RAG LangGraph AI Agent :

AI Agent Application: Architecture and Core Modules Report

1. Introduction

This report outlines the architecture and core modules of the AI Agent application. The application is designed to provide intelligent, context-aware responses by integrating a custom knowledge base with real-time web search capabilities. The architecture emphasizes modularity, scalability, and transparency, making it a robust foundation for advanced AI solutions.

2. High-Level Application Architecture

The AI Agent application is structured into several interconnected layers, facilitating clear separation of concerns and efficient data flow.

2.1. Conceptual Layers

1. **User Interface (UI) Layer:**
    - **Purpose:** Serves as the primary point of interaction for the end-user. It provides a conversational interface for querying the AI agent and functionalities for managing the application's knowledge base.
    - **Components:** Primarily implemented using Streamlit, offering a web-based chat interface, a web search toggle, and a document upload mechanism.
    - **Flow:** Initiates user queries and document upload requests, and displays the AI agent's responses along with a detailed workflow trace.
2. **API Layer:**
    - **Purpose:** Acts as the communication gateway, mediating interactions between the frontend UI and the backend's core AI logic. It exposes well-defined RESTful endpoints.
    - **Components:** Built using FastAPI, providing high-performance asynchronous API endpoints.

- **Flow:** Receives HTTP requests (e.g., chat messages, file uploads) from the UI, processes them, and dispatches them to the appropriate internal services or the AI agent's core. It also formats and sends responses back to the UI.

3. **Core Logic / Agent Layer:**
    - **Purpose:** Represents the "brain" of the application, orchestrating complex decision-making, information retrieval, and response generation. This layer defines the AI agent's behavior and workflow.
    - **Components:** Implemented using LangGraph, which enables the creation of stateful, cyclic graphs of interconnected nodes. Each node performs a specific task (e.g., routing, RAG lookup, web search, answer generation).
    - **Flow:** Manages the sequential and conditional execution of the AI agent's workflow. It dynamically routes queries based on intent and available resources, leveraging external tools and LLMs as needed.
4. **Data / Knowledge Base Layer:**
    - **Purpose:** Stores and facilitates the retrieval of custom, domain-specific information, forming the basis for the Retrieval-Augmented Generation (RAG) component.
    - **Components:** Utilizes Pinecone as a vector database, HuggingFace Embeddings for converting text into searchable vector representations, and PyPDFLoader for extracting text from PDF documents.
    - **Flow:** Processes text from uploaded documents by generating embeddings and storing them in Pinecone. The RAG Lookup node within the Agent Layer queries this database to retrieve relevant contextual information.
5. **External Services Layer:**
    - **Purpose:** Provides specialized capabilities that are external to the core application, such as large language model inference and real-time internet search.
    - **Components:** Integrates with Groq for high-performance LLM inference (used for routing, judging sufficiency, and generating answers) and Tavily Search API for real-time web search.

- **Flow:** The Agent Layer's nodes interact with Groq LLMs for intelligent decision-making and content generation. The Web Search node specifically leverages the Tavily Search API to fetch up-to-date information from the internet.

2.2. Overall Application Flow

The application's workflow can be summarized as follows:

1. A user interacts with the **Streamlit UI** by submitting a chat query or uploading a PDF document.
2. The request is sent to the **FastAPI Backend** (API Layer).
3. **For chat queries:** The FastAPI Backend dispatches the request to the **LangGraph Agent Core**. The Agent Core then dynamically routes the query, interacting with the **Pinecone Vector Database** (for RAG) or the **Tavily Search API** (for web search), and leveraging **Groq LLMs** for various decision-making and generation tasks. The agent's final answer and a detailed workflow trace are then returned through the FastAPI Backend to the Streamlit UI.
4. **For document uploads:** The FastAPI Backend utilizes **PyPDFLoader** to extract text from the PDF. This text is then processed by **HuggingFace Embeddings** to generate vector representations, which are subsequently stored in the **Pinecone Vector Database**.

3. Core Modules and Their Responsibilities

The application's codebase is organized into distinct modules, promoting modularity, reusability, and ease of maintenance.

3.1. `frontend/` Directory

- `app.py`:
  - **Role:** The primary entry point for the Streamlit application. It orchestrates the display of all UI sections and manages the overall application flow on the client side.
  - **Key Responsibilities:** Initializes Streamlit's session state, calls functions from `ui_components.py` to render the interface, and invokes functions from `backend_api.py` to communicate with the FastAPI backend.
- `ui_components.py`:

- **Role:** Contains reusable functions for rendering specific user interface elements and sections within the Streamlit application.
- **Key Responsibilities:** Displays the chat history, renders the document upload form, presents the agent settings (including the "Enable Web Search" toggle), and visualizes the detailed agent workflow trace.

- `backend_api.py`:
  - **Role:** Handles all HTTP communication between the Streamlit frontend and the FastAPI backend. It abstracts away the complexities of `requests` calls.
  - **Key Responsibilities:** Provides functions for sending PDF documents to the `/upload-document/` endpoint and chat queries to the `/chat/` endpoint, including proper error handling for network issues or API errors.
- `session_manager.py`:
  - **Role:** Dedicated to initializing and managing variables within Streamlit's `st.session_state`.
  - **Key Responsibilities:** Ensures that critical application data, such as chat message history, unique session IDs, and user preferences (e.g., `web_search_enabled`), persist across Streamlit application reruns.
- `config.py`:
  - **Role:** Stores configurations specific to the frontend application.
  - **Key Responsibilities:** Loads environment variables (e.g., `FASTAPI_BASE_URL`) from the `.env` file, making them accessible throughout the frontend modules.

3.2. `backend/` Directory

- `main.py`:
  - **Role:** The main entry point for the FastAPI backend application. It defines and manages the API endpoints that serve the frontend.
  - **Key Responsibilities:** Initializes the FastAPI app, defines the `/upload-document/` and `/chat/` API routes, handles file uploads (including temporary storage and cleanup), orchestrates the execution of the LangGraph agent for chat requests, and collects the detailed workflow trace for the frontend.

- `agent.py`:
  - **Role:** The core module defining the AI agent's intelligence and workflow using LangGraph.
  - **Key Responsibilities:** Defines the `AgentState` (the shared memory for the agent's graph), implements the individual agent nodes (`router_node`, `rag_node`, `web_node`, `answer_node`), sets up their conditional transitions, and integrates LLMs (Groq) and external tools (Tavily Search). It also contains the crucial logic for managing the `web_search_enabled` flag.
- `vectorstore.py`:
  - **Role:** Manages all interactions with the Pinecone vector database, which serves as the application's RAG knowledge base.
  - **Key Responsibilities:** Initializes the Pinecone index, provides a retriever instance for searching documents, and implements the `add_document_to_vectorstore` function for processing and upserting new text data into Pinecone.
- `config.py`:
  - **Role:** Stores environment variables and API keys specific to the backend services.
  - **Key Responsibilities:** Provides access to sensitive credentials such as `GROQ_API_KEY`, `PINECONE_API_KEY`, and `TAVILY_API_KEY`, ensuring they are loaded securely from the `.env` file.

3.3. Utility Files (Project Root)

- `requirements.txt`:
  - **Role:** Lists all Python package dependencies required for the entire project.
  - **Key Responsibilities:** Ensures that all necessary libraries are installed in the deployment environment, maintaining consistency across development and production.
- `.env`:
  - **Role:** A file used for securely storing environment-specific variables and sensitive API keys.

- **Key Responsibilities:** Prevents hardcoding credentials directly into the source code, enhancing security and facilitating easy configuration changes across different environments.

4. Technology Stack and Important Frameworks

The AI Agent application is built upon a modern and robust technology stack, leveraging key frameworks and libraries for efficient development and deployment.

- **Programming Language:**
  - **Python:** The primary language used for both the frontend (Streamlit) and backend (FastAPI, LangGraph, LLM integrations). Python's rich ecosystem and strong AI/ML libraries make it ideal for this application.
- **Backend Frameworks & Libraries:**
  - **FastAPI:** A modern, fast (high-performance) web framework for building APIs with Python 3.7+ based on standard Python type hints. It's used for the backend API endpoints.
  - **LangChain:** A framework for developing applications powered by language models. It provides tools, chains, and agents, facilitating integration with various LLMs, vector stores, and other data sources.
  - **LangGraph:** An extension of LangChain, specifically designed for building stateful, multi-actor applications with LLMs. It enables the creation of complex, cyclic workflows (graphs) for AI agents, managing state transitions between nodes.
  - **Pydantic:** A data validation and settings management library using Python type hints. It's used extensively in FastAPI for defining API request/response schemas and ensuring data integrity.
  - `python-dotenv`: A library for loading environment variables from a `.env` file, crucial for managing API keys and configurations securely.
  - `python-multipart`: A library for parsing multipart form data, used by FastAPI to handle file uploads.
- **Frontend Framework:**
  - **Streamlit:** An open-source app framework for Machine Learning and Data Science teams. It allows for rapid development of interactive web applications with pure Python, used here for the user interface.

- **Large Language Model (LLM) Providers:**
  - **Groq:** A high-performance inference engine for LLMs. It provides fast and efficient access to models like Llama 3, used in this application for routing decisions, RAG sufficiency judgment, and final answer generation.
- **Vector Database:**
  - **Pinecone:** A leading vector database designed for high-performance similarity search. It serves as the core of the RAG system, storing and retrieving vector embeddings of documents.
- **Embedding Models:**
  - **HuggingFace Embeddings (via `sentence-transformers/all-MiniLM-L6-v2`):** Used to convert text content from documents and user queries into numerical vector embeddings, enabling semantic search within the Pinecone database.
- **Document Processing:**
  - **PyPDFLoader (via `langchain-community` and `pypdf`):** A LangChain document loader specifically for extracting text content from PDF files.
- **External Tools/APIs:**
  - **Tavily Search API:** A search API optimized for AI agents, providing real-time, relevant web snippets for queries requiring up-to-date internet information.

This comprehensive stack provides the necessary tools for building, deploying, and scaling a sophisticated AI agent application.