# Job Recommendation System Report

**Project Members:**
Boutadghart Imran
Fazaz Houssam
Sirgiane Ouical

**Supervised by:**
Prof. Yassine AFOUDI

December 28, 2025

## Contents

# 1 General Introduction

This report details the design and implementation of an AI-powered Job Recommendation System. The project aims to solve the challenge of efficiently matching job seekers with relevant job opportunities by leveraging modern Artificial Intelligence (AI) and Machine Learning (ML) techniques. The system automates the extraction of user profiles from resumes and employs a sophisticated weighted algorithm to rank job listings based on semantic and keyword compatibility.

# 2 Project Context Objectives

The recruitment process is often time-consuming for both candidates and employers. Candidates struggle to find jobs that match their specific skill sets, while employers are inundated with irrelevant applications.

Our solution provides:

- **Automated Resume Parsing**: Extracting structured data (skills, experience, contact info) from PDF/DOCX resumes using Generative AI.
- **Intelligent Matching**: Going beyond simple keyword matching by using vector embeddings to understand the semantic meaning of job descriptions and user profiles.
- **Centralized Aggregation**: Gathering jobs from multiple sources into a single platform.

# 3 Technical Architecture

## 3.1 Technology Stack

The system is built using a modern, robust tech stack ensuring scalability and performance:

- **Backend**: Python 3.10+, FastAPI (high-performance async framework).
- **AI/ML**: Google Gemini API (for text extraction and vector embeddings), Scikit-learn (cosine similarity).
- **Database**: SQLite).
- **Frontend**: HTML5, CSS3 (Modern/Glassmorphism design), Vanilla JavaScript.
- **Testing**: Pytest.

## 3.2 System Pipeline

The data flow within the application follows these steps:

1. **Data Ingestion (Resume)**: User uploads a resume. The system uses `PyPDF2` to read the file.
2. **AI Extraction**: The raw text is sent to the Google Gemini API with a specific prompt to extract a structured JSON object containing: Name, Contact Info, Skills, Education, and Experience.

3. **Job Aggregation**: The system fetches jobs from external APIs (Adzuna, Jooble) or falls back to a mock data generator (see below).

4. **Embedding Generation**: Both the user profile and job descriptions are converted into high-dimensional vectors using the `models/embedding-001` model.

5. **Ranking Engine**: The system calculates match scores and sorts jobs for the user.

# 4 Implementation Details

## 4.1 Recommendation Algorithm

The core value of the system lies in its recommendation engine (`RecommendationService`). Unlike simple keyword matching, we rely on a **Weighted Scoring Algorithm** that combines four distinct metrics:

- **Skills Match (40%)**: Calculates the intersection of user skills and job required skills.
- **Title Match (25%)**: Fuzzy matching between the user's desired roles and the job title.
- **Embedding Similarity (20%)**: Uses **Cosine Similarity** between the user's profile vector and the job description vector to capture semantic relevance (e.g., matching "ML Engineer" with "Data Scientist" even if words differ).
- **Experience Match (15%)**: Heuristic scoring based on years of experience and relevance of past job titles.

```
# backend/services/recommendation.py

overall_score = (
    0.25 * title_score +
    0.40 * skills_score +
    0.15 * experience_score +
    0.20 * embedding_score
)
```

Listing 1: Weighted Score Calculation

## 4.2 Data Sources  Mock Data Strategy

The system is designed to aggregate jobs from real-time APIs like Adzuna and Jooble. However, due to **Terms of Service (ToS)** restrictions on scraping LinkedIn and rate limits/availability of public keys for other APIs, we implemented a robust **Mock Data Generator**.

The `JobAggregator` service includes a fallback mechanism:

- It attempts to fetch from configured APIs first.
- If API keys are missing or requests fail, it seamlessly serves a curated list of mock jobs covering diverse sectors (Software, Healthcare, Finance, etc.).
- This ensures the application is always testable and demonstrable even without external dependencies.

# 5 Testing Quality Assurance

We adopted a Test-Driven focus to ensure system reliability.

## 5.1 Testing Framework

The project uses **pytest** for the test suite, configured via `pytest.ini`.

## 5.2 Test Coverage

Testing covers critical components of the system:

- **Unit Tests**:
  - `tests/test_resume_extraction.py`: Validates PDF/DOCX parsing and JSON structure.
  - `tests/test_similarity.py`: Verifies math behind cosine similarity and weighted scoring.
- **Integration Tests**:
- `tests/test_job_aggregation.py`: Ensures job fetching logic works (including mock fallbacks).
- **API Tests**:
- `tests/test_api.py`: Checks HTTP endpoints (GET/POST) for profiles and recommendations.

Testing covers critical components of the system. We achieved a total coverage of 69%, with high coverage in core business logic models and services.

| Name | Stmts | Miss | Cover |
|---|---|---|---|
| backend__init__.py | 1 | 0 | 100% |
| backend\api__init__.py | 0 | 0 | 100% |
| backend\api\jobs.py | 63 | 33 | 48% |
| backend\api\profile.py | 92 | 60 | 35% |
| backend\api\recommendations.py | 52 | 24 | 54% |
| backend\config.py | 21 | 0 | 100% |
| backend\database__init__.py | 0 | 0 | 100% |
| backend\database\db.py | 14 | 2 | 86% |
| backend\main.py | 45 | 7 | 84% |
| backend\models__init__.py | 0 | 0 | 100% |
| backend\models\job.py | 60 | 0 | 100% |
| backend\models\user.py | 70 | 0 | 100% |
| backend\services__init__.py | 0 | 0 | 100% |
| backend\services\embedding_service.py | 64 | 19 | 70% |
| backend\services\job_aggregator.py | 89 | 12 | 87% |
| backend\services\recommendation.py | 135 | 27 | 80% |
| backend\services\resume_extractor.py | 110 | 68 | 38% |
| **TOTAL** | **816** | **252** | **69%** |

Table 1: Detailed Test Coverage (26 passed, 1 skipped in 63.32s)

Coverage analysis targets the `backend` package to ensure core business logic is validated. Key services like `job_aggregator.py` (87%) and `recommendation.py` (80%) are well-tested. Lower coverage in `resume_extractor.py` and API endpoints is due to reliance on external API mocks and integration boundaries.

# 6 Conclusion

The recommendation system successfully demonstrates the application of Generative AI in the HR domain. By combining traditional keyword matching with semantic vector search, it provides highly relevant results. The robust architecture, coupled with comprehensive testing and graceful fallbacks, ensures a reliable user experience.