Technical Design & Rationale: Semantic Job Matcher

1. Dataset Selection & Preprocessing

Chosen Dataset: Data Science Job Posting on Glassdoor (Kaggle)

For this project, I chose the Glassdoor Job Posting dataset because it provides a rich variety of unstructured, complex technical text. Job descriptions are an ideal testbed for semantic search as they contain overlapping terminology that requires more than simple keyword matching to differentiate.

Data Optimization Steps:

To ensure the demonstration is high-performing and focused, the following modifications were made:

- **Platform Migration:** The raw data was managed via Google Sheets to facilitate rapid structural cleaning.
- **Column Reduction:** The dataset was reduced to two core columns: 'Job Title' and 'Job Description'. This removes secondary metadata and focuses the vector engine on the most semantically significant text.
- **Row Truncation:** I capped the dataset at the first 50 rows. This provides a diverse sample, spanning healthcare, national security, and finance, while ensuring the local vector database remains lightweight and fast.

2. Expected User Questions

The system is optimized to handle natural language queries that describe an "ideal candidate" or specific project needs, such as:

- **Requirement Matching:** "Who is the ideal candidate for a role requiring 5+ years of T-SQL and Predictive Modeling?"
- **Gap Analysis:** "What specific certifications are required for the Clinical Lab roles?"
- **Cross-Functional Search:** "Find me roles that combine Healthcare knowledge with Machine Learning."

3. Vector Database Selection: Why ChromaDB?

I selected ChromaDB as the retrieval engine for three strategic reasons:

- **Local Persistence & Privacy:** Recruitment data and job requirements are often sensitive. ChromaDB runs entirely locally, ensuring no data is transmitted to a third-party cloud during the search process.

- **Explainable AI (XAI):** Chroma allows for the easy extraction of "Relevance Scores." I leveraged this to show users exactly how well a job matches their query (e.g., "Match: 88%"), moving away from "black box" search results.
- **Speed:** It provides sub-100ms retrieval times, ensuring a seamless user experience without the need for complex server infrastructure.

4. Search Logic & Quality Control

To maintain high-level standards for data quality, I implemented a Similarity Threshold:

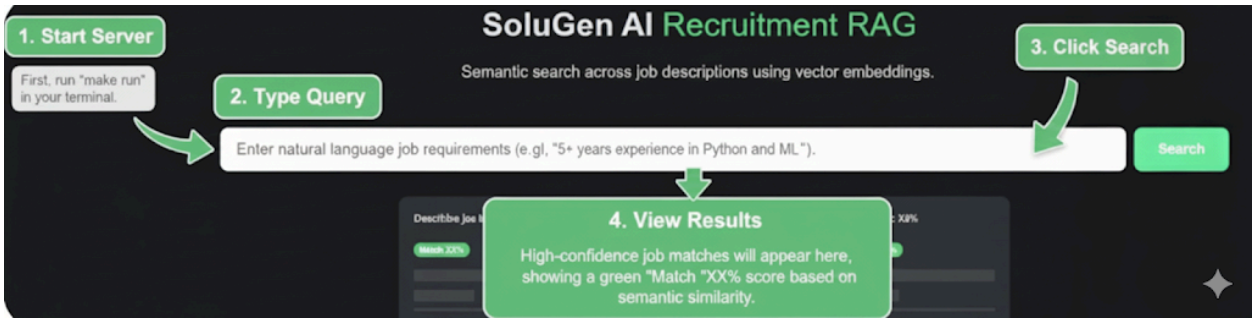
- **The Problem:** Most search engines always return the "Top K" results, even if they are poor matches.
- **The Solution:** This system calculates a mathematical cosine similarity score for every match. If a result falls below a 0.3 threshold, it is rejected.
- **The Result:** Users only see high-confidence, relevant matches, which significantly reduces "search noise" and improves trust in the system.

5. Developer Experience (DX)

The project is built for easy evaluation using a standardized environment:

- **Environment:** Managed via conda using the provided environment.yml file.
- **Automation:** A Makefile is included to simplify the workflow:
    - **make ingest**: Processes the 50-row CSV and builds the local vector store.
    - **make run**: Launches the FastAPI backend and the custom Green-themed UI.
    - **make test**: Runs the integration test suite.

6. User Interface & Interaction Guide



- Demo video