Architecture Plan: AI Resume Analyzer

Microservices-Based Architecture (Full-Stack + ML)

This document outlines the system architecture for the AI Resume Analyzer. The system is designed as a set of containerized microservices to ensure separation of concerns between the business logic (NestJS) and the heavy computational logic (Python/FastAPI).

1. High-Level Architecture

The system follows a **Hub-and-Spoke** pattern where the **NestJS Backend** acts as the central API Gateway and Orchestrator. The Frontend never communicates directly with the ML Service or the Database.

```
graph TD
    User[User / Browser] -->|HTTPS| Frontend[Next.js Client]
    Frontend -->|REST API / JSON| Backend[NestJS API Gateway]

    subgraph "Core Services"
        Backend -->|Auth & Data| DB[(PostgreSQL)]
        Backend -->|File Upload| Storage[Local Disk / S3]
        Backend -->|HTTP Request - JSON| ML[Python FastAPI Service]
    end

    subgraph "ML Pipeline"
        ML -->|Parsing| Spacy[spaCy / NLTK]
        ML -->|Vector Embeddings| Transformer[Sentence-Transformers]
        ML -->|Similarity Calc| Scikit[scikit-learn]
    end
```

2. Component Details

A. Client Layer (Frontend)

- **Tech:** Next.js (App Router), TailwindCSS, React Hook Form.
- **Role:** Handles user interaction, state management, and data visualization.
- **Key Features:**
    - **App Router:** Uses server-side rendering (SSR) for the dashboard and static generation for landing pages.
    - **React Hook Form:** Manages complex forms (Resume upload + Job Description input) with client-side validation.

- **PDF Preview:** Renders the uploaded PDF file alongside the analysis results.

B. API Gateway & Business Logic (Backend)

- **Tech:** NestJS, Prisma ORM, JWT, Multer.
- **Role:** The single source of truth. Handles authentication, data persistence, and orchestration.
- **Key Responsibilities:**
  - **Auth:** Validates JWT tokens via Passport strategies.
  - **Uploads:** Uses Multer to intercept file uploads. Files are saved to a volume (Docker) or S3, and the file path is stored in Postgres.
  - **Orchestration:** When a resume is uploaded:
    1. Saves file to storage.
    2. Creates a database record (status: PENDING).
    3. Sends a synchronous HTTP request to the **ML Service**.
    4. Updates database with ML results.
  - **Swagger:** Auto-generates API documentation at /api/docs.

C. Compute Engine (ML Service)

- **Tech:** Python, FastAPI, spaCy, Sentence-Transformers.
- **Role:** Stateless worker. Receives text/files, processes them, and returns raw JSON data.
- **Key Pipelines:**
  1. **Text Extraction:** Uses PyPDF2 or pdfminer to extract raw text from PDF.
  2. **NER (Named Entity Recognition):** Uses spaCy to find Names, Emails, and Skills.
  3. **Vectorization:** Uses Sentence-Transformers (e.g., all-MiniLM-L6-v2) to convert Resume text and JD text into high-dimensional vectors.
  4. **Similarity:** Uses scikit-learn (Cosine Similarity) to compare the vectors and generate a match score (0-100%).

D. Data Persistence (Database)

- **Tech:** PostgreSQL (via Prisma).
- **Role:** Relational storage for users and structured data.

3. Data Flow: The Analysis Lifecycle

1. **Upload:** User uploads resume.pdf via Next.js.
2. **Gatekeeping:** NestJS verifies the JWT token.
3. **Storage:** NestJS Multer saves the file to ./uploads/resumes/uid_123.pdf.
4. **Handoff:** NestJS sends a payload to FastAPI:
   POST http://ml_service:8000/analyze
   ```
   {
     "file_path": "/shared_volume/uid_123.pdf",
     "job_description": "We need a React developer..."
   }
   ```

5. **Processing:**
    - FastAPI reads the file from the shared volume.
    - Extracts text -> Cleans text -> Vectorizes -> Calculates Score.
6. **Response:** FastAPI responds to NestJS:
   ```
   {
     "score": 85,
     "skills_found": ["React", "TypeScript"],
     "missing_keywords": ["Docker"]
   }
   ```
7. **Persistence:** NestJS saves this result into the Analysis table in PostgreSQL.
8. **UI Update:** NestJS returns the result to Next.js, which renders the score chart.

4. Database Schema (Prisma Model)

```
// Conceptual Schema

model User {
  id       Int      @id @default(autoincrement())
  email    String   @unique
  password  String
  role     String   @default("CANDIDATE") // CANDIDATE, RECRUITER
  resumes   Resume[]
}

model Resume {
  id         Int       @id @default(autoincrement())
  userId     Int
  filePath    String
  fileName    String
  parsedText  String?    @db.Text
  analyses    Analysis[]
  user       User      @relation(fields: [userId], references: [id])
}

model Analysis {
  id          Int      @id @default(autoincrement())
  resumeId     Int
  jobDescText   String   @db.Text // The JD used for comparison
  matchScore    Float    // 0 to 100
  missingSkills String[] // Array of strings
  createdAt     DateTime @default(now())
```

```
  resume       Resume   @relation(fields: [resumeId], references: [id])
}
```

5. Infrastructure & DevOps (Docker)

We will use **Docker Compose** to spin up the entire stack locally or in production.

Service Structure (docker-compose.yml):

1. **postgres-db:**
   - Image: postgres:15-alpine
   - Ports: 5432:5432
   - Volumes: pgdata:/var/lib/postgresql/data
2. **ml-service:**
   - Context: ./ml_engine
   - Command: uvicorn main:app --host 0.0.0.0 --port 8000
   - Volumes: ./uploads:/app/uploads (Read access to uploaded files)
3. **backend-api:**
   - Context: ./backend
   - Ports: 3000:3000
   - Environment: DATABASE_URL, ML_SERVICE_URL=http://ml-service:8000
   - Volumes: ./uploads:/app/uploads (Write access to save uploads)
   - Depends On: postgres-db, ml-service
4. **frontend-client:**
   - Context: ./frontend
   - Ports: 3001:3000
   - Environment: NEXT_PUBLIC_API_URL=http://localhost:3000

6. Implementation Strategy (Next Steps)

1. **Initialize Monorepo:** Create a folder structure with /frontend, /backend, and /ml_service.
2. **Database First:** Set up Prisma and Postgres to define the data models.
3. **ML Prototype:** Build a simple FastAPI endpoint that accepts text and returns a dummy score.
4. **Backend Integration:** Build the NestJS controller to handle uploads and call the ML prototype.
5. **Frontend Polish:** Build the UI to consume the API.
6. **Refine ML:** Replace dummy logic with actual spacy and sentence-transformers implementation.