Job Recommendation System with RAG (LangChain + Groq/Gemma)

1. Problem Explanation

Finding relevant job opportunities efficiently is a challenge for many professionals. Job seekers often spend significant time filtering through irrelevant postings, manually searching on multiple platforms, and aligning opportunities with their specific skills, experience, and preferences.

Our objective was to **build an intelligent job recommendation system** that:

- Retrieves the most recent and relevant job postings from the web.
- Matches them to a candidate's skills, experience, and preferences.
- Allows input either **manually** or by **uploading a resume** for automated profile extraction.
- Provides ranked and contextually relevant recommendations.

The system integrates **Retrieval-Augmented Generation (RAG)** to ensure the recommendations are both **accurate** and **up-to-date**.

2. Use of RAG with LangChain + Gemini/Gemma

RAG Workflow

1. **Retriever**: Uses Tavily API (and optionally Groq's browser search (openai model)) to fetch the latest job postings from multiple online sources.
2. **Retriever (LLM)**: Uses Groq's OpenAI model - `"openai/gpt-oss-20b"` which uses inbuilt tool of browser search for efficient web search.
3. **Vector Store**: Stores job postings in **FAISS** for semantic search and caching.

4. **Generator (LLM)**: Uses **Groq's Gemma model** to re-rank, summarize, and generate recommendations tailored to the candidate's profile.

5. **UI**: Built with **Streamlit** to allow interactive inputs and display results.

The RAG pipeline ensures:

- **Freshness** of information (latest jobs).
- **Contextual relevance** via embeddings and semantic search.
- **Natural language explanations** for each recommendation.

3. Folder / Code Walkthrough

```
project_root/
│
├── app/
│   ├── retriever.py       # Retrieves job postings (Tavily + Groq)
│   ├── recommender.py     # Ranks and summarizes recommendations
│   ├── resume_parser.py   # Extracts skills, experience, location from PDF resumes
│   ├── __init__.py
│
├── ui/
│   ├── app.py             # Streamlit interface for user interaction
│
├── cache/                 # Stores cached results
│
├── faiss_index/           # Stores persistent FAISS vector index
│
├── main.py                # CLI entry point (optional)
├── requirements.txt
├── README.md
└── .env                   # API keys (Groq, Tavily)
```

4. Challenges Faced & Solutions

Challenge 1: Inconsistent results from Groq browser tool

- **Issue**: Sometimes Groq API failed or returned incomplete data.
- **Solution**: Added **Tavily API fallback** to ensure we always retrieve relevant job postings.

Challenge 2: Duplicate job listings

- **Issue**: Multiple sources returned the same postings.
- **Solution**: Added **URL-based deduplication** before storing in FAISS.

Challenge 3: Resume parsing accuracy

- **Issue**: Regex-based parsing was brittle.
- **Solution**: Switched to **Groq LLM extraction** for robust and context-aware parsing.

Challenge 4: Latency in recommendations

- **Issue**: Long processing times when fetching & re-ranking jobs.
- **Solution**: Implemented **result caching** using both JSON cache and FAISS index.

5. Summary of What I Learned

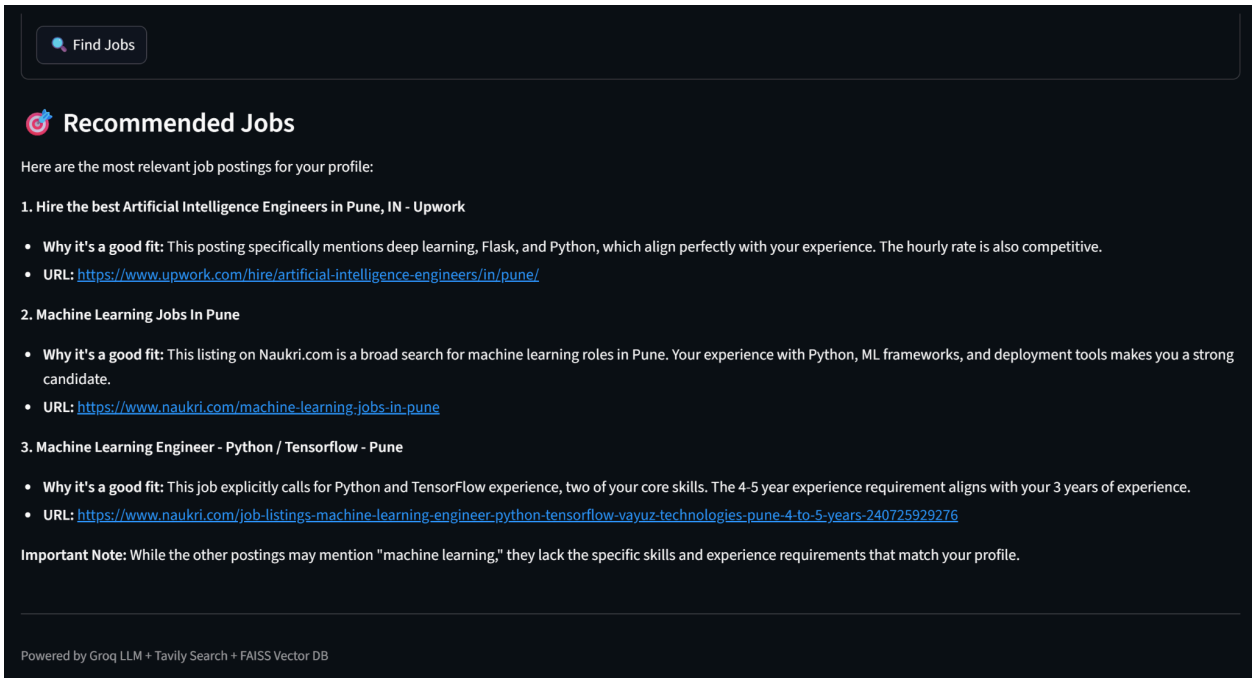
- **RAG Architecture**: How to combine retrieval (FAISS + API search) and generation (LLM re-ranking) for real-world applications.
- **LangChain Integration**: Efficiently chaining retrievers, vector stores, and LLMs.
- **Fallback Strategies**: Ensuring reliability by combining multiple data sources.

- **Resume Parsing with LLMs**: Using language models to extract structured data from unstructured documents.
- **Streamlit UI Development**: Building an intuitive, interactive frontend for non-technical users.

6. Additional Notes & Improvements for Future

- Add **job filtering** by salary range, company size, and remote/on-site preference.
- Integrate **real-time job alerts** via email/Slack.
- Use **FAISS search directly** for returning cached relevant results before hitting APIs.
- Implement **multi-language support** for parsing and recommendations.

7. Sample Input / Output



8. Project Setup & Execution

 # 1. Clone the repository
git clone https://github.com/ish-war/AI-Job-Recommendation.git
cd AI-Job-Recommendation

 # 2. Create virtual environment
uv init
uv venv
source venv/bin/activate # Linux/Mac
venv\Scripts\activate # Windows

 # 3. Install dependencies
uv add -r requirements.txt

 # 4. Add environment variables at .env
GROQ_API_KEY = ”your_groq_key"
TAVILY_API_KEY = “your_tavily_key"

 # 5. Run Streamlit app
streamlit run ui/app.py