TalentGrid

AI System Architecture & Documentation

A comprehensive guide to the RAG-powered talent sourcing platform

Document Version	1.0
Last Updated	February 2026
Embedding Model	all-mpnet-base-v2 (768 dimensions)
Vector Database	ChromaDB
Backend Framework	FastAPI + PostgreSQL
Frontend Framework	React + Vite

Table of Contents

1. Executive Summary

TalentGrid is a **RAG-powered (Retrieval-Augmented Generation)** talent sourcing platform that combines semantic search with hybrid retrieval to help recruiters find the best candidates.

System Components

- **Backend:** FastAPI with PostgreSQL (candidate data) + ChromaDB (vector embeddings)
- **Frontend:** React with Vite, TailwindCSS, and Recharts
- **AI Pipeline:** CV parsing (Mistral OCR + Groq/Gemini), embedding (SentenceTransformer), hybrid search (semantic + BM25), and re-ranking (Cohere)

Key Features

- Multi-format CV upload (PDF, DOCX, images) with AI-powered parsing
- Semantic search using 768-dimensional embeddings
- Hybrid search combining vector similarity and keyword matching
- Synonym expansion for tech terms (react ↔ reactjs, k8s ↔ kubernetes)
- Title boosting for job-specific queries
- Optional cross-encoder re-ranking with Cohere
- Real-time filtering by experience, languages, and location

2. CV Import Flow

The CV import process transforms raw documents into searchable, structured data through a multi-stage pipeline.

2.1 Flow Overview

Step	Action	Description
1	Upload	User uploads CV via drag-and-drop (PDF, DOCX, or image)
2	Save	File saved to disk with unique identifier
3	OCR	Mistral OCR extracts text from PDF/images
4	Parse	Groq (Llama 3.3 70B) or Gemini extracts structured data
5	Store DB	Candidate record created in PostgreSQL
6	Index	CV chunked, embedded, and stored in ChromaDB
7	Response	Parsed data returned to frontend

2.2 OCR Stage (Mistral)

PDFs are processed using **Mistral OCR (mistral-ocr-latest)**. The PDF is converted to base64, sent to Mistral's API, and returns structured markdown content for each page.

File: backend/app/ai/ingestion/parser.py

2.3 LLM Parsing Stage

The OCR text is sent to an LLM for structured extraction. The system uses a **primary-fallback strategy**:

Priority	Provider	Model	Rate Limit	Use Case
Primary	Groq	Llama 3.3 70B	14,400 req/day	Fast, free tier
Fallback	Google	Gemini 2.0 Flash	High limits	If Groq fails

2.4 Parsed Data Schema

The LLM extracts the following structured fields from the CV:

Field	Type	Description
name	`string`	Candidate full name (required)
email	`string`	Contact email
phone	`string`	Phone number
title	`string`	Current job title
location	`string`	City/Country
years_experience	`integer`	Total years of experience
summary	`text`	Professional summary/objective
skills	`array[string]`	List of technical skills
languages	`array[object]`	{name, level} for each language
education	`array[object]`	{degree, field, institution, from, to}
experience	`array[object]`	{role, organization, from, to, description}
certifications	`array[string]`	Professional certifications
projects	`array[object]`	Notable projects or work samples

3. Database Storage (PostgreSQL)

All candidate data is stored in PostgreSQL for reliable, queryable storage. The database serves as the source of truth for candidate information.

3.1 Candidate Model

File: backend/app/models/candidate.py

Column	Type	Purpose
`id`	`Integer (PK)`	Unique identifier
`name`	`String(255)`	Candidate name (indexed)
`email`	`String(255)`	Email (indexed)
`title`	`String(255)`	Job title (indexed)
`skills`	`ARRAY(String)`	PostgreSQL array of skills
`languages`	`JSONB`	[{name, level}] array
`education`	`JSONB`	Education history array
`experience`	`JSONB`	Work experience array
`parsed_data`	`JSONB`	Complete parsed CV for reference
`quality_score`	`Numeric(3,2)`	AI-assigned quality 0-10
`source`	`String(100)`	"upload", "import", etc.
`created_at`	`DateTime`	Record creation timestamp

3.2 CVFile Model

Tracks uploaded files and links them to candidates:

Column	Type	Purpose
`id`	Integer (PK)	Unique identifier
`candidate_id`	Integer (FK)	Links to Candidate.id
`filename`	String(255)	Original filename
`file_path`	String(500)	Storage path on disk
`file_type`	String(50)	"pdf", "docx", etc.
`upload_status`	String(50)	"pending", "processed", "failed"

4. Vector Store (ChromaDB)

ChromaDB stores vector embeddings for semantic search. Each CV is split into chunks, and each chunk is embedded as a 768-dimensional vector.

4.1 Storage Configuration

Setting	Value
Storage Type	PersistentClient (survives restarts)
Storage Path	./chroma_db/
Collection Name	"cvs"
Vector Dimensions	768 (all-mpnet-base-v2)

4.2 Chunking Strategy

CVs are split into **semantic chunks** for better retrieval. Each chunk type captures a specific aspect of the candidate's profile:

Chunk Type	ID Pattern	Content
Profile	`{cv_id}_profile`	Job title + professional summary
Skills	`{cv_id}_skills`	Comma-separated list of all skills
Experience	`{cv_id}_experience_{i}`	Role, company, duration, description (per job)
Education	`{cv_id}_education_{i}`	Degree, field, institution (per degree)
Languages	`{cv_id}_languages`	All languages with proficiency levels
Certifications	`{cv_id}_certifications`	List of professional certifications
Projects	`{cv_id}_project_{i}`	Individual project descriptions

4.3 Chunk Metadata

Each chunk stored in ChromaDB includes metadata for filtering and lookup:

- `cv_id`: UUID of the CV (for grouping chunks)
- `chunk_type`: Type of chunk (profile, skills, etc.)
- `name`: Candidate name
- `location`: Location (for filtering)

- `experience_years`: Years of experience (for filtering)
- `title`: Job title
- `languages`: Comma-separated language names
- `candidate_id`: Database ID (for direct lookup - critical for search)

5. Embedding Model

Text is converted to vectors using **SentenceTransformers**, enabling semantic similarity search.

5.1 Model Configuration

Property	Value
Model Name	all-mpnet-base-v2
Source	HuggingFace (public, no auth required)
Vector Dimensions	768
Architecture	MPNet (multilingual BERT-based)
Language Support	50+ languages
Cache Location	~/.cache/ai_recruiter_models/

5.2 Why all-mpnet-base-v2?

- **Better semantic understanding:** 768 dimensions capture more nuance than 384-dim models
- **Deeper architecture:** More layers = better context understanding
- **Speed vs quality balance:** Fast enough for real-time search, accurate enough for production
- **Multilingual:** Works well with CVs in different languages
- **No API costs:** Runs locally, no per-request charges

5.3 Embedding Process

File: backend/app/ai/ingestion/embedder.py

Step	Description
1. Input	Text string (chunk content or search query)
2. Tokenization	Model's internal tokenizer processes text
3. Encoding	Neural network produces 768-dimensional vector
4. Normalization	Vector converted to Python list for storage
5. Output	List[float] with 768 values, ready for ChromaDB

6. Search System

The search system uses a **hybrid approach** combining semantic understanding (vector similarity) with keyword matching (BM25-like scoring) for optimal results.

6.1 Search Pipeline

Stage	Description
1. Query Parsing	Extract filters, clean query text
2. Embedding	Convert query to 768-dim vector
3. Vector Search	Find similar chunks in ChromaDB
4. Keyword Scoring	BM25-like scoring with tech skill boosts
5. Score Merging	Weighted combination (dynamic weights)
6. Grouping	Group chunks by CV, use MAX score
7. Re-ranking	Optional Cohere cross-encoder
8. Enrichment	Add candidate_id for DB lookup
9. Response	Return ranked candidates with scores

6.2 Dynamic Weights

Search weights are adjusted based on query type for optimal results:

Query Type	Example	Semantic	Keyword	Reasoning
Skill Query	"flutter", "react"	20%	80%	Exact skill match is critical
Job Description	"Senior developer with React experience..."	60%	40%	Semantic understanding needed

6.3 Synonym Expansion

Tech terms are automatically expanded to match variations:

Original Term	Also Matches
`react`	`reactjs, react.js`
`vue`	`vuejs, vue.js`
`node`	`nodejs, node.js`

javascript	js, ecmascript
typescript	ts
kubernetes	k8s
postgresql	postgres, psql
mongodb	mongo
machine learning	ml, machinelearning

6.4 Title Boosting

When queries contain job title keywords (developer, engineer, senior, etc.), candidates with matching titles receive a boost up to **1.5x**. For example, "flutter developer" will boost candidates who have "flutter" prominently in their profile and job title keywords like 'developer' or 'engineer' in their experience.

6.5 Tech Skill Boosting

The system maintains a list of 40+ recognized tech skills. When these appear in documents, they receive a **3x scoring multiplier**, ensuring skill-based queries surface the right candidates.

6.6 Re-ranking (Cohere)

If a Cohere API key is configured, results are re-ranked using a cross-encoder model. This provides a more sophisticated relevance assessment but is optional - the system gracefully falls back to hybrid search scores if unavailable.

7. Frontend Integration

7.1 Technology Stack

Component	Technology
Framework	React 18 with Vite
Styling	TailwindCSS
Icons	Lucide React
Charts	Recharts
HTTP Client	Axios
State Management	React Query + useState

7.2 Key Components

Component	File	Purpose
Search Page	`pages/Search/Search.jsx`	Main search interface with results
Filter Panel	`pages/Search/FilterPanel.jsx`	Experience, language, location filters
Import Page	`pages/Import/Import.jsx`	CV upload with progress tracking
Upload Zone	`pages/Import/UploadZone.jsx`	Drag-and-drop file upload
Candidate Card	`components/CandidateCard.jsx`	Individual search result display
API Service	`services/api.js`	Backend API integration

7.3 API Endpoints Used

Endpoint	Method	Purpose
`/api/search`	POST	Semantic search with filters
`/api/import/cv`	POST	Upload and parse CV file
`/api/candidates`	GET	List all candidates
`/api/candidates/{id}`	GET	Get single candidate details
`/api/admin/reindex`	POST	Start reindexing operation

`/api/admin/reindex/status`	GET	Check reindex progress
`/api/admin/vector-store/stats`	GET	Get ChromaDB statistics

8. Admin Features

8.1 Reindexing

Reindexing rebuilds the ChromaDB vector store from all candidates in the database. Use this after deployment, database restore, or when changing embedding models.

API Method:

POST /api/admin/reindex - Starts background reindexing
GET /api/admin/reindex/status - Check progress

CLI Method:

```
python backend/reindex_candidates.py
```

8.2 Vector Store Statistics

GET /api/admin/vector-store/stats returns information about ChromaDB collections:

8.3 When to Reindex

- After initial deployment to index existing candidates
- After restoring database from backup
- If search results seem incomplete or incorrect
- After changing the embedding model (dimension change requires full reindex)
- After adding synonym or title boosting features

9. Configuration Guide

9.1 Environment Variables

Variable	Required	Description
`DATABASE_URL`	Yes	PostgreSQL connection string
`SECRET_KEY`	Yes	JWT signing key
`MISTRAL_API_KEY`	Yes	Mistral OCR API key
`GROQ_API_KEY`	Recommended	Groq API for LLM parsing (free tier)
`GEMINI_API_KEY`	Fallback	Google Gemini API (fallback parser)
`COHERE_API_KEY`	Optional	Cohere re-ranking (enhances results)
`CHROMA_DB_PATH`	Optional	Vector store path (default: ./chroma_db)

9.2 Obtaining API Keys

- **Mistral:** console.mistral.ai - Required for OCR
- **Groq:** console.groq.com - Free tier with 14,400 requests/day
- **Gemini:** aistudio.google.com - Google AI Studio
- **Cohere:** dashboard.cohere.com - Optional for re-ranking

10. File Reference

10.1 Backend AI Files

File	Purpose
`app/ai/service.py`	Unified AI service interface
`app/ai/ingestion/parser.py`	CV parsing (OCR + LLM)
`app/ai/ingestion/chunker.py`	CV chunking strategy
`app/ai/ingestion/embedder.py`	Text to vector conversion
`app/ai/ingestion/pipeline.py`	Ingestion orchestration
`app/ai/storage/vector_store.py`	ChromaDB wrapper
`app/ai/retrieval/query_parser.py`	Filter extraction
`app/ai/retrieval/hybrid_search.py`	Semantic + BM25 search
`app/ai/retrieval/retriever.py`	Search orchestration
`app/ai/ranking/cross_encoder.py`	Cohere re-ranking

10.2 Backend Route Files

File	Purpose
`app/routes/import_cv.py`	CV upload endpoint
`app/routes/search.py`	Search endpoint
`app/routes/admin.py`	Admin operations (reindex, stats)
`app/routes/candidates.py`	Candidate CRUD

10.3 Frontend Files

File	Purpose
`src/pages/Search/Search.jsx`	Main search page
`src/pages/Search/FilterPanel.jsx`	Filter sidebar
`src/pages/Import/Import.jsx`	CV import page
`src/pages/Import/UploadZone.jsx`	Drag-drop upload
`src/services/api.js`	API calls to backend