Technical Report: CV-Job Matching System

Introduction

This report documents the implementation of a CV-job matching system, which has been named "JobFinder" for this submission, developed as part of the JoBins AI Engineer technical assignment. The system performs candidate-vacancy matching and recommends jobs to candidates based on their resume using natural language processing and semantic search. Built over a two-week development cycle, it processes PDF resumes through LLM-based extraction, converts them to standardized JSON format, and generates ranked job recommendations using multi-factor scoring. The implementation handles the complete pipeline from upload to explanation, combining vector embeddings for semantic similarity with rule-based matching for explicit requirements like experience and education. All core requirements i.e. up to Task 3 were completed along with several advanced features including two-stage retrieval with reranking, comprehensive performance metrics, and intelligent caching strategies.

System Architecture Overview

The system follows a layered architecture with FastAPI handling HTTP requests, service modules managing business logic, PostgreSQL and pgvector for database, Redis for caching, and celery with Rabbit MQ for background tasks.

Core Components:

API Layer (app/main.py, app/api/routes/*): Three main endpoint groups: recommendations (resume upload → job matches), jobs (ingestion and management), and metrics (performance monitoring).

Resume Processing (app/services/resume/): Extracts text from PDF with PyMuPDF, parses with an LLM, in this case Gemini, to structured JSON Resume format, and validates output against Pydantic schemas.

Job Processing (app/services/job/): Parses job descriptions with an LLM, generates 3072-dimensional embeddings via gemini-embedding-001, and handles batch ingestion through Celery tasks.

Recommendation Engine (app/services/recommendation/): Core matching logic operates in five stages:

1. Generates resume embedding (cached for 7 days).
2. Queries pgvector for top 50 similar jobs by cosine distance.
3. Cohere reranking refines the top 25 candidates.

4. Scores each candidate on four equally-weighted factors (25% each): skills match, experience match, education match, and semantic similarity.
5. Generates explanations for the top 10 using LLM.

Data Flow: Resume PDF → Text Extraction → LLM Parsing → Structured JSON → Embedding Generation → Vector Search (50 candidates) → Reranking (25 candidates) → Multi-Factor Scoring → Top 10 with Explanations → JSON Response

Infrastructure: PostgreSQL with pgvector stores job data and vectors, Redis handles caching and metrics storage, and Celery workers process batch operations.

Key Design Decisions and Rationale

- A multi-stage filtering pipeline is used for recommendations. Vector search pre-filterig is used to identify 50 promising candidates, which is then refined to 25 using Cohere's cross-encoder reranking, and then applies expensive multi-factor scoring only to these finalists. This balances speed and quality, keeping total latency under 2 seconds.

- Caching is used for heavy computational tasks that require a lot of compute. LLM calls are the primary bottleneck and cost driver. The system caches parsed resumes (24h TTL), embeddings (7d TTL), and complete recommendations (1h TTL), all keyed by SHA-256 hashes.This reduces costs and latency.

- I adopted the JSON Resume Schema from jsonresume.org to parse the resumes since it provides a well-documented, portable structure with existing Pydantic models and clear validation rules.

- For Job Descriptions, I adopted the Job Description Schema from jsonresume.org since it is very structured and well-organized. But I noticed the structure had to be improved a bit to improve the calculation of different scoring factors. So, certain adjustments were made.

- Reranking with Fallback is used here since Cohere reranker was used with free trial keys. The system gracefully falls back to vector ranking upon failure (e.g., API rate limits/costs), ensuring reliability.

- Gemini Pro is used for high-accuracy resume parsing, given resumes are often messy and unstructured. Gemini Flash is employed for parsing job descriptions (as they are generally well-written) and for cost-efficient, fast explanation generation. This mixed model approach optimizes the quality-cost tradeoff.

- gemini-embedding-001 is used for embedding since it can embed multilingual documents very well and is ranked very high on the MTEB Leaderboard which suggests high multilingual embedding performance. http://huggingface.co/spaces/mteb/leaderboard

- Celery (distributed task queue) and RabbitMQ(message broker) Broker-Worker architecture is used to asynchronously handle computationally intensive tasks like batch job ingestion, decoupling them from the main FASTAPI web server for improved scalability and reliability.

Technology Choices and Performance Metrics

Technology Stack Rationale:

Gemini API - It provides both parsing and embedding capabilities from a single provider, simplifying API management. Pro variant offers excellent extraction accuracy for complex resume formats, while Flash enables fast parallel explanation generation.It provides state of the art multilingual embedding capabilities with gemini-embedding 001 as suggested on the MTEB leaderboard and good multilingual language capabilities for the pro and flash models as suggested by the MMLU-Pro leaderboard.

pgvector (PostgreSQL Extension) - Honestly speaking, although it was required by specification and I had never used it before, it proved excellent in practice. It allowed for vector similarity and structured queries to execute in one database without data duplication. Single database for both structured data and embeddings simplifies deployment and reduces operational complexity.

Redis - Redis is used for caching since it is very popular, an industry standard and also very easy to use. It serves dual purposes: key-value caching with automatic TTL management, and also for metrics storage due to its capability to manage high volume data. The In-memory performance provides sub-millisecond cache lookups and its persistence options prevent data loss during restarts.

FastAPI : Fast API is used because it delivers modern Python async capabilities with automatic OpenAPI documentation. Type hints enable compile-time error detection. Native async/await support handles concurrent requests efficiently. Dependency injection simplifies testing. Also, it is what I have been using for quite some time now and I'm comfortable with it.

Celery + RabbitMQ : It provides very reliable async task processing for batch job ingestion. Worker pools enable concurrent processing of multiple jobs. RabbitMQ broker is suitable for high-volume, enterprise-scale task queues. Task retry logic handles transient failures gracefully.

Cohere Rerank API: This is used for reranking, since it specializes in relevance reranking with cross-encoder models and is one of the best out there for reranking.. Its free tier supports development and testing.

Langchain: Langchain is good for LLM orchestration and provides access to a lot of tools. Normally, I would refrain from using Langchain in production due to it being a very unstable framework and other known issues but for the purposes of this task, it is good enough.

Performance Benchmarks

Here are some performance reports for time taken for different operations on average:

LLM Reranking: 583 ms ~ 0.58 seconds

Database Query: 40 ms ~ 0.04 seconds

Recommendation Generation: 3760 ms ~ 3.76 seconds

Resume Parsing: 21829 ms ~ 21.8 seconds

Challenges Faced:

- **Variable Resume Formats:** Handled with detailed LLM prompt engineering and Pydantic validation to ensure extraction accuracy.
- **Speed vs Accuracy Tradeoff for Recommendations:** Solved with a two-stage retrieval process.
- **API Rate Limits with Cohere Reranker:** Implemented automatic fallback to vector ranking, exponential backoff for retries, and configuration flags to maintain system functionality.

Future Improvements:

Evaluation Framework: Generate synthetic ground truth with labeled matches to measure precision/recall. Collect user feedback (thumbs up/down on recommendations) to improve recommendations.

Enhanced Multilingual support: Here, gemini-embedding-001 was used for multilingual embedding support which is near state of the art in terms of performance. But for generative language models, I would go with some other models that are better in terms of multilingual performance.

Production Features and Scalability: Implement pgvector HNSW indexing for faster searches at scale. Right now, I did not use HNSW indexing here because I later discovered that it does not support embeddings more than 2000 dimensions and I had already used 3072 dimension embeddings and did not wanna experiment with embeddings less than 2000 dimensions due to time constraints.Use multiple redis instances. Add more server instances for scaling to tens and hundreds of thousands of requests and add a load balancer to balance load on different server instances.

Recommendation Engine Scoring Enhancements: I would get semantic similarity scores not just for the overall resume and job descriptions but also for different sections like education, experience, skills and sort of combine rule-based scores with semantic similarity scores for different sections to get better scoring. I would have to try it out to measure performance but after some refinements to this strategy, it should give some scoring improvements. Also, I would add some intelligent matching for skills that would go beyond just direct keyword matching.