REPORT: Multi-Agentic System with Dynamic Decision Making

Project Overview

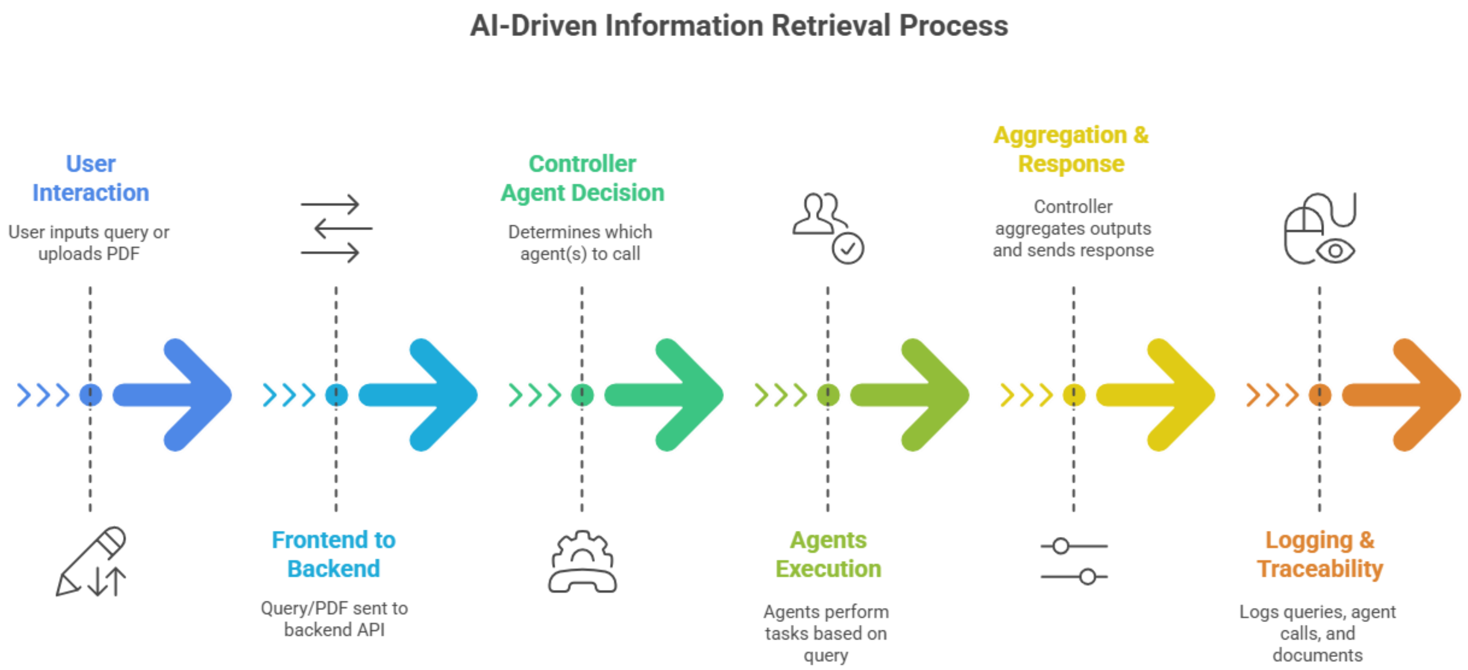
Objective:
Develop a multi-agent AI system capable of dynamically deciding which agent(s) to call based on user queries. The system integrates:

1. Controller Agent (Decision Maker): Uses an LLM (Google AI Studio).
2. PDF RAG Agent: Accepts PDFs, extracts text, chunks, embeds FAISS, retrieves relevant passages.
3. Web Search Agent: Performs real-time web search (DuckDuckGo), returns top results and short summaries.
4. ArXiv Agent: Queries ArXiv API to fetch and summarize recent papers.

Key Features:

- Dynamic query routing using rules + LLM reasoning.
- Secure PDF ingestion and retrieval.
- Multi-source information aggregation (Web, ArXiv, PDFs).
- Logging and traceability of all agent decisions.
- Frontend with search box, PDF upload, and results display.
- Backend deployed on Hugging Face Spaces. Click here

Architecture



1. User Interaction:
    - The user interacts via the frontend UI which includes a search box for queries and a PDF upload widget for document ingestion.
2. Frontend → Backend:
    - Queries or uploaded PDFs are sent to the backend API implemented with Flask
    - Backend handles request validation, security checks, and forwards the query to the Controller Agent.
3. Controller Agent:
    - Decision-making hub: determines which agent(s) to call.
    - Rule-based logic + LLM reasoning:
        - PDF present + “summarize” → PDF RAG Agent.
        - Query mentions “recent papers” → ArXiv Agent.
        - Query mentions “latest news” → Web Search Agent.
        - Multiple criteria → combination of agents.
    - Logs decisions: input query, rationale, agents called, document IDs, timestamp.
4. Agents Execution:
    - PDF RAG Agent: Extracts text from PDFs, chunks content, generates embeddings (FAISS), retrieves relevant passages.
    - Web Search Agent: Performs real-time search via DuckDuckGo, summarizes top results.
    - ArXiv Agent: Queries ArXiv API, retrieves abstracts, and generates summaries.
5. Aggregation & Response:
    - Controller aggregates all agent outputs.
    - Final synthesized answer is sent back to the frontend, along with a summary of which agents were used and decision rationale.
6. Logging & Traceability:
    - Every query, agent call, and retrieved document is logged for traceability and debugging.
    - Logs accessible via /logs endpoint or saved in /logs

Agent Interfaces

- Controller Agent
    - Input: User query + optional uploaded PDFs.
    - Function: Decide which agent(s) to call using.

- Output: Decision rationale, agents invoked, timestamp, and documents retrieved.

➢ PDF RAG Agent
  - Input: PDF file(s).
  - Process:
    - Extract text using PyPDFLoader.
    - Chunk text into passages.
    - Generate embeddings with FAISS.
    - Retrieve relevant passages based on user query.
    - Output: Summarized passage snippets.

➢ Web Search Agent
  - Input: Query with keywords like “latest news”, “recent developments”.
  - Process:
    - Call DuckDuckGo.
    - Summarize top 3 results.
  - Output: Short summaries of relevant web pages.

➢ ArXiv Agent
  - Input: Queries requesting like “recent papers”.
  - Process:
    - Query ArXiv API.
    - Retrieve abstracts and summarize.
  - Output: Summaries of top matching papers.

Controller logic:

Step 1: Decision Phase (LLM Call 1 - Routing)

This stage uses the Gemini model to perform a **task-to-tool routing decision**.

1. **System Prompting:** A strict system_instruction is provided to the LLM. This prompt clearly defines the LLM's role (Central Controller), the goal (analyze and route), and the constraints.
2. **Tool Provision:** The Pydantic AgentCall model is passed to the LLM via the tools configuration.
3. **LLM Execution:** The LLM receives the user_query and the constraints. It then performs two actions based on its training:

- **Reasoning (Thought):** It determines which specialized agent (pdf_rag, web_search, or arxiv) is the most appropriate based on the nature of the query (e.g., "current news" web search).
- **Function Call:** It formats its output into a function_calls block that strictly adheres to the Agent Call schema, providing the chosen agent_name and an **optimized search query** (agent_query) derived from the user's initial question.

4. **Parsing:** The code attempts to extract the agent name and agent query from the response function calls.

Step 2: Agent Execution Phase (Orchestration)

This stage involves handing the task off to the chosen specialized agent.

1. **Agent Lookup:** The agent_name is used as a key to retrieve the correct, initialized agent instance from the self.agent_map.
2. **Agent Invocation:** The controller calls the standardized .run() method on the specialized agent, passing the refined agent_query:
3. **Result Capture:** The raw output from the specialized agent (agent_result) is captured.

Step 3: Aggregation Phase (LLM Call 2 - Final Answer Synthesis)

The raw output from the specialized agent is often verbose or contains internal formatting (like source links). This stage ensures a polished, high-quality final answer for the end-user.

1. **Final Prompt Construction:** A new prompt (final_prompt) is created. This prompt is a form of **Contextual Reinforcement**, providing the LLM with:
    - The **original** user_query.
    - The **raw output** (agent_result) from the specialized agent.
    - A clear instruction to *analyze* the result and *formulate* a concise, professional answer.
2. **LLM Execution:** The model (Gemini) takes the original query and the long context block, and its goal is solely to act as a **professional summarizer/formatter**. It uses the provided context to strip away internal formatting and synthesize the final user-facing response.
3. **Final Return:** The synthesized text (final_answer) is returned to the user, completing the entire workflow.

Logging & Traceability

- All actions logged in /logs file or via /logs endpoint.

- **Logged Data:** query, decision, rationale, agents used, document IDs, final answer, timestamp.
- **Sample Logs:**

```
[
    {
        "id": "ba500fec-e982-4504-a20f-e834ddde67e4",
        "timestamp": "2025-10-05 17:16:26.496",
        "input_query": "alice in borderland web series",
        "controller_reasoning": "Determined by LLM-based routing logic (Agent selection is the reasoning).",
        "agents_called": "web_search",
        "docs_retrieved_info": "",
        "final_synthesized_answer": "\"Alice in Borderland\" (Japanese: \u4eca\u969b\u306e\u56fd\u306e\u30a2\u30ea\u30b9, Imawa no Kuni no Arisu) is a Japanese science fiction thriller drama television series based on the manga of the same name by Haro Aso. Directed by Shinsuke Sato, the series follows Arisu, a listless, jobless, and video-game-obsessed young man, who, along with his friends, finds himself in a strange, emptied-out version of Tokyo. In this parallel world, they are forced to compete in a series of dangerous and sadistic games to survive. The show is known ..."
    }
]
```

Trade-offs & Limitations

Aspect	Decision	Trade-off
PDF RAG	FAISS embeddings	Fast retrieval vs higher memory usage
Controller Logic	Rules + LLM	Reliable but may miss subtle intents
Web/ArXiv Agents	API-based	Dependent on rate limits & external availability
Deployment	HF Spaces / Render	Simpler hosting vs limited customization

Limitations:

- Only demo PDFs indexed.
- No authentication for public demo.
- Real-time web search latency can vary.

File Structure:

```
/flask_multi_agentic_system
  ├─ app.py
  ├─ agents/
```

```
│     ├─ controller_agent.py
│     ├─ pdf_rag_agent.py
│     ├─ web_search_agent.py
│     └─ arxiv_agent.py
├─ sample_pdfs/
│     ├─ dialog_001.pdf
│     └─ ...
├─ frontend/
│     └─ index.html
      └─ script.js
      └─ style.css

└─ logs/
      └─ decision_logs.json
```

Conclusion

- Multi-agent system demonstrates **dynamic query routing** using **LLM reasoning + rules**.
- Modular design allows easy addition of agents.
- Secure PDF handling, logging, and deployed frontend ensure usability and transparency.
- The system can integrate real-time web search, academic paper retrieval, and PDF summarization in one unified platform.