■ Multi-Agent AI System Overview

This document describes the design and structure of a multi-agent AI system built using LangGraph, FastAPI, and Next.js.

General Overview
- **Backend**: LangGraph + FastAPI (agents, routing, orchestration).
- **Frontend**: Next.js + TailwindCSS (dashboard + visualizations).
- **Storage**: Vector Database (FAISS / Pinecone / Weaviate).
- **Deployment**: Docker + Vercel (frontend) + Render/Railway/AWS (backend).

Workflow
1. User submits a query via Next.js dashboard.
2. FastAPI receives the query and routes it to the **Decision Agent**.
3. The Decision Agent selects the appropriate specialist agent:
- **Research Agent**: Queries documents via RAG.
- **News Agent**: Fetches live data via NewsAPI.
- **Sentiment Agent**: Analyzes sentiment (Positive/Negative/Neutral).
4. The **Summarizer Agent** condenses outputs into short insights.
5. The **Frontend Agent** formats results into structured JSON for the Next.js dashboard.
6. The **Documentation Agent** auto-updates system documentation in Markdown.

Agents
1. **Decision Agent (Router)**
- Routes user queries to the correct specialist agent.

2. **Research Agent (RAG)**
- Retrieves answers from knowledge base (PDF, TXT, DB).
- Output Schema:
```
{
"type": "document_answer",
"answer": "...",
"sources": ["doc1.pdf"]
}
```

3. **News Agent**
- Fetches and summarizes live news.
- Output Schema:
```
{
"type": "news_summary",
"articles": [{ "headline": "..." }]
}
```

4. **Sentiment Analysis Agent**
- Classifies text sentiment (Positive, Negative, Neutral).
- Output Schema:
```
{
"type": "sentiment_chart",
"data": [
{ "label": "Positive", "value": 10 },
{ "label": "Negative", "value": 5 },
{ "label": "Neutral", "value": 3 }
]
```

}

5. **Summarizer Agent**
- Condenses multiple agent outputs into insights.

6. **Frontend Agent**
- Converts results into JSON-ready format for Next.js.
- Output Schema:
{
"type": "dashboard_payload",
"widgets": [...]
}

7. **Documentation Agent**
- Generates and updates documentation of agents, schemas, and workflows in Markdown.

```
### Suggested Folder Structure

Backend (FastAPI + LangGraph)
backend/
■■■ main.py
■■■ requirements.txt
■■■ Dockerfile
■■■ agents/
■■■ prompts/
■■■ schemas/
■■■ utils/

Frontend (Next.js + Tailwind)
frontend/
■■■ app/
■■■ components/
■■■ lib/
■■■ styles/
```

Execution Example
1. User asks: "What is the sentiment of recent AI news?"
2. Decision Agent selects News + Sentiment.
3. News Agent fetches articles.
4. Sentiment Agent classifies tone.
5. Summarizer condenses insights.
6. Frontend Agent returns JSON payload.
7. Next.js dashboard renders charts and summaries.
8. Documentation Agent updates docs.md automatically.

Deployment as a Service

To make the Multi-Agent System a production-ready service, containerization and deployment steps ar

1. Backend (FastAPI + LangGraph)
- Wrap agents and orchestration into FastAPI endpoints (already present).
- Add a Dockerfile:

```
FROM python:3.11-slim

WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

2. Frontend (Next.js + Tailwind)
- Configure API calls to use `NEXT_PUBLIC_API_URL` for the backend endpoint.
- Example in `frontend/lib/api.js`:

```
export async function fetchResult(query) {
  const res = await fetch(process.env.NEXT_PUBLIC_API_URL + "/query", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ query }),
  });
  return res.json();
}
```

3. Docker Compose (Full System)
To orchestrate backend + frontend together:

```
version: "3.9"
services:
  backend:
    build: ./backend
    ports:
      - "8000:8000"

  frontend:
    build: ./frontend
    ports:
      - "3000:3000"
    environment:
```

Vector Database: Weaviate Cloud (WCS)

For production-ready deployment of the Research Agent (RAG), Weaviate Cloud (WCS) is recommend

Why Weaviate Cloud?
- Managed service: no version conflicts or infra management.
- Hybrid search: semantic + keyword matching.
- Scalable: handles millions of vectors efficiently.
- Open-source core: flexibility to run locally during development.

Setup Instructions

1. **Sign up for WCS**
   - Go to: https://weaviate.io/developers/weaviate/cloud
   - Create a cluster and get your endpoint + API key.

2. **Connect in Python**
```
import weaviate

client = weaviate.Client(
    url="https://YOUR-WEAVIATE-ENDPOINT.weaviate.network",
    auth_client_secret=weaviate.AuthApiKey(api_key="YOUR_API_KEY")
)
```

3. **Store Embeddings**
   - Use OpenAI or HuggingFace embeddings with LangChain.
   - Example schema: `Document` with fields `title`, `content`, `source`.

4. **Querying in Research Agent**
   - Instead of FAISS, query Weaviate for relevant docs:
```
result = client.query.get("Document", ["title", "content", "source"])     .with_near_text({"concepts": ["AI ne
```

5. **Recommended Flow**
   - Local development → FAISS / Weaviate Docker.
   - Production → Weaviate Cloud.

This ensures reliable, scalable, and version-free vector search integrated with the Multi-Agent System.