LifeHub Agent

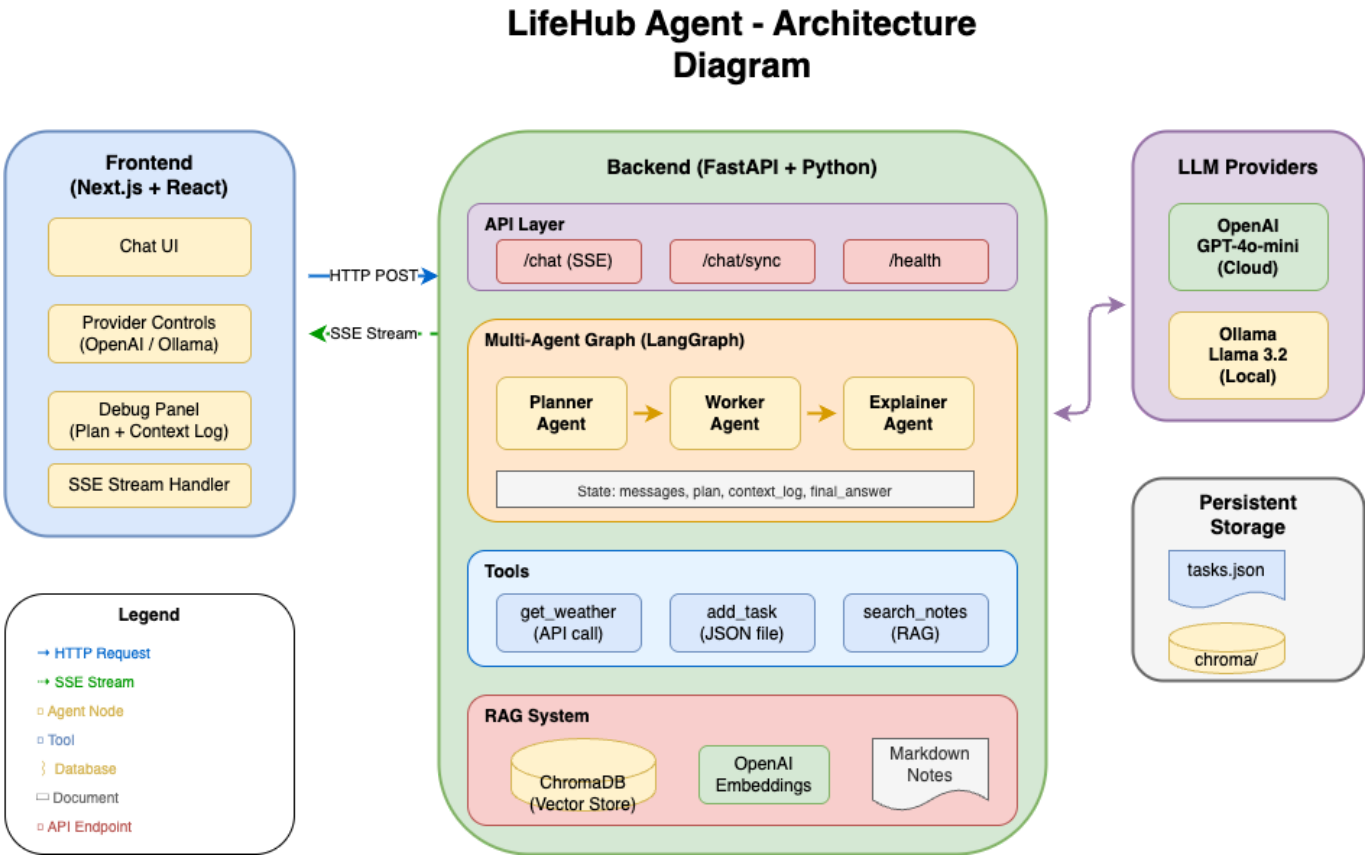
An AI-powered personal assistant demonstrating modern LLM application architecture with **multi-agent orchestration**, **RAG (Retrieval-Augmented Generation)**, **tool calling**, and **streaming responses**.

🎯 Overview

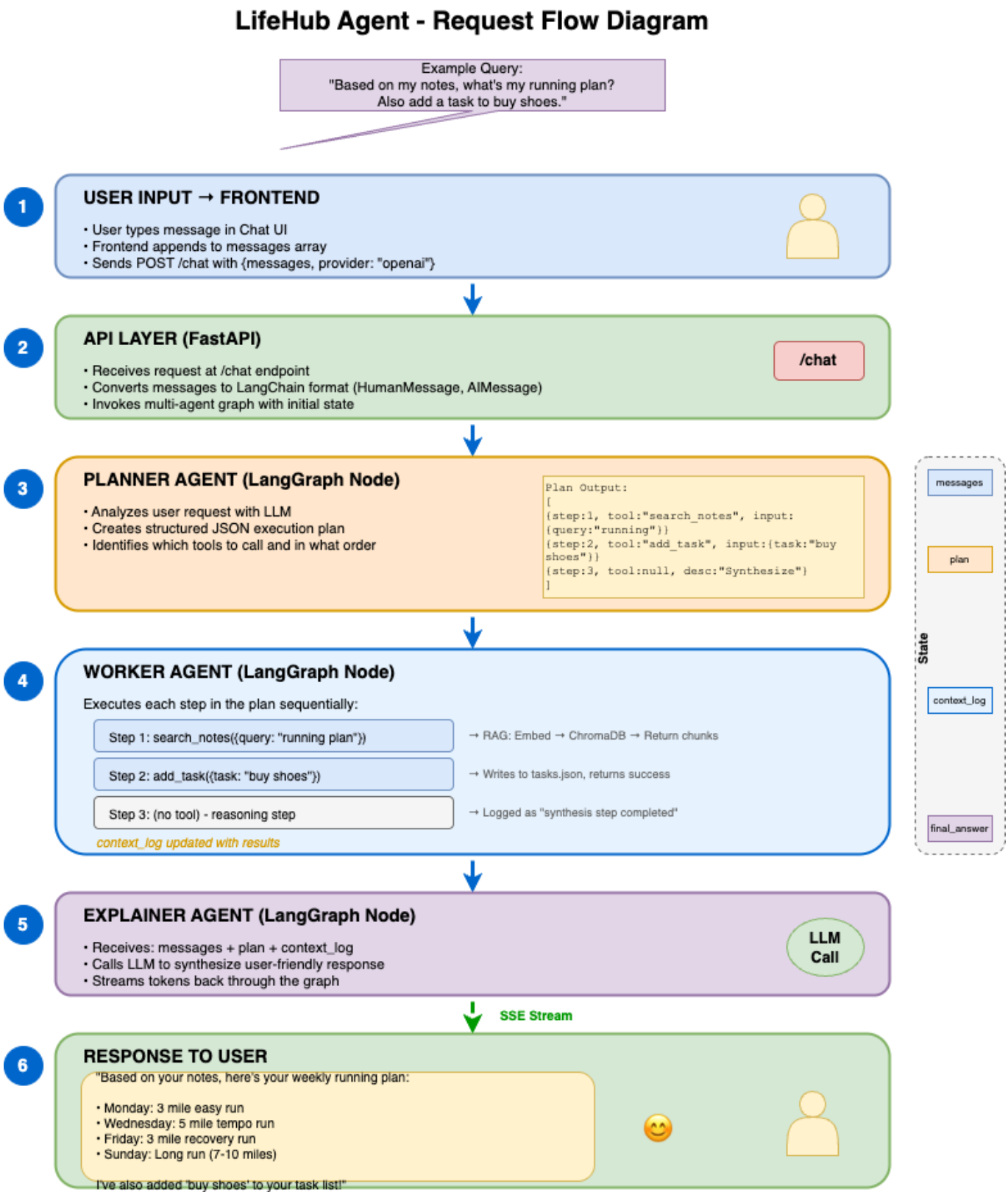
LifeHub Agent is a full-stack AI application that showcases how to build production-ready LLM applications. It features:

- **Multi-Agent Architecture**: Planner → Worker → Explainer pipeline
- **RAG System**: Search personal notes using vector embeddings
- **Tool Calling**: Weather lookup, task management, notes search
- **Streaming**: Real-time token streaming via Server-Sent Events (SSE)
- **Provider Flexibility**: Switch between OpenAI and Ollama (local)

🏗️ Architecture Diagram



🔄 Request Flow Diagram



🧠 AI Concepts Explained

1. Large Language Models (LLMs)

What: Neural networks trained on vast text data to understand and generate human language.

In this project:

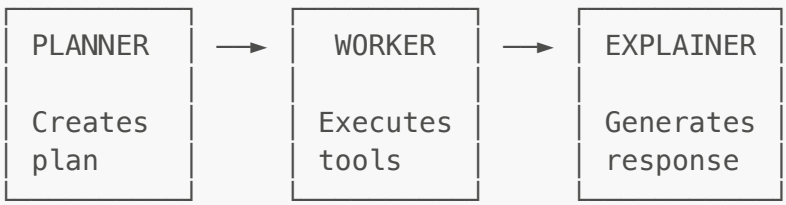
- **OpenAI GPT-4o-mini**: Cloud-hosted, high-quality responses
- **Ollama (Llama 3.2)**: Local model for privacy/offline use

```
# backend/models.py
def get_model_client(provider: str = "openai"):
    if provider == "ollama":
        return ChatOpenAI(base_url="http://localhost:11434/v1",
model="llama3.2")
    return ChatOpenAI(model="gpt-4o-mini")
```

2. Multi-Agent Architecture

What: Multiple specialized AI agents working together, each with a specific role.

In this project:



- **Planner**: Analyzes request, outputs structured JSON plan
- **Worker**: Executes plan steps, calls tools, logs results
- **Explainer**: Synthesizes everything into user-friendly response

```
# backend/agents/graph.py
graph.add_edge(START, "planner")
graph.add_edge("planner", "worker")
graph.add_edge("worker", "explainer")
graph.add_edge("explainer", END)
```

3. Tool Calling / Function Calling

What: LLMs can invoke external functions to perform actions or retrieve data.

In this project:

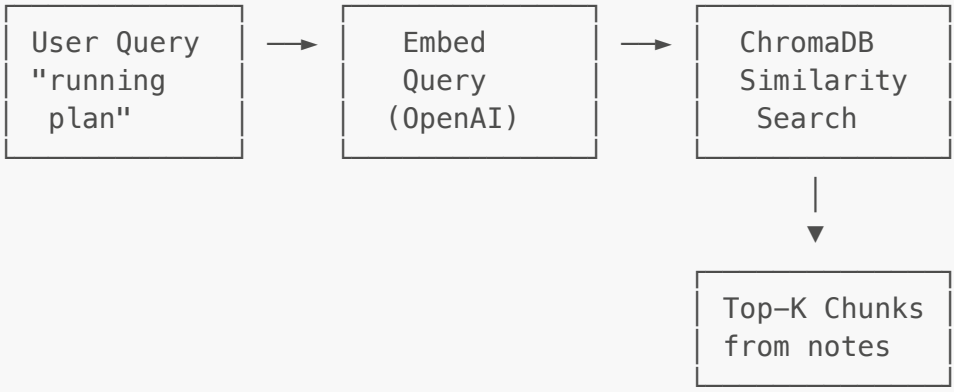
Tool	Purpose	Implementation
`get_weather`	Weather lookup	Returns mock weather data
`add_task`	Task management	Writes to `tasks.json`
`search_notes`	RAG search	Queries ChromaDB vector store

```python
# backend/tools/weather.py
@tool
def get_weather(city: str) -> dict:
    """Get current weather for a city."""
    return {"city": city, "temp": "72F", "conditions": "clear"}
```

4. RAG (Retrieval-Augmented Generation)

What: Enhancing LLM responses with relevant information retrieved from a knowledge base.

In this project:



Components:

- **Ingestion** (`backend/rag/ingest_notes.py`): Chunks markdown files, embeds with OpenAI, stores in ChromaDB
- **Retrieval** (`backend/tools/notes.py`): Embeds query, finds similar chunks
- **Vector Store** (`backend/rag/store.py`): ChromaDB persistent storage

```
# backend/tools/notes.py
@tool
def search_notes(query: str) -> list[dict]:
    """Search personal notes using semantic similarity."""
    query_embedding = embed_text(query)
    results = collection.query(query_embeddings=[query_embedding],
n_results=5)
    return format_results(results)
```

5. LangGraph State Machine

What: A framework for building stateful, multi-step LLM applications as directed graphs.

In this project:

```
# State flows through the graph
class MultiAgentState(TypedDict):
    messages: list[AnyMessage]      # Conversation history
    plan: list[PlanStep] | None     # Planner output
    context_log: list[ContextLogEntry]  # Worker results
    final_answer: str | None        # Explainer output
```

6. Streaming (Server-Sent Events)

What: Real-time token-by-token delivery of LLM responses.

In this project:

```
# backend/app/main.py
async def stream_response(messages, provider):
    async for event in agent_graph.astream_events(input, version="v2"):
        if event["event"] == "on_chat_model_stream":
            token = event["data"]["chunk"].content
            yield f"data: {json.dumps({'type': 'token', 'content':
token})}\n\n"
```

📁 Project Structure

```
lifehub-agent/
├── backend/
│   ├── app/
│   │   └── main.py              # FastAPI app, /chat & /chat/sync
endpoints
│   ├── agents/
│   │   ├── graph.py             # Multi-agent LangGraph
(Planner→Worker→Explainer)
│   │   └── graph_legacy.py      # Original single-agent implementation
│   ├── tools/
│   │   ├── weather.py           # get_weather tool
│   │   ├── tasks.py             # add_task tool
│   │   └── notes.py             # search_notes RAG tool
│   ├── rag/
│   │   ├── store.py             # ChromaDB vector store setup
│   │   └── ingest_notes.py      # Notes ingestion script
│   ├── notes/                   # Markdown notes for RAG
│   │   ├── fitness_example.md
│   │   └── recipes_example.md
│   ├── state/                   # Persistent storage
│   │   ├── tasks.json           # Task list
│   │   └── chroma/              # ChromaDB vector database
│   └── models.py                # LLM client factory (OpenAI/Ollama)
├── frontend/
│   ├── src/
│   │   ├── app/
│   │   │   ├── page.tsx         # Main chat UI
│   │   │   ├── layout.tsx       # App layout
│   │   │   └── globals.css      # Tailwind styles
│   │   └── config.ts            # Backend URL config
│   ├── vercel.json              # Vercel deployment config
│   └── package.json
├── render.yaml                  # Render deployment config
├── requirements.txt             # Python dependencies
├── pyproject.toml               # Project config (uv)
├── DEPLOY.md                    # Deployment instructions
└── README.md                    # This file
```

🚀 Quick Start

Prerequisites

- Python 3.11+
- Node.js 18+
- uv package manager
- OpenAI API key

Backend Setup

```
cd lifehub-agent

# Install dependencies
uv sync

# Set API key
export OPENAI_API_KEY="your-key"

# Ingest notes into vector store
uv run python -m backend.rag.ingest_notes

# Start server
uv run uvicorn backend.app.main:app --reload --port 8000
```

Frontend Setup

```
cd frontend
npm install
npm run dev
```

Open http://localhost:3000

🔌 API Endpoints

Endpoint	Method	Description
`/health`	GET	Health check
`/chat`	POST	Streaming chat (SSE)
`/chat/sync`	POST	Non-streaming chat with debug option

Example: Streaming Chat

```
curl -N -X POST http://localhost:8000/chat \
  -H "Content-Type: application/json" \
  -d '{"messages": [{"role": "user", "content": "What is the weather in
Tokyo?"}], "provider": "openai"}'
```

Example: Debug Mode

```
curl -X POST http://localhost:8000/chat/sync \
  -H "Content-Type: application/json" \
  -d '{"messages": [{"role": "user", "content": "Search my notes for
fitness info"}], "provider": "openai", "debug": true}'
```

🌐 Deployment

Component	Platform	URL
Backend	Render	`https://your-app.onrender.com`
Frontend	Vercel	`https://your-app.vercel.app`

See DEPLOY.md for detailed deployment instructions.

🛠️ Tech Stack

Layer	Technology
LLM Framework	LangGraph, LangChain
LLM Providers	OpenAI GPT-4o-mini, Ollama
Vector Store	ChromaDB
Embeddings	OpenAI text-embedding-3-small
Backend	FastAPI, Python 3.11
Frontend	Next.js 16, React, TypeScript, Tailwind CSS
Deployment	Render (backend), Vercel (frontend)

📚 Key Files Reference

File	Purpose
`backend/agents/graph.py`	Multi-agent orchestration logic
`backend/tools/notes.py`	RAG search implementation
`backend/rag/store.py`	ChromaDB setup
`backend/app/main.py`	API endpoints + streaming
`frontend/src/app/page.tsx`	Chat UI component

📄 License

MIT