# README - CV SCREENING AGENT PROJECT FOR

# LEADTECH

**By Eduardo Bobenrieth**

[Workflow fo CV Screening Agent 1](#_Toc968274408)

[1. CV Preparation (Input Data) 1](#_Toc156898290)

[2. Data Processing (RAG Build Pipeline) 1](#_Toc2027125161)

[3. Backend (Retrieval & Answer Generation) 2](#_Toc64521272)

[4. Frontend (User Interface) 2](#_Toc2077260416)

[5. Running the App 3](#_Toc874162433)

[6. Output 3](#_Toc1112382304)

[7. Project structure 4](#_Toc1964236193)

[Story of Building the CV Screening Agent 4](#_Toc1799074067)

[1. Goal & Initial Plan 4](#_Toc468094309)

[2. First Approach: OpenAI Embeddings + FAISS + Flask 5](#_Toc1028452402)

[3. Switch to OpenRouter 5](#_Toc103645711)

[4. Switch to Local Embeddings (SentenceTransformers) 6](#_Toc364227060)

[5. Final Stack: Cohere + OpenRouter 6](#_Toc1235803470)

[6. Frontend Evolution 6](#_Toc1834570153)

## Workflow fo CV Screening Agent

### 1. CV Preparation (Input Data)

User places **25–30 fake CVs** (PDF format) into the folder data/cvs/.

Each CV should have realistic content (name, photo, skills, education, experience).

### 2. Data Processing (RAG Build Pipeline)

**User runs rag\_build.py** in his/her terminal

What happens:

- Reads all PDFs from data/cvs/

- Extracts and chunks the text (to keep context manageable for embeddings)

- Generates vector embeddings for each chunk using Cohere’s embed-english-v3.0 model.

- Saves:

* **FAISS index** (faiss\_index), contains vector representations of CV chunks
* **Metadata** (faiss\_metadata.pkl), maps chunks back to filenames and full text

### 3. Backend (Retrieval & Answer Generation)

**cv\_chat\_web.py** is the backend server (Flask)

- Loads FAISS index and metadata into memory at startup.

- Ready to accept a question from the frontend (user input).

- Encodes the question using Cohere’s embedding model.

- Runs a similarity search in FAISS to retrieve the most relevant CV chunks

- Sends the retrieved context + user question to GPT-4o-mini via OpenRouter.

- Returns a grounded answer based on CV content

### 4. Frontend (User Interface)

Implemented as a **Flask HTML template** (templates/index.html):

- Simple text box + “Ask” button for sending questions.

- Displays AI-generated answer about candidate’s and their CV contents.

- Includes a favicon (golden star icon) to make it look more polished.

### Running the App

- Place CVs in the data/cvs/ folder

- Run this command in your terminal to (re)build the FAISS index: python rag\_build\_api.py

- Start the web app by running this command in your terminal: python cv\_chat\_web.py

- Open browser at <http://127.0.0.1:5000>

**Examples of questions and prompts for the CV Screening Agent:**

- Who has experience with Python?

- From the candidates that know python, who's the best choice overall?

- Summarize the profile of Sophia Taylor.

- Chose the best candidate for this job; Job Title: Full-Stack Developer Description: we’re looking for a talented Full-Stack Developer to join our growing tech team. The ideal candidate has strong experience building web applications from end to end, including both front-end and back-end development. Responsibilities: develop and maintain scalable web applications. Collaborate with designers and product managers to deliver high-quality features. Write clean, testable code and perform unit testing. Work with REST APIs and databases (SQL/NoSQL). Required Skills: proficiency in JavaScript/TypeScript, React (or similar framework), and Node.js. Experience with SQL databases and writing optimized queries. Familiarity with unit testing frameworks (Jest, Mocha, etc.). Understanding of CI/CD pipelines and Git workflows. Nice to Have: knowledge of Docker and cloud platforms (AWS, GCP, Azure). Experience with authentication/authorization systems.

### Output

System returns **specific, grounded answers** citing relevant candidates.

When asked questions such as “Who is the most...” it returns an answer that is a ranked list of candidates, from most to least relevant.

Each question is treated independently (no chat history).

### Project structure

ai\_prototype\_clean/

│

├── venv/ # Python virtual environment (don't commit to GitHub)

│

├── data/

│ └── cvs/ # Store all CV PDFs here

│ ├── cv\_0.pdf

│ ├── cv\_1.pdf

│ └── ... (up to cv\_24.pdf)

│

├── static/ # Static assets (CSS, images, favicon)

│ └── star.ico # Golden star icon used as favicon

│

├── templates/ # HTML templates for Flask

│ └── index.html # Chat interface page

│

├── faiss\_index # FAISS index file generated by rag\_build.py

├── faiss\_metadata.pkl # Metadata mapping filenames to extracted text

│

├── rag\_build.py # Script to extract text & build FAISS index

├── build\_metadata.py # Script to build metadata only (optional)

├── cv\_chat\_web.py # Flask web app (with OpenRouter API integration)

│

├── .env # Environment variables (e.g., OPENROUTER\_API\_KEY)

├── requirements.txt # Python dependencies (install with pip)

└── README.md # (Optional) General purpuso of project, setup & usage instructions for team

## Story of Building the CV Screening Agent

### 1. Goal & Initial Plan

The goal was to create a **chat application** that lets a recruiter ask natural-language questions about a set of CVs and get grounded answers, not generic AI fluff.  
This required three main components:

**- Data ingestion**: Extract text from PDF CVs and store it in a searchable format.

**- RAG pipeline**: Use embeddings + vector search to retrieve relevant CV snippets.

**- Chat interface**: Build a UI so recruiters can type questions and get answers.

### 2. First Approach: OpenAI Embeddings + FAISS + Flask

**- Embeddings**: text-embedding-3-small from OpenAI.

**- Vector store**: FAISS, stored locally (faiss\_index).

**- Language model**: GPT-4 via OpenAI API.

**- UI**: Minimal Flask app with a simple HTML form.

This worked in principle, but…

I hit **401 API key errors** several times because of OpenAI’s API key format mismatch with the latest Python SDK.

The new OpenAI client (openai>=1.0.0) required a different call syntax (client.embeddings.create() instead of the old openai.Embedding.create()), which broke the first version.

Billing concerns: OpenAI embeddings can get expensive with 25–30 CVs + frequent user queries.

### 3. Switch to OpenRouter

I then switched to **OpenRouter** because it provides access to multiple LLMs (including GPT-4 and Claude) under a single API key.

**Pros**: No need to manage multiple providers separately, often cheaper/free tiers available.

**Cons**: Still needed embeddings from somewhere.

At this point, the stack was:

**- OpenRouter** for chat completions.

**- OpenAI embeddings** (via OpenRouter, initially).

But embedding still returned errors (str object had no attribute data), so the pipeline was unreliable.

### 4. Switch to Local Embeddings (SentenceTransformers)

To remove dependency on external embedding APIs I used sentence-transformers/all-MiniLM-L6-v2, a free, local model for generating embeddings. This made embedding **fast and free** but required PyTorch installed locally.The FAISS index was rebuilt with these embeddings.

This solved the API key issues, but the answers felt too shallow because the context was not rich enough for the LLM.

### 5. Final Stack: Cohere + OpenRouter

To improve quality, I moved to **Cohere embeddings (**embed-english-v3.0**)**:

- Higher-quality semantic search than MiniLM.

- Easy to use (co.embed(...)) and worked reliably with FAISS.

And I kept **OpenRouter** for generating answers:

- Model: openai/gpt-4o-mini; fast, good reasoning, affordable.

- Prompt improved to instruct the model to **rank candidates** and explain reasoning.

This version produced **much better, grounded answers, that was a keeper!**

### 6. Frontend Evolution

Started as a CLI (cv\_chat.py) just for testing retrieval. Upgraded to a Flask app with index.html. Styled it with dark blue / sky blue theme + a star favicon for a more polished professional look.