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**.

Architecture Multi-Agent LLM OpenAI | Ollama Framework LangGraph Backend FastAPI Frontend Next.js

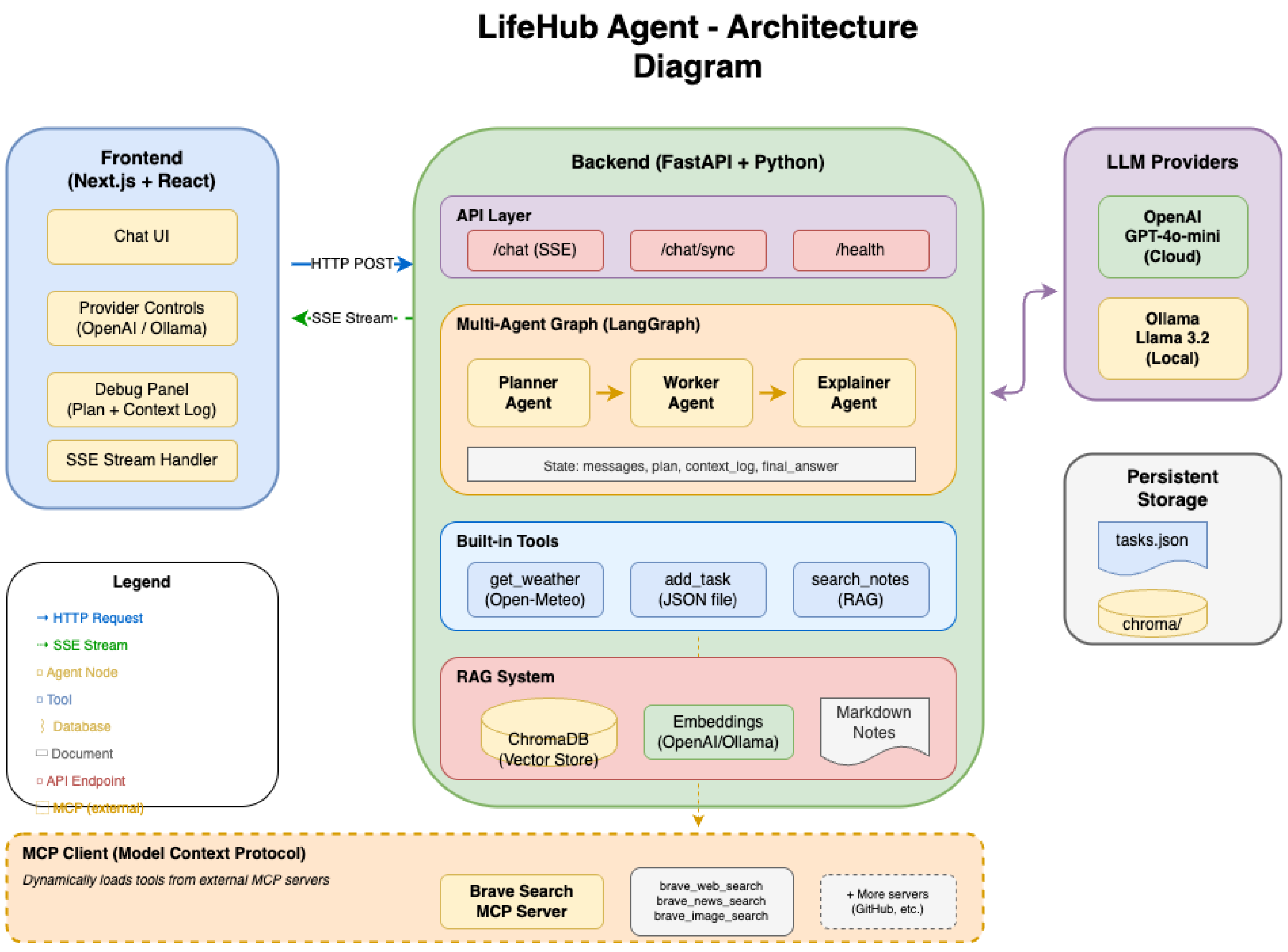
---

🎯 Overview

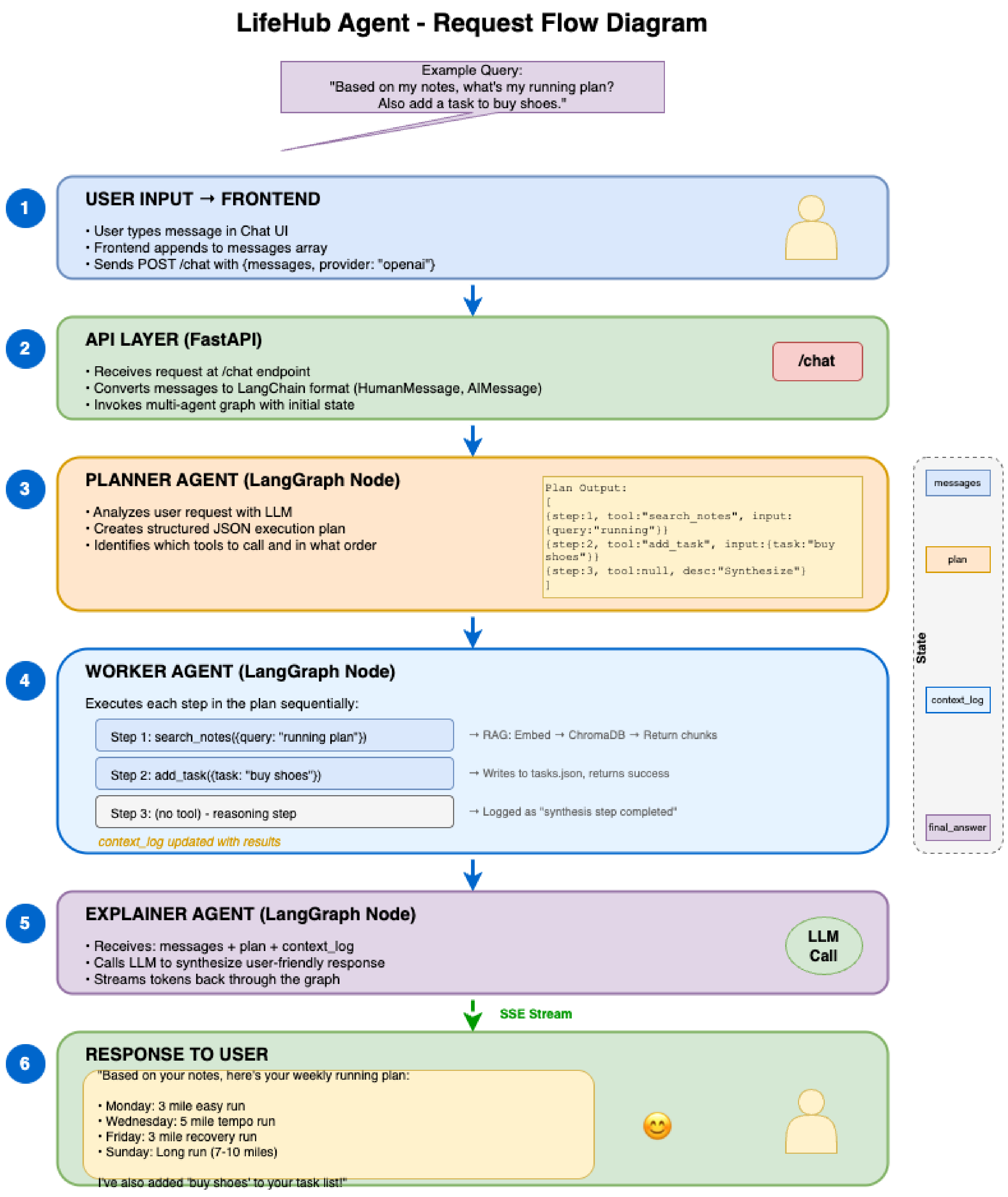
LifeHub Agent is a full-stack AI application that showcases how to build production-ready LLM applications. It features:

- **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


LifeHub Agent - Request Flow Diagram
1
USER INPUT → FRONTEND
2
API LAYER (FastAPI)
/chat
3
PLANNER AGENT (LangGraph Node)
4
WORKER AGENT (LangGraph Node)
5
EXPLAINER AGENT (LangGraph Node)
LLM Call
SSE Stream
6
RESPONSE TO USER
State
messages
plan
context_log
final_answer

Example Query:
"Based on my notes, what's my running plan?
Also add a task to buy shoes."

• User types message in Chat UI
• Frontend appends to messages array
• Sends POST /chat with {messages, provider: "openai"}

• Receives request at /chat endpoint
• Converts messages to LangChain format (HumanMessage, AIMessage)
• Invokes multi-agent graph with initial state

• Analyzes user request with LLM
• Creates structured JSON execution plan
• Identifies which tools to call and in what order

```
Plan Output:
[
{step:1, tool:"search_notes", input:
{query:"running"}}
{step:2, tool:"add_task", input:{task:"buy
shoes"}}
{step:3, tool:null, desc:"Synthesize"}
]
```

Executes each step in the plan sequentially:

Step 1: search_notes({query: "running plan"}) → RAG: Embed → ChromaDB → Return chunks
Step 2: add_task({task: "buy shoes"}) → Writes to tasks.json, returns success
Step 3: (no tool) - reasoning step → Logged as "synthesis step completed"

context_log updated with results

• Receives: messages + plan + context_log
• Calls LLM to synthesize user-friendly response
• Streams tokens back through the graph

"Based on your notes, here's your weekly running plan:

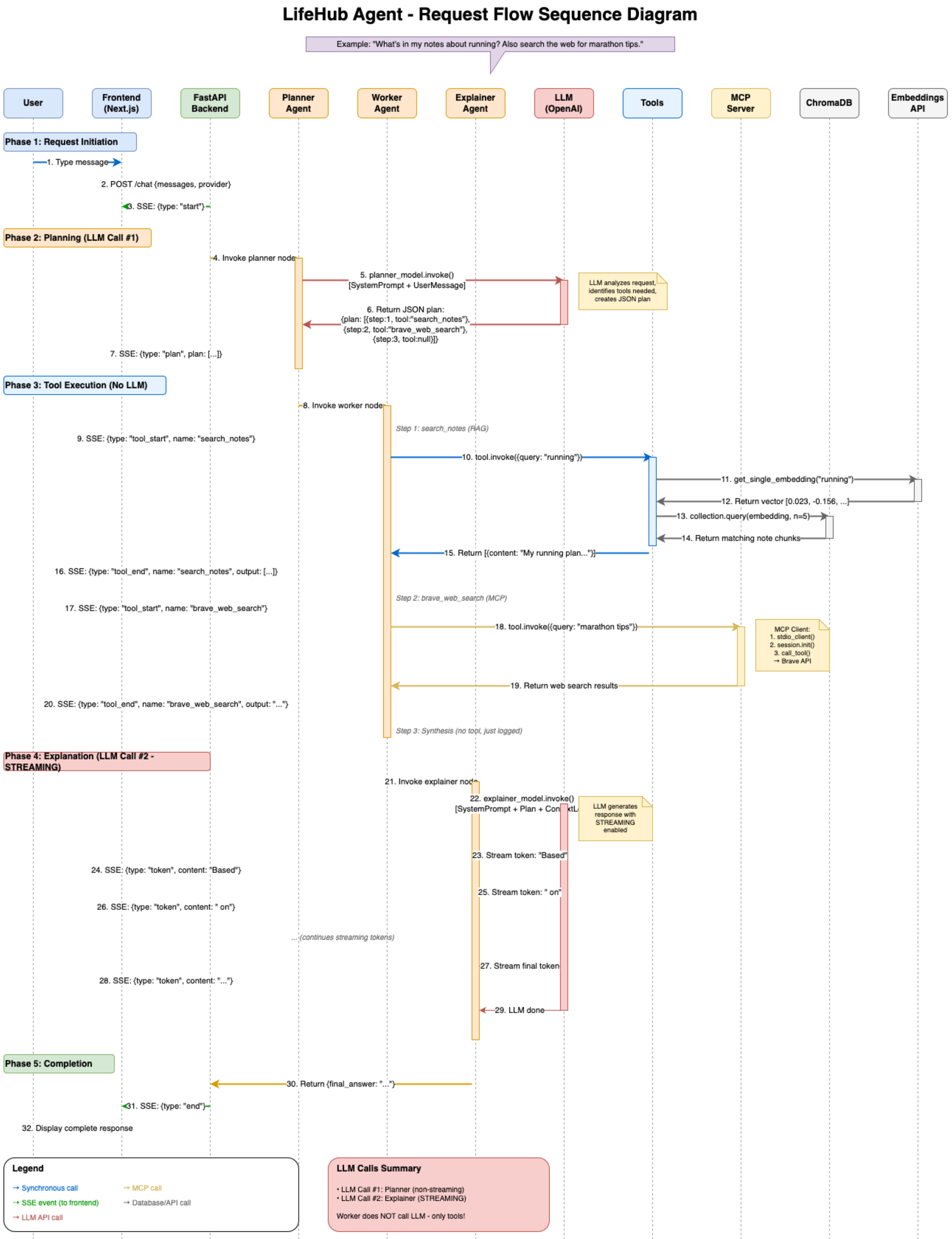
• Monday: 3 mile easy run
• Wednesday: 5 mile tempo run
• Friday: 3 mile recovery run
• Sunday: Long run (7-10 miles)

I've also added 'buy shoes' to your task list!"

😊

📊 Sequence Diagram

Detailed sequence diagram showing all back-and-forth calls between entities during an end-to-end request lifecycle, including when data is sent to the LLM.


LifeHub Agent - Request Flow Sequence Diagram
Example: "What's in my notes about running? Also search the web for marathon tips."
User
Frontend (Next.js)
FastAPI Backend
Planner Agent
Worker Agent
Explainer Agent
LLM (OpenAI)
Tools
MCP Server
ChromaDB
Embeddings API
Phase 1: Request Initiation
1. Type message
2. POST /chat {messages, provider}
3. SSE: {type: "start"}
Phase 2: Planning (LLM Call #1)
4. Invoke planner node
5. planner_model.invoke() [SystemPrompt + UserMessage]
LLM analyzes request, identifies tools needed, creates JSON plan
6. Return JSON plan: {plan: [{step:1, tool:"search_notes"}, {step:2, tool:"brave_web_search"}, {step:3, tool:null}]}
7. SSE: {type: "plan", plan: [...]}
Phase 3: Tool Execution (No LLM)
8. Invoke worker node
Step 1: search_notes (RAG)
9. SSE: {type: "tool_start", name: "search_notes"}
10. tool.invoke({query: "running"})
11. get_single_embedding("running")
12. Return vector [0.023, -0.156, ...]
13. collection.query(embedding, n=5)
14. Return matching note chunks
15. Return [{content: "My running plan..."}]
16. SSE: {type: "tool_end", name: "search_notes", output: [...]}
Step 2: brave_web_search (MCP)
17. SSE: {type: "tool_start", name: "brave_web_search"}
18. tool.invoke({query: "marathon tips"})
MCP Client: 1. stdio_client() 2. session.init() 3. call_tool() → Brave API
19. Return web search results
20. SSE: {type: "tool_end", name: "brave_web_search", output: "..."}
Step 3: Synthesis (no tool, just logged)
Phase 4: Explanation (LLM Call #2 - STREAMING)
21. Invoke explainer node
22. explainer_model.invoke() [SystemPrompt + Plan + Con ktL
LLM generates response with STREAMING enabled
23. Stream token: "Based"
24. SSE: {type: "token", content: "Based"}
25. Stream token: " on"
26. SSE: {type: "token", content: " on"}
... (continues streaming tokens)
27. Stream final token
28. SSE: {type: "token", content: "..."}
29. LLM done
Phase 5: Completion
30. Return {final_answer: "..."}
31. SSE: {type: "end"}
32. Display complete response
Legend
→ Synchronous call
→ MCP call
→ SSE event (to frontend)
→ Database/API call
→ LLM API call
LLM Calls Summary
• LLM Call #1: Planner (non-streaming)
• LLM Call #2: Explainer (STREAMING)
Worker does NOT call LLM - only tools!

Key insight: LLM is called exactly **2 times** per request:

1. **Planner** (non-streaming) → Creates JSON execution plan
2. **Explainer** (streaming) → Generates final user response

The Worker agent does **not** call the LLM - it directly invokes tools and collects results.

🧠 AI Concepts

This project demonstrates several key AI/LLM concepts:

Concept	**Description**
LLMs	OpenAI GPT-4o-mini or Ollama (Llama 3.2) for local inference
Multi-Agent	Planner → Worker → Explainer pipeline
RAG	Vector search over personal notes using ChromaDB
Tool Calling	Weather, tasks, notes search, web search (MCP)
Streaming	Real-time token delivery via SSE
MCP	Model Context Protocol for external tool integration

> 📖 **For detailed explanations with diagrams and code examples, see AI Concepts Guide**

📁 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
│   │   ├── embeddings.py        # Embedding provider (OpenAI/Ollama)
│   │   └── 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

> 📖 **For detailed installation instructions (macOS & Windows), see the Setup Guide**

Prerequisites

- Python 3.11+
- Node.js 18+
- uv package manager
- OpenAI API key (or Ollama for fully local setup)

Backend (Terminal 1)

```
cd lifehub-agent
uv sync                                            # Install dependencies
export OPENAI_API_KEY="your-key"                   # Set API key
(macOS/Linux)
uv run python -m backend.rag.ingest_notes          # Ingest notes into
vector store
uv run uvicorn backend.app.main:app --reload --port 8000  # Start server
```

Frontend (Terminal 2)

```
cd lifehub-agent/frontend
npm install
npm run dev
```

Open the App

Go to **http://localhost:3000**

Fully Local with Ollama (No API Key)

```
ollama pull llama3.2 && ollama pull nomic-embed-text  # Pull models
EMBEDDING_PROVIDER=ollama uv run python -m backend.rag.ingest_notes
uv run uvicorn backend.app.main:app --reload --port 8000
# Select "Ollama" in the UI dropdown
```

🔌 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 (Llama 3.2)
Vector Store	ChromaDB (embedded, file-based)
Embeddings	OpenAI text-embedding-3-small or Ollama nomic-embed-text
Weather API	Open-Meteo (free, no API key)
MCP Integration	Model Context Protocol for external tools
Backend	FastAPI, Python 3.11, httpx, mcp
Frontend	Next.js, React, TypeScript, Tailwind CSS
Deployment	Render (backend), Vercel (frontend)

⚙️ Environment Variables

Variable	Required	Default	Description
`OPENAI_API_KEY`	Yes*	-	OpenAI API key (*not needed if using Ollama for everything)
`EMBEDDING_PROVIDER`	No	`openai`	Embedding provider: `openai` or `ollama`
`OLLAMA_BASE_URL`	No	`http://localhost:11434`	Ollama server URL
`BRAVE_API_KEY`	No	-	Brave Search API key (enables web search via MCP)
`NEXT_PUBLIC_BACKEND_URL`	No	`http://localhost:8000`	Backend URL for frontend

🔌 MCP (Model Context Protocol)

LifeHub supports MCP for connecting to external tool servers.

Brave Search Integration

Enable web search by setting the `BRAVE_API_KEY` environment variable:

1. Get a free API key at Brave Search API (2,000 queries/month free)
2. Set the environment variable:

```
export BRAVE_API_KEY="your-api-key"
```

3. Restart the server - MCP tools will be loaded automatically

Available tools when enabled:

- `brave_web_search` - Search the web
- `brave_local_search` - Search for local businesses
- `brave_news_search` - Search news articles
- `brave_image_search` - Search images
- `brave_video_search` - Search videos

Example query: "Search the web for the latest Python 3.13 features"

📚 Key Files Reference

File	Purpose
`backend/rag/embeddings.py`	Embedding provider abstraction
`backend/agents/graph.py`	Multi-agent orchestration logic
`backend/tools/notes.py`	RAG search implementation
`backend/rag/store.py`	ChromaDB setup
`backend/mcp/client.py`	MCP client for external tools
`backend/mcp/config.py`	MCP server configuration
`backend/app/main.py`	API endpoints + streaming
`frontend/src/app/page.tsx`	Chat UI component

📄 License

MIT