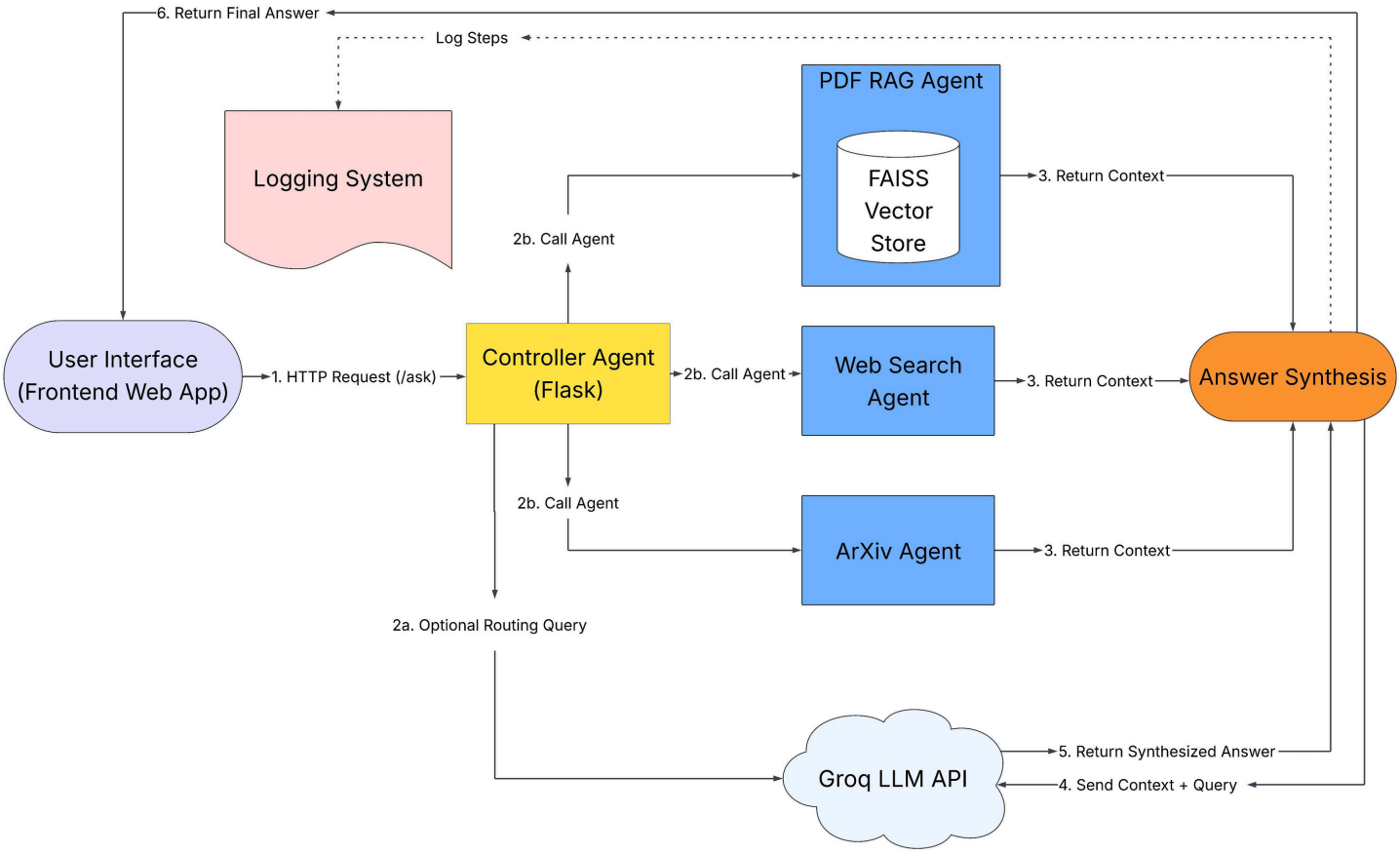
Multi-Agent AI System

1. Executive Summary

This report details the design, implementation, and deployment of a multi-agent AI system developed as part of an internship assignment. The system features a modular architecture centered around a **Controller Agent** capable of dynamically routing user queries to specialized agents: a **PDF Retrieval-Augmented Generation (RAG) Agent**, a **Web Search Agent**, and an **ArXiv Agent**. Powered by the Groq LLM API, the system offers real-time information retrieval, document analysis, and academic research capabilities. The entire solution is containerized with Docker and deployed on Hugging Face Spaces, with robust logging and traceability features.

2. System Architecture

The multi-agent system adopts a full-stack, client-server architecture. The core logic resides within a Flask backend, which orchestrates various AI agents to fulfill user requests.



2.1 Key Components and Their Roles

- **User Interface (Frontend Web App):** A simple web-based interface (HTML, CSS, JavaScript) allows users to input queries and upload PDF documents. It displays the final

answers and the controller's decision rationale.

- **Backend Server (Flask):** Written in Python, this Flask application hosts the API endpoints (/ask, /upload, /logs) and orchestrates the entire system. It manages the flow of information between the frontend, the Controller Agent, and the specialist agents.
- **Controller Agent (Flask):** This is the brain of the system. It analyzes incoming user queries using both rule-based logic and an external LLM (Groq) to determine the most appropriate specialist agent(s) to handle the request. It is responsible for logging its decision-making process.
- **Specialist Agents:**
  - **PDF RAG Agent:** Handles queries related to user-uploaded PDF documents. It includes an embedded **FAISS Vector Store** for efficient similarity search on document chunks.
  - **Web Search Agent:** Retrieves real-time information by performing web searches using the DuckDuckGo search API.
  - **ArXiv Agent:** Facilitates academic research by querying the ArXiv API and summarizing relevant papers.
- **Answer Synthesis:** This component (integrated within the backend logic) takes the context retrieved by the specialist agent(s) and the original user query, then sends it to the Groq LLM API to generate a concise and coherent final answer.
- **Groq LLM API:** An external Large Language Model service used for two primary functions:
  1. Assisting the Controller Agent in making intelligent routing decisions for ambiguous queries.
  2. Synthesizing the final human-readable answer based on the retrieved context.
- **Logging System (trace.log):** A robust logging mechanism that captures detailed traces of every significant step in the system, including controller decisions, agent interactions, and timestamps, stored in a trace.log file.

3. Agent Interfaces

Each specialist agent adheres to a clear, consistent interface, typically a single function that takes a query (and potentially a file ID) and returns formatted context.

- **PDF RAG Agent:**
  - **Interface:** query_pdf(query: str, file_id: str) -> str
  - **Description:** Processes a user query against a previously uploaded and indexed PDF. It performs vector similarity search using FAISS to retrieve the most relevant text chunks.
  - **Output Example:** A concatenated string of relevant paragraphs from the PDF.
- **Web Search Agent:**
  - **Interface:** search_web(query: str) -> str
  - **Description:** Executes a real-time web search for general knowledge or current events.
  - **Output Example:** A Markdown-formatted string containing titles, URLs, and snippets of top search results.
- **ArXiv Agent:**
  - **Interface:** search_arxiv(query: str) -> str
  - **Description:** Queries the ArXiv API for academic papers based on the user's input.
  - **Output Example:** A Markdown-formatted string listing relevant papers with their titles, authors, and abstracts.

---

4. Controller Logic

The Controller Agent is designed for efficient and intelligent query routing, employing a hybrid approach.

4.1 Rule-Based Logic

The first layer is a set of deterministic rules. These are fast and handle clear-cut cases.

- **PDF RAG Agent Trigger:** If a file_id is present (meaning a PDF has been uploaded and processed) AND the query contains keywords like "summarize," "this document," or similar phrases, the query is routed directly to the PDF RAG Agent.
- **ArXiv Agent Trigger:** If the query contains keywords such as "paper," "research," "academic," or "ArXiv," it is routed to the ArXiv Agent.
- **Web Search Agent Trigger:** If the query contains keywords like "latest news," "current events," or "what's happening," it is routed to the Web Search Agent.

4.2 LLM-Based Logic (Groq Prompt)

If a query does not trigger any of the above rules (i.e., it's ambiguous), the Controller leverages the Groq LLM API to make an intelligent routing decision.

System Prompt:

You are an expert routing agent. Your task is to analyze a user query and decide the best tool to use.

You have three tools available:

1. WebSearch: For real-time information, news, or general knowledge.

2. ArxivSearch: For scientific papers, research, or technical topics.

3. PDF-RAG: Use ONLY if the user explicitly refers to an uploaded document and a file_id is provided.

Respond in a simple JSON format like: {"tool": "YourChoice", "rationale": "Your reasoning."}

Your choice must be one of: "WebSearch", "ArxivSearch", or "PDF-RAG".

Rationale: This prompt instructs the LLM to act as a specialized router, providing a structured output (JSON) with a clear tool choice and its reasoning. This ensures consistency and enables the backend to programmatically use the LLM's decision.

5. Trade-offs and Design Decisions

During the project's development, several key technical trade-offs were considered:

- **Flask vs. FastAPI:** Flask was chosen for the backend due to its simplicity and rapid development capabilities for this assignment's scope. While FastAPI offers superior performance (asynchronous handling) and built-in validation, Flask allowed for quicker iteration on the core agentic logic.
- **FAISS vs. Persistent Vector Databases (e.g., ChromaDB):** FAISS (Facebook AI Similarity Search) was selected for the PDF RAG Agent because it provides an incredibly fast, in-memory vector indexing solution. For a demo where PDFs are processed per session, FAISS's simplicity and speed outweighed the need for a persistent, standalone vector database like ChromaDB, which would introduce more operational overhead for data management and persistence across sessions.
- **Groq API for LLM:** Groq was chosen specifically for its high inference speed. This was a critical factor for providing a responsive user experience, as both the controller's routing decisions and the final answer synthesis rely on LLM calls. The speed of Groq significantly reduced perceived latency compared to other LLMs.
- **Hybrid Routing Logic:** Combining rule-based and LLM-based routing provides an optimal balance. Rule-based checks offer deterministic, low-latency decisions for clear queries, while the LLM handles the more complex, ambiguous cases intelligently.

---

6. Deployment Notes

The application is deployed as a full-stack solution on **Hugging Face Spaces**.

- **Containerization (Docker):** The entire application (backend and frontend) is containerized using a Dockerfile. This ensures consistency across development and deployment environments. Key optimizations in the Dockerfile include:
  - Using a lightweight python:3.11-slim base image.
  - Leveraging build caching by copying requirements.txt and installing dependencies before copying the rest of the code.
  - Pre-downloading the sentence-transformers model during the build phase (RUN python -c "...") to significantly reduce application startup time.
- **Environment Variable Handling:** API keys, specifically the GROQ_API_KEY, are not hardcoded. For local development, they are managed via a .env file. In deployment on Hugging Face Spaces, the API key is securely provided as a **Space Secret**, which is automatically exposed as an environment variable to the running container.
- **Permissions Management:** Deployment to Hugging Face Spaces revealed common PermissionError issues, particularly with writing to log files and caching model data. These were resolved by:
  - Creating explicit writable directories (/app/logs and /app/cache) in the Dockerfile and setting appropriate permissions (chmod -R 777).
  - Configuring HF_HOME, TRANSFORMERS_CACHE, and SENTENCE_TRANSFORMERS_HOME environment variables to point to the /tmp directory (or /app/cache) to ensure model caches are written to writable locations.

- **run.sh Script:** A run.sh script manages the application startup. It ensures necessary cache directories are created and configures environment variables before launching the Flask application via Gunicorn.

7. NebulaByte PDF Generation & Usage

As part of the RAG agent demonstration, a dataset of five curated PDF documents was created and integrated.

- **Generation Method:** The PDFs were generated using an LLM (such as Gemini or ChatGPT) to create short, technical Q&A dialogs. These dialogs were framed as conversations between a "Senior Engineer" and a "Junior Developer" at a hypothetical tech entity named "NebulaByte," fulfilling the project's specific requirement.
- **Content:** The topics covered in these dialogs were directly relevant to the project's technical aspects, including:
  - The core principles of multi-agent AI systems.
  - How Retrieval-Augmented Generation (RAG) works.
  - The difference between FAISS and ChromaDB.
  - Best practices for API security in a Python backend.
  - Challenges of deploying containerized applications with Docker.
- **Usage:** These PDFs are placed in the sample_pdfs/ directory of the project. When a user uploads one of these (or any other) PDFs, the PDF RAG Agent processes it by extracting text, chunking it, creating embeddings, and storing them in the FAISS Vector Store. The agent then uses this indexed data to answer questions grounded directly in the document's content.