ResumeRAG — Architecture Plan & Project README

Generated: October 4, 2025

Overview

ResumeRAG is a hackathon-ready project that enables uploading, parsing, embedding, searching, and matching resumés to job descriptions. It uses Django + Django REST Framework for backend APIs, React + Tailwind for frontend, and a vector search solution for semantic similarity (FAISS / local sentence-transformers or Qdrant). This document contains architecture, API specifications, README content, deployment notes, seed data, test credentials, and judge-focused checklist.

Table of Contents

1. Architecture Summary

High-level components:

- Frontend: React + Tailwind CSS (SPA). Pages: /upload, /search, /jobs, /candidates/:id

- Backend: Django + Django REST Framework (REST API endpoints listed below)

- Database: PostgreSQL (production) or SQLite (dev)

- Vector Search: FAISS (local) or Qdrant (managed). Alternatively store vectors in DB using pgvector (Postgres).

- Embedding: SentenceTransformers locally OR OpenAI Embeddings (if available)

- File storage: Local filesystem for dev (MEDIA_ROOT) or S3 for prod

- Worker queue: Celery + Redis for background parsing & embedding

- Auth: JWT (SimpleJWT in Django) + role flags (recruiter, user)

- Rate limiting: Django Ratelimit or custom middleware

- Idempotency: Middleware using Idempotency-Key header + DB table

2. System Components & Data Flow

1. User uploads resumes (POST /api/resumes). Files stored; a background job is enqueued.

2. Worker extracts text (PDF/TXT/DOCX -> text) and structured fields (name, email, phone, skills) via rule-based + regex + optional ML parser.

3. Text is chunked (passage-level) and embeddings computed for each chunk. Store embeddings with references to resume_id and page/chunk metadata.

4. Resume 'document' record updated status -> processed, plus detections of PII redaction markers.

5. Search requests (POST /api/ask or GET /api/resumes?q=) compute query embedding and do nearest-neighbor search, return ranked results with snippet evidence (text and page/chunk pointer).

6. Jobs can be created (POST /api/jobs). Matching endpoint computes similarity between job embedding and resume embeddings, aggregates scores, lists missing requirements and evidence snippets.

3. Data Models (simplified)

```
- User (Django auth user extended)

- id, username, email, password (hashed), role {candidate, recruiter, admin}

- Resume

- id, user (owner), filename, uploaded_at, status {processing, processed, failed},
redacted (bool), summary(text), pii_removed_version_path, original_file_path

- ResumeChunk

- id, resume (FK), chunk_text, chunk_order, page_number (optional), embedding (vector
reference), char_start, char_end

- EmbeddingStore (if using DB vectors)

- id, resume_chunk (FK), vector (pgvector)

- Job

- id, owner, title, description, requirements (array), created_at, embedding (vector)

- MatchReport

- id, job, resume, score, evidence (JSON list of {chunk_id, text_snippet, start, end}),
missing_requirements

- IdempotencyKey

- key, user, endpoint, request_hash, response_snapshot, created_at
```

4. API Specification

Common headers:

- Authorization: Bearer <JWT>

- Content-Type: multipart/form-data or application/json

- Idempotency-Key: <uuid> (for POST create endpoints)

Endpoints (required):

- GET /api/health
- 200 { "status":"ok" }

- GET /.well-known/hackathon.json
- static metadata per hackathon requirements

- POST /api/register
- Request: { "username","email","password", "role": "recruiter"|"candidate" }
- Response: 201 { "id","username","email","role" }

- POST /api/login
- Request: { "username","password" }
- Response: 200 { "access": "<jwt>", "refresh":"<refresh>" }

- POST /api/resumes (multipart)
- Headers: Idempotency-Key
- Body: file=<file.pdf>, owner_id (optional for recruiter upload)
- Background: enqueues parse & embed job
- Response: 202 { "id","filename","status":"processing" }

- GET /api/resumes?limit=&offset;=&q;=
- Returns paginated list: { "items":[...], "next_offset": <int|null> }
- q param does text or semantic search fallback

- GET /api/resumes/:id
- Returns resume metadata (redacted by default for non-recruiters) and list of chunks (no full PII unless recruiter)

- POST /api/ask
- Body: { "query": "experience with django", "k": 5 }
- Response: 200 {
"query_id":"<uuid>",

```
"answers":[
{ "resume_id", "score", "evidence":[ { "chunk_id", "text", "page", "start", "end" } ] }
]
}
```

```
- POST /api/jobs
- Headers: Idempotency-Key
- Body: { "title","description","requirements": ["a","b"] }
- Response: 201 { "id","title","created_at" }
```

```
- GET /api/jobs/:id
- Response: job metadata
```

```
- POST /api/jobs/:id/match
- Body: { "top_n": 10 }
- Response: 200 {
"job_id", "matches":[ { "resume_id","score","evidence":[...],
"missing_requirements":[...] } ]
}
```

4.1 Example: Upload Resume

```
Request (curl):
curl -X POST "https://your-domain/api/resumes" \
-H "Authorization: Bearer <token>" \
-H "Idempotency-Key: 123e4567-e89b-12d3-a456-426614174000" \
-F "file=@/path/to/resume.pdf"

Response (202):
{
"id": "res_123",
"filename": "resume.pdf",
"status": "processing",
"uploaded_at": "2025-10-04T05:00:00Z"
}
```

4.2 Example: Ask Query

```
Request:
POST /api/ask
Body: { "query": "django rest framework experience", "k": 5 }

Response (200):
{
"query_id":"q_001",
"answers":[
{
"resume_id":"res_123",
"score": 0.87,
"evidence":[
{ "chunk_id":"c_11", "text":"Worked on Django REST Framework to build APIs...",
"page":1, "start":120, "end":240 }
]
}
]
}
```

5. Authentication & Authorization

Use Django REST Framework + SimpleJWT for token auth.
Roles: candidate, recruiter, admin.
Access rules:
- Candidates can upload and view their resumes (redacted view for others).
- Recruiters can see unredacted PII fields for résumés they have permission for.
- Admin can do anything.
Token lifetime: access 15 min, refresh 7 days.

6. Rate limiting, Idempotency, Pagination & Errors

```
Rate limiting: Enforce 60 req/min/user. If exceeded return 429: { "error": {
"code":"RATE_LIMIT" } }
Idempotency: All POST create endpoints must accept Idempotency-Key header. Implement
middleware storing key, endpoint, user -> response snapshot for 24 hours.
Pagination: Support ?limit=&offset;=. Return { items: [...], next_offset: }
Errors format (uniform): { "error": { "code": "FIELD_REQUIRED", "field": "email",
"message": "Email is required" } }
```

7. Embedding Pipeline & Matching Algorithm

Parsing & chunking: Extract text via PyMuPDF or pdfminer. Keep page numbers. Split into overlapping chunks (e.g., 500 tokens with 50 token overlap).
Embeddings: Use SentenceTransformers or OpenAI embeddings. Compute embedding per chunk and store.
Search & Match: For a query: compute embedding -> k-NN search over chunk embeddings -> aggregate chunk scores to produce resume-level score. Deterministic ranking: tie-break by uploaded_at or resume.id.

8. PII Redaction Policy

By default redact PII (email, phone, address, national ID) in responses unless requestor is a recruiter with permission. During parsing, detect PII via regex and NLP NER (spaCy) and store both original and redacted versions.

9. Deployment & Dev Notes

Dev: SQLite, local filesystem, no vector DB. Use Celery worker with Redis. Prod: PostgreSQL, S3 for media, Qdrant or FAISS + persisted index, HTTPS via nginx, gunicorn/uvicorn. Containers: Dockerfile for backend, docker-compose for local orchestration (db, redis, qdrant).

10. README (short)

Quickstart:\n1. Clone repo\n2. Backend: python -m venv venv && source venv/bin/activate; pip install -r requirements.txt; python manage.py migrate; python manage.py createsuperuser; celery -A proj worker -l info (if Celery used)\n3. Frontend: cd frontend && npm install && npm run dev

11. Seed Data & Test Credentials

```
Test users (seed):
- recruiter@example.com / TestPass123 (recruiter)
- candidate1@example.com / TestPass123 (candidate)
- admin@example.com / AdminPass123 (admin)

Seed resumes:
- resume_john_doe.pdf (Django REST work)
- resume_jane_dev.pdf (Python, ML)
```

12. Judge Checklist & How to Test

Ensure /api/health and /.well-known/hackathon.json present. Upload 3+ resumes, POST /api/ask returns answers with evidence, POST /api/jobs/:id/match returns matches with evidence and missing_requirements, Pagination works, Rate limit enforced, Idempotency validated, PII redaction enforced.

Short Architecture Note (100-200 words)

ResumeRAG uses a modular architecture where the Django API handles authentication, upload, and CRUD. Asynchronous Celery workers process files to extract text and compute embeddings. Embeddings are stored in a vector store (FAISS/Qdrant/pgvector) for semantic retrieval. Postgres stores metadata. The React frontend communicates via REST and displays evidence snippets returned from the backend. Rate limiting, idempotency middleware, and role-based access provide robustness and reproducibility. AI components are pluggable so you can swap local inference for cloud APIs without changing the main app flow.

Final Notes & Next Steps

I can generate the Django project scaffold, Celery tasks, or a React frontend scaffold next. Tell me which and I will produce code files.