Multi-Agentic System with Dynamic Decision Making

Executive Summary

This report provides an overview of the multi-agent AI system built using Flask, Groq LLM, and specialized agents for handling queries related to PDFs, web searches, and ArXiv research. The system routes user queries intelligently, processes them via agents, and synthesizes responses. Key components include a controller agent for routing and synthesis, along with domain-specific agents. The system supports PDF uploads for RAG-based querying and includes logging for decisions and results. Sample PDFs themed around "NebulaByte" (fictional AI discussions) are auto-generated for demonstration.

Architecture Overview

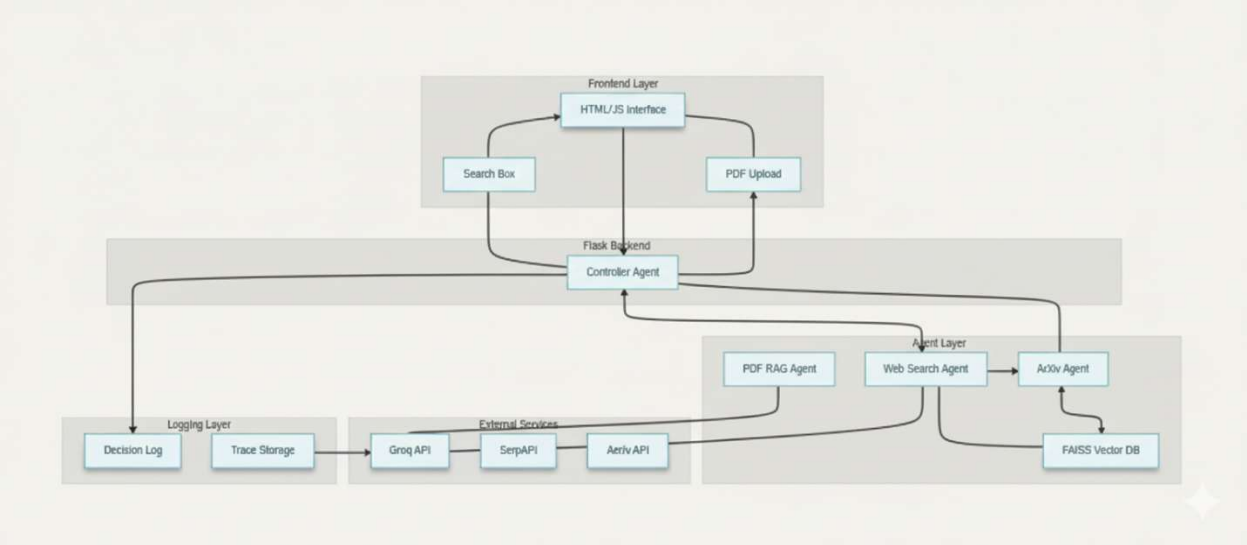
The system follows a modular, agent-based architecture where a central controller orchestrates interactions between specialized agents. User queries enter via a Flask web app, are routed by the controller (using LLM or rules), processed in parallel by selected agents, and synthesized into a final response. Data flows include PDF ingestion into a FAISS vector store for RAG, web/ArXiv searches via APIs, and logging to files.

Key components:

- **Frontend**: HTML-based UI for query input, PDF upload, and log viewing.
- **Backend (Flask App)**: Handles API routes, file uploads, and agent orchestration.
- **Controller Agent**: Routes queries and synthesizes answers using Groq LLM.
- **Specialized Agents**: PDF RAG (vector search), Web Search (DuckDuckGo/SerpAPI), ArXiv (API-based).
- **Storage**: FAISS for vector embeddings, file-based logs, and temporary uploads.
- **Dependencies**: Groq for LLM, SentenceTransformers for embeddings, FAISS for vector DB, PdfPlumber for PDF handling.

Data flow:

1. User submits query via /ask or uploads PDF via /upload_pdf.
2. Controller routes to agents (e.g., pdf_rag for document queries).
3. Agents process independently (e.g., embed and query vectors for PDFs).
4. Controller synthesizes results.
5. Logs are stored and retrievable via /logs.



(Diagram Description: A high-level UML or flowchart showing Flask App as entry point, arrow to Controller Agent, branching arrows to PDFRAGAgent, WebSearchAgent, ArxivAgent. Return arrows to Controller for synthesis. Side elements: FAISS DB for PDFs, Logs folder. Tools like Groq LLM and external APIs connected via lines.)

Agent Interfaces

Each agent exposes a primary process_query(query: str) -> Dict[str, Any] method that takes a string query and returns a structured dictionary with 'answer', 'sources', 'agent', and optional metadata (e.g., 'num_results'). Agents are initialized in app.py and called conditionally based on controller decisions. Ingestion methods (e.g., for PDFs) are separate.

- **ControllerAgent** (controller_agent.py):
    - route_query(query: str) -> Dict[str, Any]: Returns {'agents': List[str], 'reasoning': str}. Uses LLM prompt for decision or rule-based fallback.
    - synthesize_answer(query: str, agent_results: Dict[str, Any]) -> str: Combines agent outputs into a coherent response using LLM.
    - Interface: Not query-processing; orchestrates others.
- **PDFRAGAgent** (pdf_rag_agent.py):
    - ingest_pdf(pdf_path: str) -> bool: Extracts text chunks, embeds with SentenceTransformer ('all-MiniLM-L6-v2'), adds to FAISS index (384 dims). Saves index/metadata to rag_data/.
    - process_query(query: str) -> Dict: Embeds query, searches FAISS (top-5, uses top-3 for context), generates answer via Groq LLM prompt. Returns answer and sources (with filename, page, score).
    - Additional: retrieve_relevant_docs(query: str, k: int=5) -> List[Dict]: FAISS search helper.

- **WebSearchAgent** (web_search_agent.py):
    - process_query(query: str) -> Dict: Searches via DuckDuckGo (primary) or SerpAPI (fallback if key set). Uses top-5 results for context, synthesizes answer via Groq LLM. Returns answer and sources (title, link, snippet).
    - Helpers: search_duckduckgo(query: str, max_results: int=5), search_serpapi(...) – return List[Dict].
- **ArxivAgent** (arxiv_agent.py):
    - process_query(query: str) -> Dict: Queries ArXiv API (export.arxiv.org), parses XML for top-5 papers (sorted by last updated). Uses top-3 for context, synthesizes via Groq LLM. Returns answer and sources (title, summary, published, authors, pdf_url).
    - Helper: search_arxiv(query: str, max_results: int=5) -> List[Dict]: API call and XML parsing.

All agents use Groq's "llama-3.1-8b-instant" model (temperature 0.3 for generation, 0.1 for routing) and handle errors by logging and returning error dicts.

Controller Logic

The ControllerAgent handles routing and synthesis.

Routing Logic

- Primary: LLM-based using Groq prompt (temperature 0.1 for determinism).
- Fallback: Rule-based if LLM fails (e.g., JSON parse error).
- Output: JSON dict with 'agents' (list of strings: 'pdf_rag', 'web_search', 'arxiv') and 'reasoning'.

LLM Prompt for Routing:

```
You are a controller agent that decides which specialized agents should
handle a user query.
Available agents:
1. pdf_rag - For questions about uploaded PDF documents
2. web_search - For current information, news, recent developments
3. arxiv - For academic papers, research, scientific information

Query: "{query}"

Analyze the query and decide which agent(s) should handle it. Consider:
- If it mentions "recent", "latest", "news", "current" -> web_search
- If it mentions "paper", "research", "academic", "arxiv" -> arxiv
- If it's about uploaded documents or asks to "summarize this" ->
pdf_rag
```

```
- You can select multiple agents if needed

Respond with JSON in this format:
{{
    "agents": ["agent1", "agent2"],
    "reasoning": "Brief explanation of why these agents were selected"
}}
System message: "You are a routing agent. Always respond with valid
JSON."
```

Rule-Based Fallback (_rule_based_routing):

- Lowercase query check for keywords:
    - ['recent', 'latest', 'news', 'current', 'today'] → 'web_search'
    - ['paper', 'research', 'academic', 'arxiv', 'study'] → 'arxiv'
    - ['document', 'pdf', 'summarize this', 'uploaded'] → 'pdf_rag'
- Default: 'web_search' if no matches.
- Reasoning string built from detections.

Synthesis Logic

- Uses Groq LLM (temperature 0.3) to integrate agent results.
- Prompt includes query and JSON-dumped agent_results.
- Fallback: Simple concatenation if LLM fails.

LLM Prompt for Synthesis:

```
You are an AI assistant that synthesizes information from multiple
specialized agents.
User Query: "{query}"
Agent Results:
{json.dumps(agent_results, indent=2)}
Based on the information provided by the agents, create a comprehensive
and coherent answer to the user's query.
If multiple agents provided information, integrate their responses
seamlessly.
If an agent encountered an error, acknowledge it briefly but focus on
available information.
```

Provide a clear, helpful response that directly addresses the user's question.

Trade-Offs

- **Performance vs. Cost**: Groq LLM is fast/low-latency but incurs API costs; rule-based fallback reduces LLM calls. DuckDuckGo is free but less reliable than paid SerpAPI (fallback only if key set).
- **Accuracy vs. Simplicity**: FAISS for RAG is lightweight/fast but lacks advanced features (e.g., hybrid search). Simple sentence-based chunking in PDFs may miss context; no overlap/advanced splitting.
- **Scalability**: File-based FAISS/logs work for small-scale but not production (e.g., no distributed storage). Max 16MB PDF uploads limits large docs.
- **Privacy/Security**: PDFs deleted post-ingestion (retention policy), but embeddings persist. No auth on routes (e.g., /logs exposes decisions).
- **Flexibility vs. Maintenance**: Modular agents ease extension but require consistent interfaces. Sample PDFs auto-generate for demo but may confuse real use.
- **Error Handling**: Graceful (error dicts/logs), but no retries on API failures. Temperature settings balance creativity/determinism.
- **Dependencies**: Relies on external APIs (Groq, ArXiv, DDG/SerpAPI) – outages could break agents.

Deployment Notes

- **Runtime**: Flask app runs on host '0.0.0.0', port 7860 (env PORT override). Debug mode if FLASK_ENV='development'.
- **Environment**: Requires Python libs (Flask, Groq, sentence-transformers, faiss, fitz, requests, xml.etree, ddgs). Set GROQ_API_KEY (hardcoded in code – insecure; use env). Optional SERPAPI_KEY for web fallback.
- **Setup**: Creates dirs (uploads, logs, sample_pdfs, rag_data). Auto-ingests samples if FAISS empty.
- **Scaling**: Single-threaded; use Gunicorn/WSGI for prod. Add auth (e.g., JWT). Monitor Groq usage for costs.
- **Testing**: Pytest in test_app.py (basic routes, agent methods). /test_logs creates sample logs.
- **Monitoring**: Logs to app.log/decisions.log/agent_results.log. View via /logs (last 50).
- **Prod Tips**: Use Docker; env vars for keys; persistent volume for rag_data (embeddings). HTTPS via nginx. Limit max_results to control costs.

How NebulaByte PDFs Were Generated & Used

"NebulaByte" is a fictional theme in sample PDFs, representing AI-related dialogs (e.g., ethics, ML trends). They are generated for demo if FAISS index is empty (checked in PDFRAGAgent.**init** via _ingest_sample_pdfs).

Generation (in _create_sample_pdfs):

- Uses PdfPlumber to create PDFs dynamically.
- 5 samples: ai_ethics_dialog.pdf, machine_learning_trends.pdf, neural_networks_basics.pdf, deep_learning_applications.pdf, ai_future_predictions.pdf.
- Each has hardcoded text content (dialogs on AI topics).
- Process: Open new doc, add page, insert text in rect (fontsize=12, helv font), save to sample_pdfs/.
- Only creates if file missing; logs creation.

Usage:

- Ingested via ingest_pdf if index empty: Extracts text (page-by-page, sentence-split chunks <1000 chars), embeds, adds to FAISS.
- In queries: If routed to pdf_rag (e.g., document-related), retrieves via vector similarity, uses in RAG prompt.
- Purpose: Bootstrap knowledge base for testing (e.g., query "What are ethical considerations in AI?" hits ai_ethics_dialog).
- Retention: Original PDFs kept in sample_pdfs/; embeddings in rag_data/. User uploads deleted post-ingest.