REPORT: Multi-Agentic System with Dynamic Decision Making

Submitted by:

Manit Bhattarai
Platform: Render (Backend + Frontend)
Repository: github.com/ManitB320/multi-agent-system

1. Overview and Objective:

The goal of this project is to build a multi-agent AI system that can dynamically decide which specialized agent(s) to use for a given user query.
The system integrates RAG (Retrieval-Augmented Generation) for PDF understanding, real-time Web and ArXiv search, and an LLM-based controller for intelligent routing and answer synthesis.

Core Goals

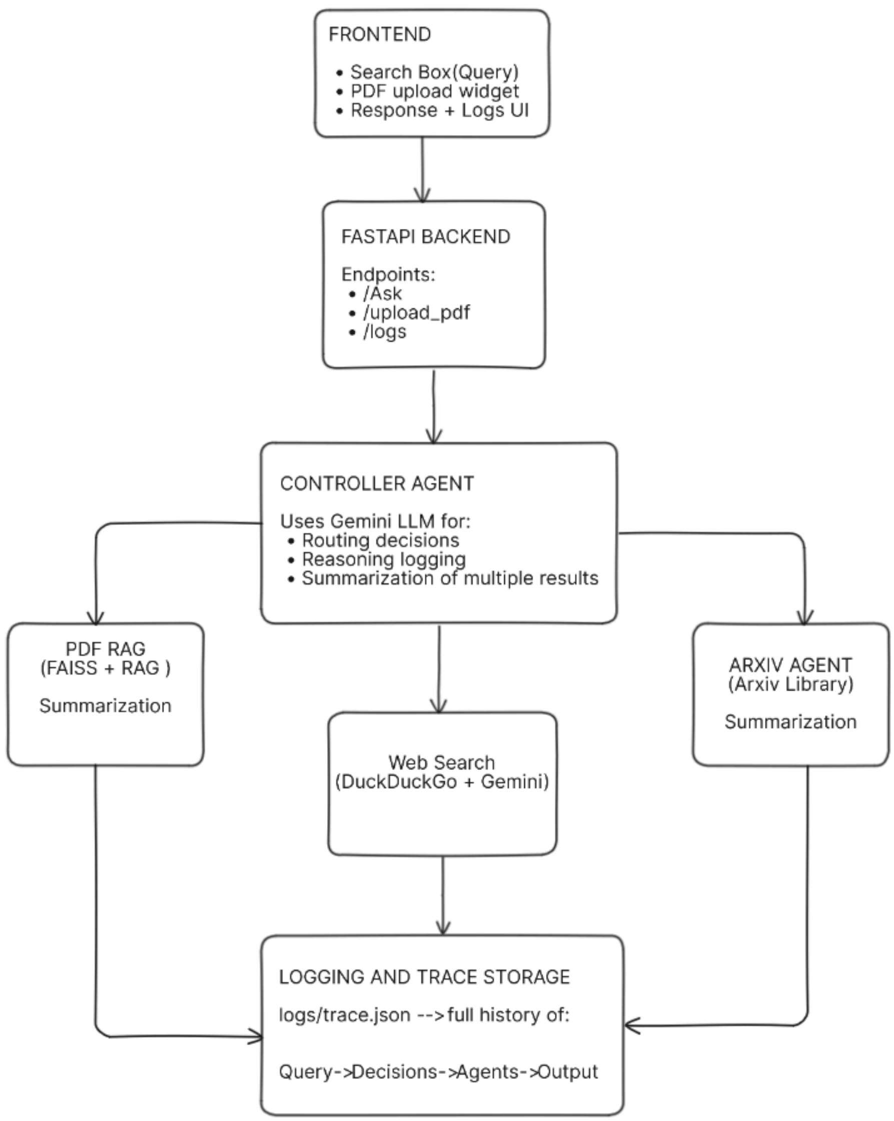
- Build a modular multi-agent framework using FastAPI.
- Use Gemini (Google AI Studio) for dynamic decision-making and summarization.
- Implement RAG-based PDF querying with FAISS embeddings.
- Provide a minimal, transparent frontend for user interaction.
- Ensure traceability and explainability through structured logging.
- Deploy the full stack using Render.

2. System Architecture:

Agent	Description	Key Technology
Controller Agent	Central brain that decides which agent(s) to invoke. Uses LLM (Gemini) + rule-based fallback. Logs full reasoning.	google.generativeai, FastAPI
PDF RAG Agent	Extracts, chunks, embeds, and retrieves text from user-uploaded PDFs.	PyMuPDF, SentenceTransformer, FAISS, numpy, pickle

Web Search Agent	Retrieves real-time news and factual data from verified web sources.	duckduckgo_search, Gemini for summarization
ArXiv Agent	Fetches latest academic papers and summarizes key points.	arxiv API
Frontend	HTML/CSS/JS interface for query and PDF upload. Displays result + reasoning + logs.	Fetch API, FastAPI CORS

3. Architecture Diagram: *(fig: block diagram of multi-agent-system)*



4. Controller Logic

Dual-layer Decision Making:

1. Primary Layer:
 The controller queries Gemini (LLM) with a prompt describing all available agents and their capabilities.
 Gemini responds with a JSON structure:

```
{

  "agents_used": ["Web_Search"],

  "reason": "User asked for the latest news about technology."

}
```

2. Fallback Layer (Rule-based):
 If the LLM fails or response is malformed, the controller uses hard-coded logic:

  - "pdf", "document" → PDF_RAG
  - "paper", "research", "arxiv" → Arxiv_Search
  - "news", "recent", "latest" → Web_Search
  - Else defaults to Web_Search.

3. Synthesis Layer:
 If multiple agents are called, their responses are combined and summarized via Gemini.

5. Agent Details

5.1 Controller Agent

- File: agents/controller.py
- Uses google.generativeai for:
  - Routing (llm_decide)
  - Final synthesis (synthesize_answer)
- Saves logs with timestamp, query, agents, reasoning, and outputs in logs/trace.json.

5.2 PDF RAG Agent

- File: agents/pdf_agent.py
- Workflow:

    1. Extracts text from uploaded PDFs via PyMuPDF.
    2. Splits text into 500-character chunks.
    3. Embeds chunks using SentenceTransformer (all-MiniLM-L6-v2).
    4. Builds FAISS index for similarity search.
- Returns top 3 relevant passages.
- Stores data in /pdf_store.

5.3 Web Search Agent

- File: agents/web_agent.py
- Performs real-time search via DuckDuckGo Search (DDGS).
- Queries restricted to reliable safe searches
- Takes top 5 results and summarizes using Gemini LLM.

5.4 ArXiv Agent

- File: agents/arxiv_agent.py
- Uses the official arxiv Python library.
- Fetches top 3 results and summarizes titles and abstracts.

6. Logging and Traceability

Each user query produces a structured log entry stored in logs/trace.json:


```
{
  "timestamp": "2025-10-07 22:12:51",
  "query": "latest AI research",
  "decision": "LLM decision",
  "agents_used": ["Arxiv_Search"],
  "reason": "Query references research papers and ArXiv.",
  "retrieved_docs": [{"Arxiv_Search": "Title - Abstract..."}],
  "final_answer": "Summarized key findings from recent ArXiv papers."
}
```

7. Frontend Design

Technologies: HTML, CSS, JavaScript (Fetch API)

- Search Box: User enters query → calls /ask.
- PDF Upload: Uploads file → /upload_pdf.
- Results Display: Shows response, agents used, and reasoning.
- Logs Section: Fetches /logs and displays the JSON trace.

Minimal, functional, and styled for clarity.

8. Deployment

<u>Platform: Render Cloud</u>

Steps:

1. Created Dockerfile to containerize the backend.
2. Added requirements.txt with all dependencies:

   fastapi
   uvicorn[standard]
   python-dotenv
   google-generativeai==0.7.2
   PyMuPDF==1.24.5
   numpy
   faiss-cpu
   sentence-transformers
   langchain-text-splitters
   duckduckgo-search
   arxiv
   reportlab
   protobuf>=4.25.3
   python-multipart

3. Configured environment variable:

   GOOGLE_API_KEY=<your-key>

4. Exposed port 8000.
5. Render automatically detects the port from Uvicorn logs.

9. Security and Privacy

- File Handling: Uploaded PDFs are stored in the /pdfs directory and can be cleared manually when needed.
- No PII Retention: Only extracted text chunks and FAISS indexes are stored; the system does not retain any personal or sensitive data.
- Environment Variables: All API keys and sensitive configurations are managed through a .env file and environment variables. No credentials are hardcoded in the source code.
- Logging: Logs are stored locally in logs/trace.json, containing only query details, agent decisions, and timestamps — no uploaded content or personal data is recorded.

10. NebulaByte Dataset

- A sample dataset (sample_pdfs/) of 5 PDFs based on NebulaByte conversation logs was included for testing.
- These serve as domain PDFs for the RAG agent.
- Demonstrate system's ability to handle contextual retrieval and summarization.

11. Trade-offs, Limitations and Future Improvements:

Category	Description	Future Improvement
Memory Constraints (Deployment)	The current Render free-tier environment has limited RAM (512 MB). Uploading large PDFs or running LLM-based summarization can exceed this limit, causing backend restarts.	Migrate to a paid Render plan or use Hugging Face Spaces with GPU/High-RAM runtime. Implement lazy loading and streaming summarization to reduce memory footprint.
DuckDuckGo Search Reliability	The `duckduckgo-search` API sometimes returns irrelevant or non-English results due to lack of region/language filtering consistency.	Integrate SerpAPI or LangChain's Tavily tool for higher-quality and localized search results.

LLM Cost & Latency (Gemini)	Gemini-based routing and summarization occasionally experience slow response times or temporary quota limits.	Add a Groq API fallback or a local lightweight model (e.g., `Mixtral`) for offline routing.
RAG Retrieval Scalability	FAISS index performance degrades slightly with very large PDF corpora.	Use ChromaDB with metadata filtering and persistent vector storage for scalable retrieval.
Frontend Limitations	The current UI is intentionally minimal (search box + upload + logs). It lacks advanced visualization of multi-agent cooperation or intermediate reasoning.	Develop an interactive dashboard showing agent decision paths, confidence scores, and retrieved snippets.
Security / Privacy	PDFs are stored temporarily and not encrypted, which may expose sensitive data if used carelessly.	Implement temporary file auto-deletion and hashed filenames. Optionally add AES encryption for PDF cache.
Error Logging	Occasional incomplete JSON traces if Gemini output parsing fails.	Add strict schema validation and logging middleware with retry-on-failure.

12. Simplified Example Flow

<u>User Query:</u>

> "Summarize this uploaded document about AI ethics."

<u>System Flow:</u>

1. `/upload_pdf` → text extracted + indexed in FAISS.
2. `/ask` → Controller → LLM selects PDF_RAG.
3. PDF agent retrieves 3 top chunks based on the similarity measure.

4. Gemini synthesizes a coherent summary.
5. Response and reasoning logged.

13. Conclusion

This project demonstrates a scalable and explainable multi-agent AI system combining retrieval, search, and synthesis using a mix of symbolic (rules) and neural (LLM) intelligence. The architecture is modular, easily extensible, and deployable across cloud environments.