Intelligent Resume-Based Job Suggestion System

AI-powered job matching using AWS Bedrock, MongoDB & Streamlit

By: Kanish Midhun

Problem Statement

- - Job seekers manually search and filter through hundreds of postings.
- - Recruiters struggle to match candidates at scale.
- - Resumes are unstructured and vary by format.
- - There is a need for an AI-powered, resume-aware job recommendation system.

Motivation & Use Cases

- Motivation:
- - Use LLMs and embeddings to understand resumes.
- - Reduce time to discover relevant jobs.
- - Provide transparent, explainable matching.

- Use Cases:
- - Fresh graduates looking for suitable roles.
- - Experienced professionals exploring targeted opportunities.
- - Career guidance platforms or university placement cells.

High-Level Architecture

Streamlit UI
(Resume Upload)

AWS S3
(Resume Storage)

Resume Lambda
Claude Parsing + Titan
Embeddings

Job Fetch Lambda
(JSearch API)

MongoDB Atlas
(resumes, jobs,
matches)

Job Matcher Lambda
Hybrid Ranking Engine

Streamlit UI
(Top Matches +
Heatmap)

Detailed Data Flow

- End-to-End Flow:

- 1. User uploads resume via Streamlit.
- 2. Resume stored in S3 → S3 event triggers Lambda.
- 3. Lambda extracts text, calls Claude to parse structured JSON.
- 4. Titan generates embeddings from parsed resume.
- 5. Resume JSON + embeddings stored in MongoDB.
- 6. Job Fetch Lambda queries JSearch API using inferred job title.
- 7. Jobs enriched, deduplicated, embedded & stored in MongoDB.
- 8. Job Matcher Lambda computes scores and writes final matches.
- 9. Streamlit UI reads from MongoDB and displays ranked job list.

Resume Parsing Pipeline

- Steps:
- 1. S3 triggers Resume Lambda when a PDF is uploaded.
- 2. PyPDF2 extracts raw text from the resume.
- 3. Claude (via Bedrock) receives a structured parsing prompt.
- 4. Output is cleaned JSON with name, contact, skills, education, experience.
- 5. Titan embeddings are generated from the merged resume text.
- 6. Resume document + embedding stored in MongoDB (resumes collection).

Example Parsed Resume JSON (Conceptual)

- {
- "name": "Jane Doe",
- "email": "jane.doe@example.com",
- "skills": ["Python", "Data Analysis", "SQL"],
- "experience": [
- {
- "title": "Data Analyst",
- "company": "ABC Corp",
- "start_date": "2021-01",
- "end_date": "2023-03"
- }
-],
- "education": [
- {"degree": "B.Sc.", "field": "Computer Science"}
-]
- }

- This JSON is produced by Claude from the raw resume text.

Job Fetching Pipeline (JSearch API)

- 1. Job Fetch Lambda reads the user's inferred job title from their parsed resume.
- 2. It calls the JSearch API (RapidAPI) with a query like 'nurse jobs in India'.
- 3. The API returns a list of job postings with title, company, description, and links.
- 4. Each job is enriched with a short summary and key skills via Claude.
- 5. Titan embeddings are computed for each job description.
- 6. Jobs are deduplicated and stored in MongoDB (jobs collection).

Example Job Record (Conceptual)

- {
- "title": "Registered Nurse - ICU",
- "company": "City Hospital",
- "location": "Chennai, India",
- "description": "Provide ICU patient care, monitor vitals, coordinate with doctors...",
- "job_link": "https://example.com/job123",
- "skills": ["ICU nursing", "Patient monitoring", "BLS", "ACLS"],
- "embedding": [... 1536-d vector ...],
- "created_at": "2025-11-01T10:00:00Z"
- }

- This enriched job data is used by the ranking engine.

Job Deduplication Logic

- To avoid repeated job postings, the pipeline uses:

- - A hash of the job description text.
- - Combination of (title + company + user_id).
- - Job link uniqueness check.

- If any of these match an existing job record for the same user, the new job is skipped.

Hybrid Ranking Engine

- Final score is computed using:

- final_score = 0.55 * semantic_similarity
- + 0.25 * keyword_overlap
- + 0.10 * recency_weight
- + 0.10 * popularity_score

- • semantic_similarity: cosine similarity between resume and job embeddings
- • keyword_overlap: overlap between resume tokens and job tokens
- • recency_weight: higher for recently fetched jobs
- • popularity_score: based on metadata (e.g., apply link, employer rating)

Skill-Gap Detection & Explanation

- Claude is used again to compare:
- - Extracted resume skills
- - Required job skills

- It outputs:
- • missing_skills: list of skills the candidate does not mention
- • match_reason: short explanation why the job fits the candidate

- These are stored with each match and visualized as a skill-gap heatmap in Streamlit.

Streamlit User Interface

- The UI provides:

- - Resume upload (PDF) → upload to S3 + trigger pipeline.
- - 'Refresh Jobs' button → calls API Gateway → Job Fetch Lambda.
- - 'Load Matches' → reads MongoDB matches collection.
- - Top ranked jobs with scores and explanations.
- - Clickable job links to apply.
- - Skill-gap heatmap (matplotlib) .

Daily Auto Refresh (Optional)

- Design:
- 1. User enables 'Daily Refresh' from the UI (or is registered in a list).
- 2. A scheduled CloudWatch Event or external cron triggers Job Fetch Lambda daily.
- 3. For each registered user_id, it re-fetches and re-ranks jobs.
- 4. Updated matches are written to MongoDB.
- 5. The next time the user opens the UI, they see fresh recommendations.

Performance & Scalability

- Performance:
- - End-to-end flow can complete in a few seconds (depending on Bedrock & API latency).
- - Embeddings and ranking are efficient once data is in MongoDB.

- Scalability:
- - Lambda functions scale horizontally with parallel requests.
- - MongoDB Atlas can be scaled vertically and horizontally.
- - S3 is highly scalable for file storage.
- - Entire architecture is serverless and pay-per-use.

Error Handling & Robustness

- Key strategies:
- - Try/Except around external API calls (JSearch, Bedrock).
- - Logging to CloudWatch for each Lambda.
- - Fallback behaviors if Claude output is not valid JSON.
- - Timeouts and retries for network operations.
- - Graceful UI messages when no resume or matches are found.

Future Enhancements

- - Support multi-language resumes & job descriptions.
- - Integrate additional job boards and APIs.
- - Add user-specific preference learning (e.g., location, salary).
- - Build an admin dashboard for analytics.
- - Experiment with fine-tuned embedding models for specific domains.

Conclusion & Repository

- This project demonstrates:
- - End-to-end AI job recommendation using LLMs and embeddings.
- - A complete serverless pipeline on AWS.
- - Integration of Bedrock, MongoDB, JSearch API, and Streamlit.

- GitHub Repository:
- https://github.com/KanishMidhun/intelligent_resume_based_job_suggestion

- Thank you!