ChatGPT

AI Resume & Job Match Platform – Comprehensive Build Instructions for Cursor

Overview

Build a **full-stack web application** that allows users to upload a résumé (PDF), parse its content, compare it against job descriptions using AI and similarity metrics, and return an overall match score with detailed feedback. The project should demonstrate modern front-end and back-end skills and highlight real-world problem solving, data processing and AI integration. Use **Next.js 14** for the front end and either **Next.js server actions** alone or **Next.js + Spring Boot** for a separate back end. Use a relational database (e.g., PostgreSQL via **Prisma** or Spring Data JPA) to persist user data, résumés, parsed information, jobs and matching results.

The guide below explains the architecture, technology choices, key features, data flow, and implementation details. Citations from authoritative sources justify the design choices, performance techniques and best practices.

1. Technology stack

Layer	Recommended technologies & rationale
Front end	**Next.js 14 with the App Router**. Next.js offers server components, server actions and automatic image optimization. Server actions allow data mutations from client components without separate API routes [1]; use them for resume uploads and job creation. The framework also supports SSR/SSG/ISR for better performance and SEO [2]. **Tailwind CSS** for styling, **Recharts** for charts (simple API and clean SVG rendering [3]). Use **NextAuth.js** for authentication because it abstracts session management and sign-in/out complexities [4]; install via `next-auth@beta` and generate a secret key [5].
Back end	Option A – **Server actions only**: Use Next.js server actions for all CRUD operations. This reduces code overhead since server functions live alongside components and can process form data, handle file uploads and interact with the database in one file [6]. Option B – **Spring Boot**: create a REST API using Spring Boot 3.4 with Java 24 support and virtual threads for scalability [7]. Expose endpoints for résumé upload/parsing, job management and matching. Configure CORS using `@CrossOrigin` or global configuration [8] [9] so the Next.js front end can call the API. Use Spring Security for JWT-based auth and integration with NextAuth.
Database	**PostgreSQL** hosted on Supabase, PlanetScale or a self-managed instance. Use **Prisma ORM** for Next.js or **Spring Data JPA** for Spring Boot.

Layer	Recommended technologies & rationale
AI & résumé parsing	Use a Python micro-service (FastAPI) or Node scripts to parse PDFs and extract structured résumé data. DIY parsing with pdfminer and spaCy is possible [10] but may lack accuracy; consider using Affinda's Python API for high-quality parsing [11]. For job matching, compute embeddings with OpenAI's embedding API [12] or Cohere; measure similarity via cosine similarity and Jaccard for skills [13]. Optionally call an LLM (OpenAI, Claude or Llama-2) to generate a suitability report [14].
Notifications (optional)	Use email (e.g., SendGrid), Slack or Discord webhook to notify users about new matching jobs.

2. Key features

1. **User authentication & onboarding**
2. Sign-up/sign-in using NextAuth with email/password or OAuth (Google, GitHub). Create a `users` table with id, name, email, hashed password (if using credentials), role (user/admin) and timestamps.

3. Provide account settings page where users can upload/update their résumé and manage their profile.

4. **Résumé upload and storage**

5. Use a **Next.js server action** or a Spring endpoint to accept file uploads. With server actions, apply the `'use server'` directive and access `FormData` directly [15]. Save the PDF to object storage (e.g., Supabase Storage or AWS S3) and store metadata (file name, size, upload date, user_id) in the database. Show progress indicators on the UI.

6. Validate file size/type (limit to PDF). Generate a preview using `react-pdf` if needed.

7. **Résumé parsing**

8. After upload, call the résumé parsing service. For high-quality extraction use Affinda's Python library which uses deep-learning models to extract names, contact info, education, work history, skills and languages [11]. Alternatively, implement a Python script using `pdfminer.six` and spaCy to extract text and named entities [10]. Save parsed data to a `resumes` table as JSON. Provide fallback to manual editing in the UI.

9. **Job postings**

10. Provide a dashboard for recruiters/admins to create job postings. Use server actions or Spring endpoints for CRUD operations. Each job includes title, description, required skills, years of experience, job type (remote, on-site), location and other metadata. Persist jobs in a `jobs` table.

11. Implement job listing pages (public and private) with search and filter (by location, type, skills). Use server-side rendering for SEO. Offer an API route if external systems need to fetch jobs.

12. **Matching algorithm**

13. When a user views a job or runs “match my résumé”, compute a match score:
    - Generate embeddings: call OpenAI’s `embeddings` endpoint on the job description and résumé text [12].
    - Compute the **cosine similarity** between embedding vectors (measure semantic closeness).
    - Extract skills from both documents and compute **Jaccard similarity** (intersection ÷ union) [13].
    - Use `difflib` or Levenshtein ratio for experience match if you implement it like the research paper [13].
    - Combine scores with configurable weights (e.g., 0.6 for semantic match, 0.3 for skill match, 0.1 for experience).

14. Optionally ask an LLM to assess the résumé vs job and produce a human-readable justification or suggestions [14]. Prompt the model with both texts and ask for feedback like “Key match factors, gaps and improvements”. Limit model cost by summarising first.

15. **Results & visualization**

16. Present match results to the user: show overall score, breakdown by category (skills, experience, education), and highlight which parts of their résumé align with job requirements. Use **Recharts** to display bar or radial charts; the library is praised for simplicity and clean SVG output [3].

17. Provide recommendations: highlight missing skills and link to relevant learning resources.

18. **Admin & analytics (optional)**

19. Admins can view user statistics, job engagement and matching outcomes. Create dashboards summarising the number of résumés uploaded, average match scores, distribution of skills, etc. Use server actions to fetch aggregated data and display charts.

20. Expose API endpoints or download CSVs for further analysis.

21. **Notifications (optional)**

22. Allow users to sign up for alerts when new jobs match their résumé above a threshold. Implement a background job (using Next.js cron via Vercel Cron or Spring scheduled tasks) to run matches periodically. Send notifications via email or Discord using webhooks (similar to Firecrawl’s use of Discord [16]). Persist notification preferences in the database.

23. **Security & compliance**

24. Use HTTPS in production. Store secrets (DB connection, API keys) in environment variables. Follow OWASP guidelines: validate inputs, sanitise file uploads, limit file size, and apply CSRF protection on API routes.

25. If using Spring Boot, implement CORS configuration: annotate controllers with `@CrossOrigin(origins="<frontendURL>")` or configure globally via `WebMvcConfigurer` [8] [9].

26. For GDPR/CCPA compliance, include a privacy policy, explain data retention, and allow users to delete their data.

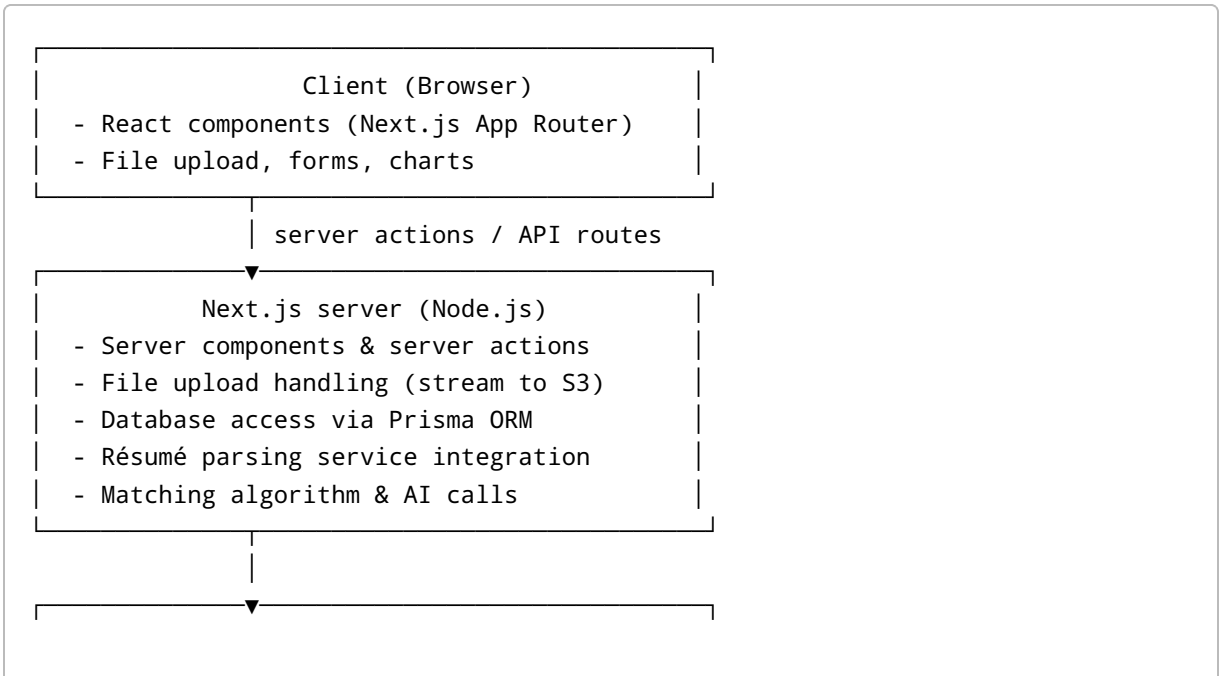
27. **DevOps & quality**

28. **Testing**: Write unit tests (Jest/React Testing Library for front end, JUnit for Spring), integration tests for server actions/API endpoints, and e2e tests (Playwright or Cypress).
29. **CI/CD**: Use GitHub Actions to run linting, tests and build on each PR. Deploy Next.js to Vercel or Netlify; deploy Spring Boot to Render or DigitalOcean.
30. **Docker**: Provide `Dockerfile`s for both services and a `docker-compose.yml` to spin up the stack locally with PostgreSQL.
31. **Code quality**: Configure ESLint, Prettier and TypeScript for Next.js; use SpotBugs and Checkstyle for Java.
32. **Documentation**: Include a README with setup instructions, architecture diagrams and contributions guidelines. Generate API docs via Swagger for Spring Boot. Add commentary on why server actions were chosen vs API routes (server actions are ideal for tightly coupled UI operations; API routes for external integration [17]).

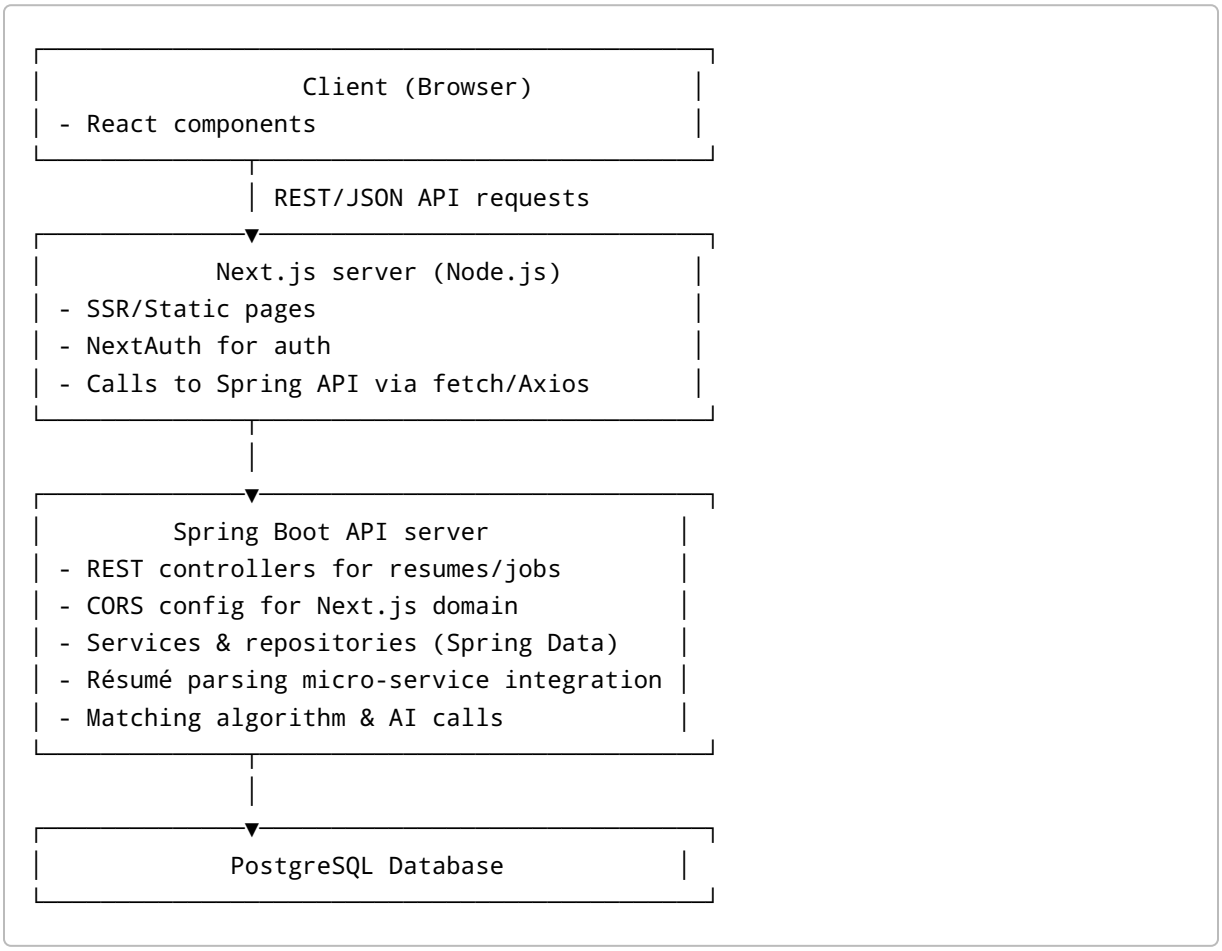
3. Architectural design

Below is a suggested architecture that can be implemented using either a single Next.js app (left) or a Next.js + Spring Boot stack (right).

Option A – Next.js only (full-stack):



Option B – Next.js + Spring Boot:



Choose option A if you want a single deployment and to highlight modern Next.js features. Choose option B if you want to show backend skills with Java and Spring Boot (e.g., concurrency via virtual threads, explicit API design, and CORS control) [7] [8].

4. Key implementation details and code hints

1. **Next.js setup**: initialise a new project using `npx create-next-app@latest my-resume-matcher --typescript --tailwind`. Organise the app using the **App Router**: `app/` for pages; `components/` for UI components; `lib/` for database and AI utils; `actions/` for server actions; `prisma/` for schema. Use environment variables for DB connection and API keys via `.env.local`.

2. **Server actions for file uploads**:

```tsx
// app/(dashboard)/resume/upload-form.tsx
'use client';

import { uploadResume } from '@/actions/resume-actions';

export default function ResumeUpload() {
  const onSubmit = async (formData: FormData) => {
    await uploadResume(formData);
  };
  return (
    <form action={onSubmit} className="space-y-4">
      <input type="file" name="file" accept="application/pdf" required />
      <button type="submit" className="btn">Upload</button>
    </form>
  );
}
```

```tsx
// actions/resume-actions.ts
'use server';

import { prisma } from '@/lib/prisma';
import { parseResume } from '@/lib/parse';
import { uploadToS3 } from '@/lib/storage';

export async function uploadResume(formData: FormData) {
  const file = formData.get('file') as File;
  if (!file) throw new Error('No file');
  // Save file to S3 and get URL
  const url = await uploadToS3(file);
  // Parse the PDF (call Python service or Node parser)
  const parsed = await parseResume(file);
  // Save metadata and parsed JSON to DB
  await prisma.resume.create({ data: { url, parsed, userId: /* get from session
*/ } });
}
```

1. **Python résumé parser micro-service** (FastAPI example):

```python
# parser_service/main.py
from fastapi import FastAPI, UploadFile
from affinda import AffindaAPIClient

app = FastAPI()
affinda_client = AffindaAPIClient("AFFINDA_API_KEY")
```

```
@app.post("/parse")
async def parse(file: UploadFile):
    content = await file.read()
    parsed = affinda_client.parse_resume(content)
    return parsed.to_dict()
```

Deploy this service separately (Docker) and call it from the Next.js server action using `fetch` or `axios`.

1. **Embedding & matching utility**:

```
// lib/matching.ts
import { OpenAI } from 'openai';
import similarity from 'compute-cosine-similarity';
import { jaccardIndex } from './jaccard';

export async function computeMatch(resume: string, job: string, resumeSkills:
string[], jobSkills: string[]) {
  const api = new OpenAI({ apiKey: process.env.OPENAI_API_KEY! });
  const [resumeEmb, jobEmb] = await Promise.all([
    api.embeddings.create({ model: 'text-embedding-ada-002', input: resume }),
    api.embeddings.create({ model: 'text-embedding-ada-002', input: job }),
  ]);
  const cosine = similarity(resumeEmb.data[0].embedding,
jobEmb.data[0].embedding);
  const jaccard = jaccardIndex(new Set(resumeSkills), new Set(jobSkills));
  // Additional metrics can go here
  return { cosine, jaccard, overall: 0.7 * cosine + 0.3 * jaccard };
}
```

1. **NextAuth configuration**:

```
// app/api/auth/[...nextauth]/route.ts
import NextAuth from 'next-auth';
import CredentialsProvider from 'next-auth/providers/credentials';

export const authOptions = {
  providers: [
    CredentialsProvider({
      name: 'Credentials',
      credentials: {
        email: { label: 'Email', type: 'email' },
        password: { label: 'Password', type: 'password' },
      },
      async authorize(credentials) {
        // validate credentials and return user object
      },
```

```
    }),
  ],
  secret: process.env.NEXTAUTH_SECRET,
};
const handler = NextAuth(authOptions);
export { handler as GET, handler as POST };
```

1. **Spring Boot API sample**:

```
@RestController
@RequestMapping("/api/resumes")
@CrossOrigin(origins = "http://localhost:3000")
public class ResumeController {
    private final ResumeService resumeService;
    @PostMapping(consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
    public ResumeResponse upload(@RequestParam("file") MultipartFile file) {
        ParsedResume parsed = resumeService.parseAndSave(file);
        return new ResumeResponse(parsed);
    }
}
```

5. Timeline & deliverables

1. **Week 1 – Planning & setup**
2. Finalise feature scope and choose Option A (Next.js) or Option B (Next.js + Spring Boot).
3. Design DB schema and ER diagram. Set up repository with GitHub, install dependencies and create boilerplate code.
4. Integrate NextAuth authentication and sign-in/sign-up pages.
5. **Week 2 – Résumé upload & parsing**
6. Implement file upload UI and server action (or Spring endpoint). Set up S3 or Supabase Storage.
7. Build the résumé parsing micro-service using Affinda or a custom parser. Test integration and store parsed data in DB.
8. Build a page for users to view and edit parsed résumé data.
9. **Week 3 – Job management**
10. Create pages for recruiters/admins to add and manage job postings. Implement search and filter.
11. Add API routes or Spring controllers for job CRUD. Secure endpoints using role-based auth.
12. **Week 4 – Matching algorithm & AI integration**

13. Implement embedding generation and similarity computation.
14. Tune weighting between semantic similarity and skills match.
15. Integrate LLM calls to produce narrative feedback. Optimize prompt design and summarization to reduce cost.
16. **Week 5 – Visualization & user dashboard**
17. Build match result pages with charts (Recharts). Show category breakdown and highlight matches/mismatches.
18. Build admin dashboard with analytics.
19. Add export/download options for reports (PDF or CSV). Use `reportlab` or `python-docx` in Python if generating docs.
20. **Week 6 – Notifications & polishing**
21. Implement optional notification system (scheduled tasks).
22. Add dark mode toggle, accessibility improvements, and responsive design.
23. Write tests, set up CI/CD and deploy to production. Document the project with a comprehensive README.

6. Final advice

- **Keep components modular.** Separate UI logic from data fetching and computation. Use server actions for operations tied to the UI and API routes (or Spring controllers) for endpoints that might be consumed by external clients [18].
- **Monitor performance.** Use Next.js image optimization to serve modern formats like WebP and reduce layout shifts [19]. Use SSR or ISR for pages that rely on dynamic data to improve SEO and time-to-first byte [2].
- **Stay up to date.** If using Spring Boot, keep dependencies updated (Spring 6.2, Hibernate 6.6, etc.) [20] and adopt virtual threads for concurrency [7]. Explore building native images with GraalVM if necessary [21].
- **Prioritise accuracy and fairness.** When building the matching algorithm, clearly document that the system only provides suggestions and does not make final hiring decisions. Avoid using sensitive data such as race or health conditions; design the system to comply with equal employment opportunity laws and avoid biases.

This comprehensive guide should equip Cursor to build a compelling AI-powered résumé & job matching platform. Follow the timeline and best practices to deliver a polished project that impresses recruiters and demonstrates modern full-stack competence.

---

[1] [17] [18] Sparkle Web - Leading IT Services & Solutions | Surat, India
https://www.sparkleweb.in/blog/server_actions_in_next.js:_the_future_of_api_routes

[2] [19] Next.js Best Practices in 2025: Performance & Architecture
https://www.raftlabs.com/blog/building-with-next-js-best-practices-and-benefits-for-performance-first-teams/

3 Best React chart libraries (2025 update): Features, performance & use cases - LogRocket Blog
https://blog.logrocket.com/best-react-chart-libraries-2025/

4 5 App Router: Adding Authentication | Next.js
https://nextjs.org/learn/dashboard-app/adding-authentication

6 15 File Upload with Next.js 14 and Server Actions | Akos Komuves
https://akoskm.com/file-upload-with-nextjs-14-and-server-actions/

7 20 21 Spring Boot and Java 24: What Developers Need to Know in 2025 - DEV Community
https://dev.to/initialm503/spring-boot-and-java-24-what-developers-need-to-know-in-2025-3jae

8 9 Getting Started | Enabling Cross Origin Requests for a RESTful Web Service
https://spring.io/guides/gs/rest-service-cors/

10 11 Here's How to Extract Skills from a Resume Using Python
https://www.affinda.com/blog/extract-skills-from-a-resume-using-python

12 14 Using OpenAI and SingleStore for Automated Resume Scans + Assessments
https://www.singlestore.com/blog/openai-singlestore-automated-resume-scans-assessments/

13 Hybrid Transformer-Based Resume Parsing and Job Matching Using TextRank, SBERT, and DeBERTa
https://www.internationaljournalssrg.org/IJECE/paper-details

16 Building an AI Resume Job Matching App With Firecrawl And Claude
https://www.firecrawl.dev/blog/ai-resume-parser-job-matcher-python