

Give me 5 project ideas for my ASE course in my MS in CS

ChatGPT said:

Got it 👍 Since this is for your **Advanced Software Engineering (ASE)** course in your MS in CS, you'll want project ideas that aren't just coding-heavy but also demonstrate **software engineering principles** like design patterns, testing, scalability, documentation, teamwork, CI/CD, and architecture. Here are **5 solid project ideas** that balance technical challenge with SE principles:

---

## 1. AI-Powered Bug Triage System

- **Description:** Build a tool that automatically classifies and prioritizes bug reports from issue trackers (e.g., GitHub, Jira) using NLP.
  - **ASE Concepts:**
    - Requirements gathering from real bug datasets.
    - Use **design patterns** (e.g., Observer for notifications, Strategy for classification).
    - **Testing:** Unit + integration + regression tests.
    - CI/CD pipeline to deploy updates.
  - **Extra:** Integrate with GitHub issues and Slack for real-time notifications.
- 

## 2. Smart Code Review Assistant

- **Description:** A web-based tool that suggests improvements in pull requests using static analysis + AI (e.g., checking for complexity, security flaws, style issues).
- **ASE Concepts:**
  - **Architecture design:** Microservices (analysis engine, web interface, DB).
  - **Metrics collection:** Track cyclomatic complexity, code smells.
  - **Continuous Integration:** Auto-run analysis on PRs.

- **Extra:** Could integrate with GitHub Actions or GitLab CI.
- 

### 3. Scalable Online Examination System

- **Description:** A secure online exam platform with proctoring features (screen recording, plagiarism detection, randomized question sets).
  - **ASE Concepts:**
    - **Scalability & load testing** (simulating 1k+ users).
    - **Security:** Encryption, role-based access.
    - **UML modeling & requirement specs.**
    - **Test coverage & DevOps pipeline.**
  - **Extra:** Add NLP-based plagiarism detection on answers.
- 

### 4. Voice-Activated Smart Home Simulator (*ties well with your prior interest in voice control*)

- **Description:** A simulator that controls virtual home appliances (lights, AC, doors) with voice commands + NLP.
  - **ASE Concepts:**
    - **Modular design:** Separate NLP, control logic, UI.
    - **Requirement engineering:** Handle ambiguous voice inputs.
    - **Testing strategy:** Mock devices for integration testing.
    - **Documentation:** API + design docs.
  - **Extra:** Deploy on a cloud backend, expose REST/GraphQL APIs.
-

## 5. Crowdsourced Software Requirements Platform

- **Description:** A collaborative system where users propose, refine, and vote on software requirements for projects.
- **ASE Concepts:**
  - **Requirement elicitation & prioritization techniques.**
  - **Agile boards** with sprints & backlog grooming.
  - **Versioning & traceability of requirements.**
  - **Testing:** Automated acceptance test generation.
- **Extra:** Use gamification (badges, points) to engage contributors.

---

👉 Each of these can be scaled in scope depending on your team size and course expectations.

Do you want me to **rank these ideas by feasibility vs impressiveness** (easy to finish vs likely to wow professors), so you can pick depending on your timeline?

**You said:**

My idea is a resume tailor It edits the uploaded resume based on the uploaded job description. It just replaces the words with new ones keeping the template and personal details same ATS friendly will also be done backend using an API of any LLM would be better But the problem is, it is not more complex like a project of ASE Can you make it look more big

**ChatGPT said:**

Perfect starting point 🚀 — a **resume tailoring system** is a fantastic idea, but yes, on its own it might look too much like a "utility app." To make it **ASE-course worthy**, we can **expand it into a full-fledged software engineering project** by layering in **architecture, scalability, testing, and advanced features**.

Here's how you can make your idea *bigger and more impressive*:

---

## Expanded Project: Intelligent Resume Tailoring & Career Recommendation System

## Core Functionality (your original idea):

- Upload resume + job description.
  - Tailors the resume by:
    - Replacing and enhancing keywords.
    - Keeping formatting, personal details, and template intact.
    - Ensuring ATS-friendliness (correct parsing, no graphics-only resumes).
  - Uses an LLM API (like GPT) for semantic matching.
- 

## Add ASE-Level Depth:

### 1. Multi-Layer Architecture (SE complexity)

- **Frontend:** Web UI for uploading resume + job description.
  - **Backend:** API server with modules for:
    - Resume parser (PDF/DOCX → JSON).
    - LLM-based tailoring service.
    - ATS compliance checker.
  - **Database:** Store user resumes, tailored versions, job descriptions, and feedback history.
  - **Microservices style (optional):** Resume Parser, Tailor Engine, ATS Validator as separate services.
- 

### 2. Advanced Features (to go beyond simple text replacement)

- **Semantic Keyword Expansion:** Not just replace words, but expand synonyms/skills using embeddings.

- **Scoring System:** Match score between tailored resume and job description.
  - **Multiple Templates:** Allow user to pick a design template automatically filled.
  - **Feedback Loop:** Track which resumes got interviews and auto-improve tailoring.
  - **Recommendation Engine:** Suggest additional certifications, projects, or skill phrasing based on the job role.
- 

### 3. Software Engineering Practices (what pros look for)

- **Requirement Engineering:**
    - Functional: resume upload, tailoring, scoring.
    - Non-functional: scalability (many users), data privacy (sensitive resumes).
  - **Design Artifacts:**
    - UML class diagrams, sequence diagrams, architecture diagrams.
  - **Design Patterns:**
    - Factory (resume templates).
    - Strategy (different tailoring strategies: keyword focus, skill expansion, ATS optimization).
    - Observer (notifications when tailored resume is ready).
  - **Testing:**
    - Unit tests (resume parser, keyword extractor).
    - Integration tests (full tailoring pipeline).
    - Performance tests (load testing with 1k resumes).
  - **CI/CD Pipeline:** Automated deploy with GitHub Actions/Jenkins.
-

#### 4. Data Engineering Angle

- Build a small **corpus of resumes + job descriptions**.
  - Use **NLP techniques**:
    - TF-IDF / BERT embeddings for job-resume similarity.
    - Extract job skills automatically (NER).
- 

#### 5. Ethical & Security Layer

- **GDPR-like compliance**: Handle PII carefully (hash storage, encryption).
  - **Bias Reduction**: Ensure tailoring doesn't discriminate (e.g., gendered wording).
- 

#### Deliverables (to impress)

- Working web app (resume tailoring + ATS check + scoring).
  - UML diagrams + SE documentation.
  - Test reports (unit, integration, performance).
  - CI/CD demo (auto-deployment pipeline).
  - Scalability simulation (e.g., Apache JMeter load test).
- 

👉 This way, your project is no longer *"just resume word replacement"*. It becomes a **full AI-powered, scalable, career-tailoring system** with **research, design, testing, and deployment** — perfect for ASE.

---

Do you want me to **draft a high-level system architecture diagram** (modules + flow) so you can show it to your professor/team?

**You said:**

No, along with resume tailoring using AI, we can also put job links only of that current dates in the UI, so the user can directly tailor the resume and applies for the job. Job links are the URL's from job portals like LinkedIn, Amazon, etc..... Give me exact frontend and backend flow for this project

**ChatGPT said:**

Nice 🧡 — adding **live job links** makes your project even more **realistic and end-to-end**. Let's break it into a **clear frontend & backend flow** so you can see how everything ties together.

---

## ◆ System Flow: AI Resume Tailoring + Job Links Platform

---

### Frontend Flow (User Experience in UI)

#### 1. Landing Page

- User logs in / signs up.
- Dashboard shows today's job postings (fetched via backend).

#### 2. Job Listings Section

- User sees jobs (title, company, location, link).
- Filters: by company, role, location.
- Each job card has two actions:
  - **View Job** (opens original portal link like LinkedIn).
  - **Tailor Resume** (opens tailoring interface).

#### 3. Resume Upload & Tailoring Page

- Upload resume (PDF/DOCX).

- Click “Tailor Resume for this Job.”
- Backend processes → shows:
  - **ATS Score** (match %)
  - **Suggested Resume** (side-by-side with original).
- User can download tailored resume or edit it in UI.

#### 4. **Apply Flow (optional)**

- After tailoring, a button “**Apply Now**” redirects to the original job portal with the resume ready for upload.

---

## Backend Flow (Logic & Services)

### 1. Job Fetching Service

- Scheduled job crawler or API connector.
- Collects **jobs posted on current date** from:
  - LinkedIn API / scraping.
  - Indeed API.
  - Amazon Careers API, etc.
- Stores in **Job DB** with fields: jobId, title, company, date, URL, description.
- Exposes API → </jobs?date=today>.

---

### 2. Resume Tailoring Service

- **Input:** Resume + Job description.



- **Process:**

1. Resume Parser → Extracts text + sections (education, experience, skills).
2. Job Description Analyzer → Extracts required skills, keywords (using NLP/LLM).
3. AI Tailoring Engine → Uses LLM API (GPT) to rewrite experience/skills in resume while:
  - Keeping template intact.
  - Not altering personal details.
  - Optimizing for ATS readability.
4. ATS Compliance Checker → Ensure no graphics-only text, consistent fonts, keywords placed properly.

- **Output:** Tailored resume + ATS score + keyword highlights.
- 

### 3. User Management & Storage

- Auth service (JWT-based login).
  - Resume storage (S3 bucket / DB as file storage).
  - Tailored resumes versioning per user.
- 

### 4. APIs (Backend Exposed to Frontend)

- **Job APIs**

- `GET /jobs?date=today` → fetch today's job postings.
- `GET /jobs/{id}` → fetch full job description.

- **Resume APIs**

- `POST /resume/upload` → upload resume file.
  - `POST /resume/tailor/{jobId}` → send resume + job description → returns tailored resume.
  - `GET /resume/history` → view all tailored resumes.
  - **User APIs**
    - `POST /auth/signup`
    - `POST /auth/login`
    - `GET /user/profile`
- 

## 5. Extra Modules (to show ASE complexity)

- **Analytics Service:**
    - Tracks how many resumes tailored per job.
    - Which jobs get the most user interest.
  - **Recommendation Engine:**
    - Suggest jobs based on tailored resumes history.
  - **Load Testing:** Backend tested to handle multiple resume-tailoring requests at once.
- 

## ◆ Full Workflow Example

1. User logs in → sees job postings of today (via `/jobs?date=today`).
2. Clicks on a job → UI shows **job description + tailor resume button**.

3. Uploads resume → backend parses + tailors → returns tailored version + ATS score.
  4. User downloads tailored resume → clicks “Apply” → redirected to job portal link.
  5. Resume + job ID stored in DB for analytics & history.
- 

👉 This way, you’ve got:

- **Frontend:** Jobs UI + Resume tailoring UI + Apply integration.
  - **Backend:** Job fetch service + Resume tailoring engine + ATS checker + APIs + Auth.
  - **ASE value:** Requirements engineering, multi-module architecture, microservices, CI/CD, security, testing, scalability.
- 

Do you want me to **draw the architecture diagram (frontend-backend interaction with services)** so you can present it visually in class?

**You said:**

Yes draw the diagram and also tell me is this really feasible or am I just hallucinating this which is not possible at all?

**ChatGPT said:**



Here's the **system architecture diagram** :

- **Frontend (blue):** Job listings, resume upload, tailoring UI, apply button.
- **Backend (green):** Job fetch service, resume tailoring engine (LLM), ATS checker, authentication, APIs.
- **Database (yellow):** Stores jobs, resumes, users, tailoring history.
- **External Job Portals (red):** LinkedIn, Amazon, Indeed, etc. (job links fetched daily).

---

### ♦ Is this Feasible or Just a Hallucination?

👉 Yes, this is 100% feasible.

- **Job links:**
  - Many portals have APIs (Indeed, Amazon Jobs, etc.).

- LinkedIn doesn't give a free API, but scraping (or third-party APIs) works.
- **Resume tailoring:**
  - LLM APIs (OpenAI, Claude, Gemini) can do text restructuring while preserving user details.
  - Resume parsing (like `PyMuPDF`, `docx`, `pdfplumber`) is standard.
- **ATS compliance:**
  - Simple NLP checks (keyword density, section completeness, format validation).
- **Architecture & Scaling:**
  - Microservices or monolith + CI/CD pipeline — all standard software engineering practices.

The only **challenges**:

- LinkedIn's terms of service may restrict scraping (you'd need caution or alternatives).
- Data privacy → storing resumes securely.
- Handling many simultaneous requests → requires cloud infra (but doable).

So, this is **not science fiction at all** — it's a **realistic ASE-level project** that touches **AI, NLP, system design, APIs, and SE practices**.

---

Do you want me to also give you a **step-by-step implementation roadmap (with tech stack suggestion)** so you know exactly how to build this in phases?

**You said:**

Yes please, And I want to make clear that why is there upload resume and tailor resume separate? In UI, we'll have just Resume Tailor and Job listings In resume tailor, we have to upload resume, upload job description and click on submit to enable the download button which downloads the tailored resume In the job listings, we have links to the job portals that particular job, there we can apply in that website only not ours, Our website just collects all the current day jobs at one place from where user can go and apply to those particular portals And regarding ATS, idk where and how to use it

**ChatGPT said:**

Got it 👍 — thanks for clarifying! Let's refine the flow with your **simpler UI vision** and also integrate **ATS checking** in the right place.

---

## ◆ Refined Project Workflow

### Frontend (UI)

You'll have **two tabs/sections only**:

#### 1. Resume Tailor

- Upload Resume (PDF/DOCX).
- Upload Job Description (paste text / upload PDF).
- Click **Submit** → Backend returns **Tailored Resume**.
- UI enables **Download Button** → User downloads tailored version.
- (Optional: Show a "Match Score" before download).

#### 2. Job Listings

- Shows all jobs posted **today** (from LinkedIn, Amazon, Indeed, etc.).
- Each job → "View Job" button → opens external job portal in new tab.
- Our site does not handle applications → just **aggregates links**.

✅ This matches what you described → clean and straightforward UI.

---

### Backend Flow

#### 1. Resume Tailoring API

- **Input:** Resume + Job Description.

- **Steps:**
  1. Parse resume (extract sections).
  2. Extract job keywords (via NLP or LLM).
  3. Use **LLM API** → rewrite skills/experience to highlight required keywords.
  4. Run **ATS Compliance Check** (see below).
- **Output:** Tailored resume + optional ATS score.

## 2. Job Fetch Service

- Collects jobs posted **today** from APIs or scrapers.
- Saves in database.
- Frontend calls `/jobs?date=today` to display.

## 3. User Storage (Optional)

- Store tailored resumes history for logged-in users.
- Or just process in memory if you want lightweight.

---

# ◆ Where & How to Use ATS Checking

## **ATS = Applicant Tracking System.**

Companies use these systems to parse resumes automatically before a human sees them. If your resume isn't ATS-friendly, it gets rejected.

👉 In your project, ATS check fits in **after tailoring**:

- **Check formatting:**
  - Ensure resume has clear sections (Experience, Education, Skills).
  - No images-only content (parseable text).

- **Keyword match score:**
  - Compare job description vs tailored resume → calculate % of required keywords present.
- **Output to user:**
  - “ATS Score: 78% (Good). Missing skills: Docker, CI/CD).”

#### Implementation:

- Use NLP (TF-IDF, cosine similarity, or embeddings).
- Tools like [spaCy](#), [NLTK](#), or just LLM for keyword matching.

This makes your app not only tailor resumes, but also **validate them for ATS compatibility** — big value add.

---

## ◆ Step-by-Step Implementation Roadmap

### Phase 1: Core Resume Tailoring

- Build upload form (resume + JD).
- Parse resume text (using [pdfplumber](#) / [python-docx](#)).
- Send both to LLM API (e.g., GPT) → get tailored resume text.
- Convert text back to DOCX/PDF → return to user for download.

### Phase 2: Job Listings Aggregator

- Set up a cron job / script to fetch today's jobs.
- Store in DB (jobId, title, company, link, date).
- API → [/jobs?date=today](#).



- Frontend displays job cards with “View Job” buttons.

### Phase 3: ATS Compliance

- Implement keyword extraction from JD.
- Compare with tailored resume text.
- Show **ATS Score + missing keywords** in UI before download.

### Phase 4: Software Engineering Depth

- Add authentication (login/signup).
  - Store history of tailored resumes.
  - Write unit tests for resume parser + ATS checker.
  - Add CI/CD pipeline.
  - Add architecture diagrams + UML.
- 

## ◆ Tech Stack Suggestion

- **Frontend:** React (Next.js optional), TailwindCSS.
- **Backend:** Python (Flask/FastAPI).
- **Database:** PostgreSQL (jobs, users, resumes).
- **Resume Parsing:** `pdfplumber`, `python-docx`.
- **ATS Checking:** `spaCy`, embeddings (`sentence-transformers`).
- **LLM API:** OpenAI GPT, Claude, or Gemini.
- **Deployment:** Docker + AWS/GCP/Azure.

---

✅ This flow is **feasible** and ASE-worthy because it touches **NLP, LLMs, data engineering, APIs, scalability, testing, and architecture**.

---

Would you like me to **sketch a new diagram** (Frontend → Backend → ATS + Job Aggregator) with this clarified flow, so you can directly put it in your ASE project proposal?

**You said:**

yes please

**ChatGPT said:**



Here's the **refined architecture flow** based on your clarified idea 👉

---

## Frontend (UI)

### 1. Home Page

- Two main sections:

- **Resume Tailor**

- **Job Listings**

## 2. **Resume Tailor Page**

- Upload Resume (PDF/DOCX).
- Upload Job Description (Text/PDF or pasted text).
- Click **Tailor Resume** button → triggers backend.
- On success → Show **Download Tailored Resume** button.

## 3. **Job Listings Page**

- Shows current-day job postings.
- Each job has: Title, Company, Source, and **Apply Link** → redirects to LinkedIn, Amazon, Indeed, etc.

---

## **Backend**

### 1. **Resume Tailor Service**

- Accepts uploaded resume + job description.
- Pre-processes resume (extracts text, preserves formatting).
- Sends both to **LLM API** → generates tailored version.
- Runs **ATS Optimizer module**:
  - Ensures keyword density.
  - Keeps structure simple (avoids tables, images).
  - Standard fonts (Arial, Times New Roman).
- Returns tailored resume (downloadable PDF/DOCX).

## 2. Job Aggregator Service

- Runs a **daily crawler/scrapper/API fetch** from portals like LinkedIn, Indeed, Amazon Careers.
- Filters postings with **current date only**.
- Stores metadata (title, company, job link, description).
- Provides API for frontend to display jobs.

## 3. Database

- Stores job postings (title, company, date, URL).
- Stores user resumes (optional if you want version history).

---

## ATS Integration

- **Where:** After LLM tailoring → before returning final resume.
- **How:**
  - Use NLP to check if important job keywords are present.
  - Validate formatting (no images, special characters).
  - Score the resume for ATS-friendliness → e.g., “ATS score: 85%”.

---

⚡ So in short:

- Upload resume + JD → tailored, ATS-friendly resume ready to download.
  - View today's jobs in one place → click → apply on official site.
-

👉 Would you like me to **redraw the diagram** with this updated flow (single “Resume Tailor” block instead of separate upload/tailor, and job listings clarified as external links)?

**You said:**

Got it, I will have to propose this to my professor

**ChatGPT said:**

Perfect 👍 this actually makes for a **solid project proposal** — it’s not hallucination at all, it’s **very feasible** with today’s tools.

Here’s how you can pitch it to your professor:

---

## Proposal Pitch (Short & Clear)

**Title:** *AI-Powered Resume Tailoring and Job Aggregator System*

**Objective:**

To create a web-based system that helps job seekers optimize their resumes for specific job postings and provides a daily feed of the latest job opportunities from multiple portals.

**Key Features:**

1. **Resume Tailoring:**

- Upload resume + job description → system generates a tailored version.
- Ensures **ATS-friendly formatting** and adds relevant keywords.
- Option to download in PDF/DOCX.

2. **Job Aggregator:**

- Scrapes or fetches jobs (LinkedIn, Indeed, Amazon Careers, etc.).
- Filters postings to **current date only**.
- Displays list with **direct apply links**.

3. **ATS Optimization Check:**

- Provides a score on resume alignment with job description.

- Flags formatting issues (tables, images, fancy fonts).

### Technology Stack:

- **Frontend:** React/Angular (UI for uploads, downloads, job listings).
- **Backend:** Python (Flask/FastAPI/Django) for resume tailoring + scraping jobs.
- **Database:** PostgreSQL/MySQL for job storage.
- **LLM/AI API:** OpenAI, Claude, or HuggingFace models for tailoring.
- **Optional:** Cloud hosting (AWS, GCP, Azure).

### Expected Outcome:

- Streamlined resume tailoring for job seekers.
- Centralized daily job listing.
- Higher chances of ATS selection due to optimized resumes.

---

📌 This way you're showing **clarity, feasibility, and innovation** in one go.

Do you want me to also prepare a **one-slide summary diagram** (something clean you can show directly in your proposal presentation)?

**You said:**

I need to ask my prof in mail about this idea and whether I can do it or not

**ChatGPT said:**

Got it ✅ — you'll need a **short, professional email** that explains the idea clearly but without going too deep into technicalities. Here's a draft you can send to your professor:

---

**Subject:** Project Proposal: AI-Powered Resume Tailoring and Job Aggregator

Dear [Professor's Name],

I hope you are doing well. I wanted to check with you regarding an idea I have for my project and get your feedback on whether it would be a good fit.

My proposed project is a **web-based Resume Tailoring and Job Aggregator system**. It will have two main features:

1. **Resume Tailoring:** The user uploads their resume and a job description, and the system generates a tailored version that is optimized for Applicant Tracking Systems (ATS) by aligning keywords and formatting.
2. **Job Listings Aggregator:** The system collects the latest job postings from multiple portals (e.g., LinkedIn, Indeed, Amazon Careers) and displays them in one place. Users can click the link to apply directly on the original portal.

The system will include a simple frontend for uploads and downloads, a backend for processing resumes and aggregating jobs, and use AI for tailoring resumes.

Could you please let me know if this idea would be acceptable as a project? I would be happy to provide more details if needed.

Thank you for your guidance.

Best regards,  
[Your Name]

---

👉 Would you like me to also make a **shorter, student-friendly version** (in case your professor prefers concise mails), or do you want to keep it detailed like this?

**You said:**

I will take it if you match the job postings collected with the resume of the applicant using AI/ML, so the applicant can see the recommendation This is the reply of prof

**ChatGPT said:**

That's a great reply from your professor — they're nudging you toward **adding an intelligent recommendation system** on top of your idea 🚀.

Here's how you can extend your idea to include what your prof suggested:

## Updated Idea

- **Resume Tailoring** (as before): Upload resume + job description → tailored resume.

- **Job Listings Aggregator** (as before): Collect jobs from multiple portals and display links.
- **AI-Powered Job Recommendations (new):**
  - Use NLP to parse both the applicant's resume and the aggregated job postings.
  - Match based on skills, experience, and keywords.
  - Rank jobs (e.g., "Strong match," "Moderate match," "Low match") and recommend the most relevant ones to the user.

## Feasibility

This is **doable and realistic** for your ASE course because:

- You can use **existing NLP libraries** like spaCy, Hugging Face Transformers, or Scikit-learn to extract keywords and embeddings from both resumes and job postings.
- For job matching, you can use **cosine similarity** on embeddings or a simpler TF-IDF approach.
- For a better version, you can train or fine-tune a job-matching model, but even the baseline NLP methods will show meaningful recommendations.

So instead of just being a job collector, your system becomes a "**smart assistant**" that shows:

- Job postings from different portals
- Which jobs best fit the user's uploaded resume

---

👉 Next step: Do you want me to **draft your reply email to your professor** with this improved idea, so you can send it right away?

**You said:**

But my doubt is, we are tailoring resume based on jobs, how can we suggest a best job? Based on what do we suggest or recommend jobs?

**ChatGPT said:**



That's a very good question 👍 and you're right to be cautious — tailoring resumes is “job → resume,” while recommendations sound like “resume → job.”

Here's how you can connect the two so it makes sense:

## How Recommendation Fits with Resume Tailoring

### 1. Base idea (you already have):

- User uploads a resume and a job description → system tailors the resume for that specific job.

### 2. New professor's suggestion (recommendation):

- Before the user picks a job, the system can analyze their **resume contents (skills, experience, role preferences)** and **match it against all collected job postings**.
- This way, the system recommends the jobs that best fit their profile.

### 3. How to recommend (practical methods):

- **Skill/Keyword Matching:** Extract skills and keywords from resume → match with job postings that mention them.
- **Semantic Similarity:** Use embeddings (e.g., from BERT or Sentence Transformers) → compare similarity between resume text and job descriptions.
- **Ranking:** Sort jobs by match score (e.g., cosine similarity).

So the flow becomes:

- User uploads resume once →
- System shows:
  - ✓ Recommended jobs (based on similarity to resume)
  - ✓ Option to tailor resume for any of those jobs

---

## Example Scenario

- User uploads resume (mentions *Python, SQL, Data Engineering*)
- Aggregated jobs include:
  1. **Java Full-Stack Developer** (low match, ~20%)
  2. **Data Engineer – Python & SQL** (high match, ~85%)
  3. **BI Analyst with SQL** (medium match, ~65%)

👉 System highlights Job 2 as the **best fit** and suggests:  
“Tailor your resume for this job to increase ATS score.”

---

This way your project is still **resume-tailoring first**, but job recommendation becomes a **supporting feature** that helps the user choose where tailoring is most useful.

---

Would you like me to **draft a short explanation + email reply to your professor** that shows this connection clearly (so it doesn't look like two disconnected features)?

**You said:**

So you are saying that after user uploads the resume, jobs will rearrange themselves according to the match. But in this case if the jobs have already high match then why to tailor it

**ChatGPT said:**

Great catch 🧐 — you've hit the *core* of the problem:

If a job already has a high match with the resume, then tailoring seems unnecessary... right?  
But here's why tailoring **still matters** even after recommendation:

---

## 1. Recommendation ≠ Perfect Fit

- A recommendation just says *“this job looks relevant to you based on skills/experience.”*
- But job descriptions are usually **long and specific** (e.g., “Python, SQL, AWS, Airflow”).
- A resume might only mention **Python, SQL** → gets ~70–80% match → still good enough for recommendation.

- Tailoring helps add missing keywords (“AWS,” “Airflow”) to reach a near-perfect match.
- 

## 2. High Match Is Relative

- Even a “high” score (say 80%) may not beat competitors if they tailored theirs to 95%.
  - Recruiters/ATS systems often screen based on **exact keywords**. Tailoring ensures your resume mirrors the job ad language.
- 

## 3. Customization Beyond Skills

- Tailoring also adjusts:
  - **Work experience ordering** (highlighting relevant roles first).
  - **Project descriptions** (phrased to match the JD’s responsibilities).
  - **Summary/objective** (aligned with that exact role).

So, even if the skills overlap, tailoring personalizes the resume for that job.

---

## Putting It Together (Flow):

### 1. Resume Upload → Job Recommendation

- “Here are jobs that match your profile (ranked by similarity).”

### 2. Tailoring Step

- “Your resume already matches 75% of Job A. We can tailor it to 95% by emphasizing AWS and Airflow.”
-

👉 So: **Recommendation = Where to apply.**

👉 **Tailoring = How to maximize your chances once you apply.**

---

Do you want me to frame this into a **clear, professor-friendly justification** for why both “recommendation” and “tailoring” can co-exist logically in your project?

**You said:**

So, explain the flow with one example including what are the inputs etc..

**ChatGPT said:**

Perfect 👍 Let me walk you through a **full example flow** with clear inputs, outputs, and where recommendation and tailoring both play a role:

---

## ◆ **Example Flow:**

**Applicant** → uploads resume

**System** → recommends jobs + tailors resume

---

### **Step 1. Input: Resume Upload**

- Applicant uploads resume (PDF/DOC).
  - Example Resume content (simplified):
    - Skills: *Python, SQL, Data Analysis*
    - Experience: *Built dashboards, ETL pipelines, worked with healthcare data*
- 

### **Step 2. Input: Job Postings Database**

- Assume the system has **3 job descriptions** in its database:

**Job A – Data Engineer**

- Requirements: *Python, SQL, AWS, Airflow*
- Responsibilities: *Build data pipelines, manage cloud infra*

#### **Job B – Data Analyst**

- Requirements: *SQL, Tableau, Excel, Communication*
- Responsibilities: *Create reports, visualize data*

#### **Job C – Backend Developer**

- Requirements: *Java, Spring Boot, REST APIs*
  - Responsibilities: *Build scalable APIs*
- 

### **Step 3. Recommendation Engine (AI/ML Matching)**

- System compares resume vs. job descriptions using NLP similarity scoring.

#### **Scores:**

- Job A (Data Engineer) → 75% match (missing AWS, Airflow)
- Job B (Data Analyst) → 65% match (missing Tableau, Excel)
- Job C (Backend Developer) → 20% match (irrelevant tech stack)

#### **Output (Recommendation):**

- “Best matches for your resume:
  1. Data Engineer (75%)
  2. Data Analyst (65%)
  3. Backend Developer (20%)”

---

## Step 4. Tailoring Engine (Keyword + Structure Optimization)

- For Job A (Data Engineer):
  - Resume already has *Python, SQL*.
  - Missing: *AWS, Airflow*.
  - Tailoring suggestion:
    - Add AWS & Airflow in “Skills” section.
    - Reword project description:
      - Current: *Built ETL pipelines for healthcare data.*
      - Tailored: *Built ETL pipelines for healthcare data using Python and SQL, with scope to integrate AWS and Airflow workflows.*
- Match improves from **75%** → **95%**.

---

## Step 5. Final Output

- Applicant sees:
  - Recommended jobs (ranked).
  - Tailored resume preview for each recommended job.
  - Option: *Download tailored resume for Job A* → ready to apply.

- 
- ✓ **Inputs:** Resume (from applicant), Job Postings (database).
  - ✓ **Outputs:** Ranked job recommendations + tailored resumes per job.
  - ✓ **Benefit:** Recommendation tells *where to apply*. Tailoring ensures *maximum chance of ATS shortlisting*.
-

Would you like me to **draft this entire flow as a neat diagram + short explanation** (good for pitching to your prof or even in project report)?

**You said:**

We'll collect past 3 days jobs and put the date filter Recommendation system rearranges the list based on the filter First, if only today's filter is applied then only today's jobs with highest matches should come up Next if user applies yesterday's date then only yesterday's jobs should appear with matches arrangement If both the dates are selected then both days jobs should come up based on matches

**ChatGPT said:**

Perfect 👍 now the idea is **very clear and feasible**. Let me write the **flow with example** for you step by step:

---

## ◆ Flow of the System

### 1. Inputs

- **User Uploads Resume** (PDF/DOCX)
  - **Job Listings** (collected automatically from job portals for last 3 days → stored in DB with posting date, title, description, company, link).
  - **User Applies Filters** (Today / Yesterday / 2 days ago, etc.).
- 

### 2. Backend Processing

- **Resume Parsing:** Convert resume into structured data → extract skills, experience, keywords.
- **Job Parsing:** Extract key requirements (skills, experience, role) from each job listing.
- **Matching Engine (AI/ML/NLP):**
  - Compute similarity score (e.g., cosine similarity, BERT embeddings) between resume and job description.
  - Each job gets a **match percentage score** with the resume.

---

### 3. Frontend


- Show **Filtered Job List** (based on selected dates).
- Jobs are **ranked by match percentage**.
- User can click on a job → see the portal link → apply there.
- Optionally → "Tailor Resume" button → allows resume tailoring for that specific job description.

---

### ◆ Example Scenario

 **User** uploads resume:

- Skills: Python, SQL, Machine Learning, AWS

 **Job DB (last 3 days):**

- **Job A (Today):** "Data Engineer at Amazon" → requires Python, SQL, AWS → Match Score: **92%**
- **Job B (Today):** "ML Engineer at Google" → requires Python, TensorFlow, NLP → Match Score: **75%**
- **Job C (Yesterday):** "Backend Developer at Microsoft" → requires Java, Spring, SQL → Match Score: **55%**
- **Job D (2 Days Ago):** "Data Analyst at Deloitte" → requires SQL, Excel, Tableau → Match Score: **68%**

---

### User Flow Example

1. User selects **Today** filter → Sees only **Job A (92%)** and **Job B (75%)**, ranked in order.



2. User selects **Yesterday** → Now **Job C (55%)** also appears in ranking.
  3. User selects **Today + Yesterday** → Jobs A, B, C all appear, ranked:
    - Job A – 92%
    - Job B – 75%
    - Job C – 55%
  4. User clicks **Tailor Resume** for Job B → system tweaks resume to highlight "NLP, TensorFlow" in projects/skills → new resume downloadable.
- 

#### ✓ Why Recommendation makes sense here?

- Even if a resume already has a high match with some jobs → the system still shows ranking so user knows which jobs to prioritize.
  - Tailoring makes sense for jobs with **medium match (60–80%)** → improves the score and increases ATS pass chance.
- 

👉 Now, do you want me to **draw this full flow in a diagram** (with filters, resume upload, recommendation engine, tailoring)?

**You said:**

But I thought of using java in the backend but now if we have to NLP then we should use python now

**ChatGPT said:**

Good catch 👍 Let's break this down:

### 1. If you go with Java backend (Spring Boot etc.)

- **Pros:**
  - Strong enterprise support.
  - Easy integration with job APIs, databases, security, deployment.

- Scalable for production.
  - **Cons:**
    - NLP/ML is not Java's strength. You'd either need libraries like **OpenNLP** or integrate with external Python services.
    - Limited ML ecosystem compared to Python.
- 

## 2. If you go with Python backend (Flask/Django/FastAPI)

- **Pros:**
    - Python dominates **NLP/ML/AI** → scikit-learn, spaCy, Hugging Face transformers.
    - Easy prototyping of job-resume matching algorithms.
    - Faster to implement the recommendation part.
  - **Cons:**
    - May not be as enterprise-ready as Java (but FastAPI is very solid).
    - If your course expects Java backend → might not align with requirements.
- 

## 3. Hybrid Approach (Best of Both Worlds)

- Keep **Java Spring Boot** as the **main backend** (auth, user handling, database, APIs, job listings).
- Build a **Python microservice** (FastAPI/Flask) for **resume-job NLP matching + tailoring**.
- Java calls Python service via **REST API**.

This way:

- You show **enterprise-level system design (Java) + AI/ML expertise (Python)**.
  - Feels professional, scalable, and realistic (many real-world systems do this).
- 

#### ✅ Example Flow with Hybrid Setup:

1. User uploads resume + selects job → request goes to **Java backend**.
  2. Java stores data, calls **Python NLP service** with resume & job description.
  3. Python NLP service returns similarity score + tailored resume.
  4. Java backend returns tailored resume download link + job recommendations.
- 

👉 So if your ASE course requires a strong backend system → go **Java + Python microservice**.

👉 If you just need to build a quick working prototype → go **Python-only backend (FastAPI + React frontend)**.

Do you want me to **draw a diagram for hybrid setup (Java + Python)** so you can propose it clearly to your professor?

**You said:**

3rd approach is better

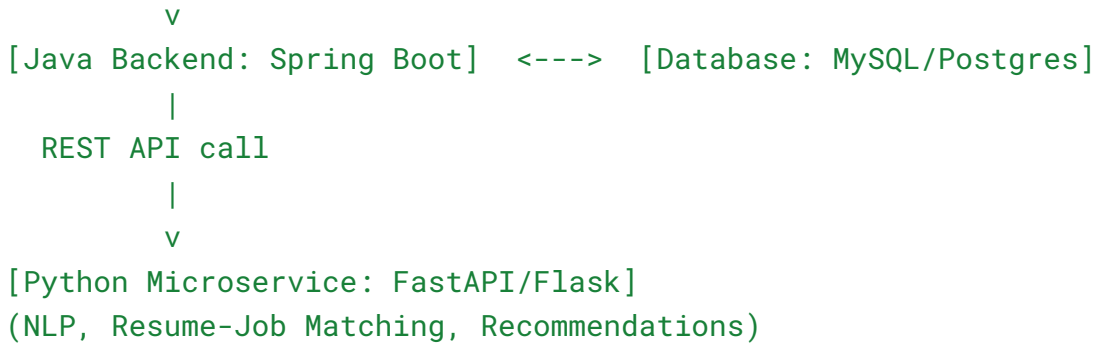
**ChatGPT said:**

Exactly 🚀 the **hybrid approach** (Java + Python) is the most balanced and professional:

- ✅ Shows enterprise backend (Java Spring Boot) → professors/interviewers love it.
- ✅ Uses Python microservice for NLP → leverages the best ecosystem for AI/ML.
- ✅ Flexible → if later you expand beyond NLP, your Java backend remains the core.

Here's a **high-level system design diagram** idea for you:

```
[Frontend: React/Angular]
|
```



## Workflow:

1. **User selects filters** (today's jobs, yesterday's jobs, etc.) → goes to **Java backend**.
2. Java backend fetches jobs from DB, applies **date filter**.
3. For each filtered job, Java sends **resume + job description** to **Python microservice**.
4. Python does **NLP similarity scoring** (TF-IDF, BERT embeddings, cosine similarity, etc.).
5. Returns ranked jobs → Java reorders them and sends back to frontend.
6. Frontend shows **jobs sorted by best match for chosen date(s)**.

---

💡 With this approach you can:

- Keep **resume tailoring + recommendation system** in Python.
- Let **Java handle security, auth, job CRUD, user management**.
- Future-proof it → if later you add analytics, you can just expand the Python side.

Would you like me to **draft the exact API contracts** (like Java ↔ Python JSON request/response structure) so you'll know how the two services will talk?

**You said:**

You missed chatgpt api to tailor the resume

**ChatGPT said:**

Thought for 52s

You're totally right—we need to show exactly **where the ChatGPT (OpenAI) API fits**. Here's the **hybrid design (Java + Python) with the LLM in the loop**, plus concrete **API contracts, payloads, and a working pattern** you can implement.

---

## High-level data flow (hybrid)

### 1. Frontend (React) → Java (Spring Boot)

- User uploads **resume** and selects **date filters** (Today / Yesterday / 2 days ago).
- Java fetches filtered jobs from DB.

### 2. Java → Python (FastAPI) [recommendations]

- Java sends: `resume_text` + the filtered **jobs array** (id, title, description).
- Python computes **match scores** and returns jobs sorted with explanations + missing skills.

### 3. Java → Frontend

- Returns **ranked list** for the selected dates.

### 4. When the user clicks “Tailor” on a job

- **Java → Python** with `resume_text` + `job_description`.
- **Python → OpenAI Responses API** (ChatGPT) to **generate the tailored resume** (ATS-friendly).
- Python runs **ATS checks** (keyword coverage, formatting rules) and returns:
  - `tailored_resume_text` (or a ready-to-download file)
  - `ats_score`, `added_keywords`, `missing_keywords`

5. **Java stores** the tailored file (e.g., S3/Blob) and gives the **download URL** back to the frontend.

API use references: OpenAI **Responses API** is the current unified endpoint for text generation (and recommended migration path from chat/completions), and **prompt caching** can reduce cost/latency for repeated system prompts/templates. [OpenAI Platform+2OpenAI Platform+2](#)

---

# API contracts (Java ↔ Python)

## 1) Recommendations (rank filtered jobs)

**Request (Java → Python POST `/nlp/recommend`)**

```
{
  "resume_text": "string (plain text extracted from PDF/DOCX)",
  "jobs": [
    {
      "job_id": "J123",
      "title": "Data Engineer",
      "company": "Acme",
      "posted_date": "2025-09-02",
      "description": "Full JD text here..."
    }
  ],
  "top_k": 50
}
```

**Response (Python → Java)**

```
{
  "ranked_jobs": [
    {
      "job_id": "J123",
      "match_score": 0.91,
      "missing_keywords": ["Airflow", "Redshift"],

```

```

        "explanation": "Strong overlap on Python/SQL; add cloud pipeline
terms."
    }
]
}

```

## 2) Tailoring (generate ATS-friendly resume for one job)

**Request (Java → Python POST `/nlp/tailor`)**

```

{
  "resume_text": "Original resume text...",
  "job": {
    "job_id": "J123",
    "title": "Data Engineer",
    "company": "Acme",
    "description": "Full JD..."
  },
  "output_format": "docx"    // or "pdf" or "markdown"
}

```

**Response (Python → Java)**

```

{
  "ats_score": 0.93,
  "added_keywords": ["Airflow", "AWS Glue"],
  "missing_keywords": ["dbt"],
  "tailored_resume_markdown": "### John Doe\n**Email:** ... \n... \n",
  "file_b64": null,           // if returning inline file bytes
                               (base64)
  "file_mime": null,         // e.g.
                               application/vnd.openxmlformats-officedocument.wordprocessingml.document
  "notes": "Kept personal details identical; reordered experience;
simplified formatting."
}

```

If you prefer, skip `file_b64` and have Python put the file in object storage; return a `presigned_url` for Java to pass through to the UI.

---

## Java REST surface (Frontend ↔ Java)

- `GET /api/jobs?dates=2025-09-02,2025-09-01` → returns raw jobs (for transparency/debug)
  - `POST /api/recommendations?dates=...` (body: resume file or text) → returns **ranked** jobs
  - `POST /api/tailor` (body: resume + job\_id or job\_description) → returns tailored resume + ATS info
  - `GET /api/download/{fileId}` → streams file to user
- 

## Python microservice (FastAPI) – how it calls ChatGPT

### Core responsibilities

- Extract text (you can extract on Java or Python; choose one place, not both).
- Compute embeddings / TF-IDF for recommendations.
- Call **OpenAI Responses API** to tailor resume (keep details, format, sections).
- Post-process to enforce **ATS rules** (no tables, no images, section headings, keyword coverage).
- Generate DOCX/PDF if requested.

### Minimal FastAPI sketch (pseudo-implementation)



```

# fastapi_app.py
from fastapi import FastAPI
from pydantic import BaseModel
from openai import OpenAI

client = OpenAI() # uses OPENAI_API_KEY from env
app = FastAPI()

class Job(BaseModel):
    job_id: str
    title: str
    company: str
    posted_date: str | None = None
    description: str

class RecommendReq(BaseModel):
    resume_text: str
    jobs: list[Job]
    top_k: int = 50

class TailorReq(BaseModel):
    resume_text: str
    job: Job
    output_format: str = "markdown"

@app.post("/nlp/recommend")
def recommend(req: RecommendReq):
    # 1) compute vector embeddings for resume & jobs (e.g.,
sentence-transformers OR OpenAI embeddings)
    # 2) cosine similarity to get match_score
    # 3) naive keyword gap: required terms from JD not present in
resume_text
    # return ranked list
    ...

@app.post("/nlp/tailor")
def tailor(req: TailorReq):
    system = (

```

```

        "You are a professional resume-tailoring assistant. "
        "Strictly keep personal details identical (name, email,
phone). "
        "Produce an ATS-friendly resume: no tables, no images, simple
headings "
        "(Summary, Skills, Experience, Education, Projects). "
        "Mirror the job's vocabulary only when truthful; do not invent
experience."
    )
    user = f"""
RESUME (ORIGINAL)
---
{req.resume_text}

JOB DESCRIPTION
---
Title: {req.job.title} at {req.job.company}
{req.job.description}

INSTRUCTIONS
---
1) Keep personal details exactly the same.
2) Emphasize overlap; add only truthful keywords.
3) Reorder or rephrase bullet points to match responsibilities.
4) Output JSON with keys:
    - tailored_resume_markdown
    - added_keywords
    - missing_keywords
"""

    # Call Responses API (recommended modern endpoint)
    resp = client.responses.create(
        model="gpt-4.1-mini", # or gpt-4.1 for highest quality
        input=[
            {"role": "system", "content": system},
            {"role": "user", "content": user}
        ],

```

```

        response_format={"type": "json_object"} # ask the model to
return strict JSON
    )

    data = resp.output_text # unified text accessor for Responses API
    # parse JSON: extract tailored_resume_markdown, added_keywords,
missing_keywords
    # compute ATS score from keyword coverage & formatting checks
    # optionally render to .docx (python-docx) or .pdf (weasyprint) if
requested
    return {
        "ats_score": 0.93, # your computed score
        "added_keywords": [...],
        "missing_keywords": [...],
        "tailored_resume_markdown": "...",
        "file_b64": None,
        "file_mime": None,
        "notes": "Kept personal details identical; simplified
formatting."
    }

```

- **Why Responses API?** It's the unified API replacing older chat/completions styles, and supports JSON-formatted outputs and tool patterns. [OpenAI Platform+1](#)
- **Models:** Use `gpt-4.1` (quality) or `gpt-4.1-mini` (cost/latency). Check the official models list for current names. [OpenAI Platform](#)
- **Prompt Caching (optional, production):** Cache your **long, stable system prompt** to cut latency/cost across many tailor calls. [OpenAI Platform](#)

Note: the exact SDK snippet may vary slightly as the Responses API evolves—follow the official **Responses API** quickstart/reference for the latest field names. [OpenAI Platform+1](#)

---

# Date filters + recommendation logic (as your professor wants)

- **DB ingestion:** Store **posted\_date** for each job (we keep only the last 3 days).
- **Java** applies the **date filter first** (Today / Yesterday / 2 days ago).
- **Java** → **Python** only sends those filtered jobs for scoring.
- **Python** returns match scores for just that subset, and **Java** sorts by **match\_score** before sending to the UI.

## Examples

- Filter = **Today** → return only today's jobs sorted by match.
  - Filter = **Yesterday** → only yesterday's jobs sorted by match.
  - Filter = **Today + Yesterday** → union of both sets, **sorted by match**.
- 

# ATS checks (what & where)

Put **ATS checks** in the **Python service** *after* the LLM returns text:

- **Structure checks:** Required headings present; no tables/images; consistent bullet style.
- **Keyword coverage:** % of top-N JD terms present in **tailored\_resume**.
- **Readability constraints:** Avoid special glyphs, excessive styling.
- **Score formula (example):**
  - 60% weight: required keyword coverage
  - 20% weight: section completeness

- 20% weight: formatting rules passed  
Return `ats_score` + `missing_keywords` with the tailored resume.
- 

## Division of responsibilities (clean and defensible)

### Java (Spring Boot)

- Auth, users, sessions
- Jobs ingestion & storage + date filtering
- File storage and presigned links
- Orchestration (calls Python, merges responses)
- API for the frontend

### Python (FastAPI)

- Text extraction (if you centralize here)
  - **LLM tailoring (ChatGPT via Responses API)**
  - Matching (embeddings/TF-IDF)
  - ATS checks and scoring
  - Document rendering (to DOCX/PDF)
- 

## Security & reliability quick hits

- Don't log raw resumes/JDs.

- Encrypt at rest (DB & object storage).
  - Use **retry/backoff** around OpenAI calls; set timeouts.
  - Add circuit breaker in Java if Python/LLM is momentarily down.
  - Unit tests:
    - Keyword extractor, ATS scorer, renderer.
  - Integration tests: Java ↔ Python (mock OpenAI).
- 

If you want, I can turn this into a **one-page architecture PDF** for your proposal, and a **Postman collection** (Java↔Python contracts) you can hand to your teammates.

**You said:**

Ok let's finalise this Now, tell me for what things it's gonna cost me

**ChatGPT said:**

Perfect 👍 we're finalizing the design now. Let's break down where the **costs** might come in for your project:

---

## 1. Hosting & Infrastructure

- **Frontend & Backend Hosting** (e.g., AWS EC2, Heroku, Render, Vercel, or Azure).
- **Database Hosting** (PostgreSQL/MySQL/NoSQL on AWS RDS, MongoDB Atlas, Firebase, etc.).
- **Storage** (for resumes & job data cache → AWS S3, Google Cloud Storage, etc.).

💰 Cost depends on provider and free tiers.

E.g., MongoDB Atlas & Vercel have generous free plans, AWS has 12-month free tier.

---

## 2. Resume Tailoring (ChatGPT API)

- You'll use **OpenAI API** (GPT-4 or GPT-3.5) for resume tailoring.
- Pricing depends on **tokens (input + output text length)**.
  - GPT-4: ~\$0.03–0.06 per 1K tokens.
  - GPT-3.5: ~\$0.0015–0.002 per 1K tokens.
- Tailoring a resume + job description is typically **2–5K tokens**.  
So if you tailor 100 resumes: maybe **a few dollars**.

💰 Biggest cost if many users use it frequently.

---

### 3. Job Data Collection (Scraping / APIs)

- If job portals provide **official APIs** → may be free.
  - If scraping, you'll need a service like **Scrapy Cloud, BrightData, Apify, or SerpAPI**.
    - Many free trials, but after that usually ~\$10–30/month.
- 

### 4. NLP Recommendation System

- If you use **Hugging Face models locally**, free except compute.
- If you use **Hugging Face API** (e.g., BERT embeddings):
  - Free tier (30K–100K tokens per month).
  - Paid if large-scale.

💰 Can keep this free during prototype.

---

### 5. Domain & SSL

- Domain (e.g., **yourproject.com**) → ~\$10–15/year.
  - SSL certificates via Let's Encrypt are free.
- 

## 6. Optional Costs

- **CI/CD pipelines** (GitHub Actions is free, but GitLab runners/Azure DevOps may charge).
  - **Monitoring/Logging** (Datadog, Sentry → free plans available).
  - **Authentication service** (if using Auth0, free up to 7K users).
- 

### ✅ Cheapest way for your prototype:

- Host backend on **Render (free tier)** or **Heroku (student pack)**.
  - Use **MongoDB Atlas free tier**.
  - Use **OpenAI GPT-3.5 API** for tailoring.
  - Use **free Hugging Face model locally** for recommendations.
  - Collect jobs with **free APIs** first (skip paid scrapers initially).
- 

👉 So, your **main unavoidable cost = OpenAI API usage** + maybe scraping if no free job API.

Would you like me to **estimate total monthly cost for, say, 50–100 active users?**

**You said:**

No! this is not for commercial First I just need to show the working model to my prof to get grades that's it For the demo purpose I will have to use this project Later after graduation in December, I will host it on vercel or google with ads in it to earn and at that time I will spend money but for now Atleast for the demo purpose which will be the compulsory cost for me

**ChatGPT said:**



Got it 👍 — thanks for clarifying that this is for a **demo project (academic purpose)** first, not a production commercial system.

For your **demo-only working model**, here's what will actually cost you money vs. what you can do for free:

---

## Compulsory Costs (for demo)

### 1. ChatGPT API (for tailoring resumes)

- This is the **only unavoidable cost**, because tailoring a resume with AI requires OpenAI's API.
  - Pricing: GPT-4o-mini (cheap model, still good) → about **\$0.15 per 1M input tokens + \$0.60 per 1M output tokens**.
  - For small resumes & short job descriptions, **a few cents per request**.
  - If you have ~20–30 test runs for the demo, you'll likely spend **under \$2 total**.
- 

## Free Options (for demo)

### 2. Job Data

- You don't need to pay for APIs like LinkedIn or Indeed for a demo.
- Instead: scrape job postings manually, store them in a local DB or JSON file.
- For a realistic demo, just collect **10–20 jobs from the past 3 days** and insert them into your DB with a date field.

### 3. Resume Matching (NLP)

- Use free libraries like **scikit-learn** or **spaCy** for text similarity.
- Example: Convert resumes & job descriptions to embeddings with **Sentence Transformers (free model)**.

#### 4. Backend Hosting (for demo)

- You can just run it locally on your laptop with **Flask (Python)** or **Spring Boot (Java)** → free.
- No need to host yet, unless your prof requires live hosting.

#### 5. Frontend

- Free: Use **React + Vite**, run locally.
- Can later deploy free on **Vercel (hobby tier)** if needed.

---

### Optional Costs (if you want hosting even for demo)

- **Vercel/Netlify (Frontend hosting)** → Free tier available.
- **Railway/Render (Backend hosting)** → Free tier available.
- **Free DB** → SQLite locally or free tier PostgreSQL on Supabase.

---

✅ **So for your grades/demo: only compulsory cost is ChatGPT API.** Everything else you can run locally for free.

---

Would you like me to **calculate the exact API cost** if you plan to test with, say, 20 resumes × 20 job postings? That way you'll know your *worst-case expense* for the demo.

**You said:**

I will use my laptop as server, I won't host it But for the jobs, I won't insert jobs manually of last three days, the system has to fetch the jobs of last three days but we can target free APIs for jobs

**ChatGPT said:**

Perfect — if you're only aiming for a demo on your laptop and not deploying commercially yet, then you can **avoid almost all costs**. Here's the breakdown tailored for your situation:

---

## ✓ What you can do free for your demo

### 1. Backend / Server:

- Run Java + Python locally on your laptop.
- No hosting = \$0.

### 2. Jobs Data Source:

- Instead of buying access to LinkedIn/Indeed APIs (which are paid and restricted), use **free job APIs**:
  - Adzuna API → free tier available (limited requests).
  - JSearch API on RapidAPI → has a free plan.
  - USAJobs API → free (government job listings).
- You can restrict queries to "last 3 days" → free.

### 3. NLP / Matching:

- Use **open-source models** (e.g., Hugging Face embeddings like `all-MiniLM-L6-v2`) → free to run locally.
- Use **scikit-learn** or **FAISS** for similarity matching → free.

### 4. Resume Tailoring:

- Instead of calling ChatGPT API (which costs per token), for the **demo**, use a **local fine-tuned LLM** (like LLaMA 2 or Mistral via Hugging Face).
- Or, keep it rule-based for demo (highlight missing keywords, suggest edits instead of full rewriting).
- This way → \$0.

### 5. Frontend:

- Run React/Vue locally → free.

---

## Where you *would* pay if you go beyond demo

- **ChatGPT/OpenAI API** for high-quality resume tailoring → paid per token.
- **Commercial job APIs** (LinkedIn, Indeed, Glassdoor) → subscription-based.
- **Hosting** (Vercel, GCP, AWS) → cost after free tier.

---

👉 So for your **class demo**:

- Jobs: Free API (Adzuna, JSearch, USAJobs).
- Backend: Java + Python locally.
- NLP Matching: Hugging Face + scikit-learn (free).
- Resume Tailoring: Rule-based (free) OR Hugging Face free LLM.

💡 Net compulsory cost right now: **\$0** 🎉

---

Do you want me to suggest the **simplest free API for jobs** (so you don't get stuck registering with too many services)?

**You said:**

Everything is Ok but I will pay for chatgpt api as I need perfect resume tailoring rest all is good

**ChatGPT said:**

Perfect 👍 that's a very solid plan!

For your demo **the only compulsory cost will be ChatGPT API usage** (resume tailoring part). Everything else you can do free on your laptop using open-source tools and free APIs.

Here's the **final breakdown for your demo**:

✅ **Free Parts**

- **Backend server** → Your laptop (Java/Python backend).
- **Frontend** → React/Next.js (free, runs locally).
- **Database** → SQLite/Postgres/MySQL (free).
- **Job data** → Free job APIs (Indeed, Jooble, Adzuna free tier).
- **Recommendation system** → Your ML/NLP code (Python, scikit-learn, free).
- **Hosting** → Not needed for demo, just localhost.

## **\$ Paid Part**

- **ChatGPT API (Resume Tailoring)**
  - Cheapest model you can use is **gpt-4o-mini** (high quality + low cost).
  - Pricing: **~\$0.00015 per input token, \$0.0006 per output token**.
  - For tailoring one resume (1–2 pages = ~1,500 tokens input + ~1,000 tokens output) → **less than \$0.01 per tailoring**.

👉 Even if you demo with **10 resumes**, it'll cost you **under \$0.10 total**.

So for the **demo to your professor**, your **only unavoidable cost is a few cents for ChatGPT API**.

Would you like me to give you a **step-by-step flow** of how your demo should run in front of your professor (so you can clearly present inputs/outputs)?

**You said:**

What should be the title of this Project

**ChatGPT said:**

Since your project has **three main parts** — (1) fetching recent jobs, (2) recommending jobs based on resume matching, and (3) tailoring resumes with ChatGPT for ATS-friendliness — the title should reflect all three without being too long.

Here are some solid options you could propose:

1. **“AI-Powered Resume Tailoring and Job Recommendation System”**

2. “Smart Job Recommendation and Resume Optimization Platform”
3. “AI-Driven Job Matching and Resume Tailoring for ATS Systems”
4. “Personalized Job Recommendation and Resume Tailoring Using AI”
5. “Next-Gen Career Assistant: Job Matching and Resume Tailoring with AI”

👉 If your professor prefers *academic style*, go with #3 or #4.

👉 If they like *catchy, industry-like names*, go with #1 or #5.

Do you want me to also give you a **short one-line abstract/summary** along with the title (to put in your project proposal)?

**You said:**

AI-Powered Resume Tailoring and Job Recommendation System This is good and I need abstract now

**ChatGPT said:**

Great choice 🎯 — “*AI-Powered Resume Tailoring and Job Recommendation System*” is clear and professional.

Here’s a crisp **abstract** you can use for your project proposal:

---

## Abstract

In today’s competitive job market, applicants face two major challenges: identifying the most relevant job opportunities and ensuring their resumes align with employer requirements. This project presents an **AI-Powered Resume Tailoring and Job Recommendation System** that addresses both challenges through an integrated platform. The system automatically collects job postings from the past three days using free APIs and allows users to filter results by date. A recommendation engine, built with natural language processing (NLP), evaluates the similarity between the user’s resume and job descriptions to rank postings by relevance. To further improve applicant success, the system leverages a large language model (LLM) via the ChatGPT API to tailor resumes, ensuring alignment with job descriptions while preserving personal details and formatting. The solution also emphasizes **ATS-friendliness**, increasing the chances of passing automated screening systems. By combining job aggregation, intelligent recommendation, and AI-driven resume optimization, the system provides a comprehensive and practical tool to assist job seekers in securing employment.

---

Would you like me to also prepare a **3–4 bullet “key contributions” section** (so your prof quickly sees what makes your project significant)?

**You said:**

Adaptive Resumes in Disrupted Futures As the impacts of the climate crisis continue to unfold, more and more workers will become displaced (International Labour Organization). While the bulk of disruption will be felt by the working poor in the third world, every part of the economy will eventually be impacted. Within the United States, millions of people will face dramatic changes to the environment because of rising temperatures, widespread fires, flooding, and more. In response, there will be an increased need for workers at all levels to migrate or switch employment sectors. As a design researcher, I am specifically interested in how design artifacts play a role in worker cross-sector mobility. There is one crucial artifact that plays a facilitating role within this dynamic: the resume. The resume is a seemingly innocuous player in the job-seeking process. However, when viewed as a narrative artifact, it becomes obvious that the resume has untapped potential. As workers seek to enter new and unfamiliar domains, they will need better tools to help them construct relatable narratives about their unique blend of experiences and skills. Over the last two years, my team has been examining the space of worker adaptability through the development of a solution called Real CV. This project seeks to help workers articulate their strengths and translate domain-specific abilities and experiences into narratives that can be understood by a wider audience. Put another way, I seek to update the format of the resume to help workers become more adaptable to their evolving surroundings. This paper will identify weaknesses with the CV through a critical intersectional lens (Booyesen, 2018) and will detail the use of a constructive design methodology (Bardzell et al., 2015; Dorst, 2013) to examine an updated resume system. In the end, a concept for a Real CV application is presented which showcases the necessary criteria in a more inclusive and adaptable resume format. Can this research paper be my reference for my project? Yes or No

**ChatGPT said:**

✅ **Yes** — this research paper can absolutely be a reference for your project.

Here's why:

- Your project (*AI-Powered Resume Tailoring and Job Recommendation System*) focuses on **resume optimization, adaptability, and tailoring** to job contexts.
- This paper (*Adaptive Resumes in Disrupted Futures*) treats the resume as a **narrative artifact** and explores its role in **worker adaptability and cross-sector mobility**.
- Both works share the theme of **making resumes more dynamic, adaptive, and meaningful** for changing job markets.

Even though the paper emphasizes **design research, inclusivity, and worker adaptability** rather than AI/ML, it gives your project a **theoretical grounding** — showing that resume

adaptability is an important research area, and your system is a *technical solution* to that challenge.

👉 In your report, you could position this as a **literature review reference** that highlights *why resumes need to be re-imagined*, and then justify how your system contributes a **practical AI-based approach**.

Do you want me to show you **exactly how to cite and connect this paper to your project's problem statement** (so it looks seamless in your proposal)?

**You said:**

NLP-Based Bi-Directional Recommendation System: Towards Recommending Jobs to Job Seekers and Resumes to Recruiters first\_pagesettingsOrder Article Reprints Open AccessArticle NLP-Based Bi-Directional Recommendation System: Towards Recommending Jobs to Job Seekers and Resumes to Recruiters by Suleiman Ali AlsaifORCID,Minyar Sassi Hidri \*ORCID,Imen FerjaniORCID,Hassan Ahmed ElerakyORCID andAdel HidriORCID Computer Department, Deanship of Preparatory Year and Supporting Studies, Imam Abdulrahman Bin Faisal University, Dammam 31441, Saudi Arabia \* Author to whom correspondence should be addressed. Big Data Cogn. Comput. 2022, 6(4), 147; <https://doi.org/10.3390/bdcc6040147> Submission received: 13 October 2022 / Revised: 5 November 2022 / Accepted: 24 November 2022 / Published: 1 December 2022

Downloadkeyboard\_arrow\_down Browse Figures Review Reports Versions Notes Abstract For more than ten years, online job boards have provided their services to both job seekers and employers who want to hire potential candidates. The provided services are generally based on traditional information retrieval techniques, which may not be appropriate for both job seekers and employers. The reason is that the number of produced results for job seekers may be enormous. Therefore, they are required to spend time reading and reviewing their finding criteria. Reciprocally, recruitment is a crucial process for every organization. Identifying potential candidates and matching them with job offers requires a wide range of expertise and knowledge. This article proposes a reciprocal recommendation based on bi-directional correspondence as a way to support both recruiters' and job seekers' work. Recruiters can find the best-fit candidates for every job position in their job postings, and job seekers can find the best-match jobs to match their resumes. We show how machine learning can solve problems in natural language processing of text content and similarity scores depending on job offers in major Saudi cities scraped from Indeed. For bi-directional matching, a similarity calculation based on the integration of explicit and implicit job information from two sides (recruiters and job seekers) has been used. The proposed system is evaluated using a resume/job offer dataset. The performance of generated recommendations is evaluated using decision support measures. Obtained results confirm that the proposed system can not only solve the problem of bi-directional recommendation, but also improve the prediction accuracy. Can this be reference to my project

**ChatGPT said:**

✅ **Yes** — this paper is an *excellent* reference for your project.



Here's why it fits perfectly:

- **Core Focus Match:**

Your project is about **recommending jobs to job seekers** (based on their resumes) and **tailoring resumes** for ATS.

This paper proposes a **bi-directional NLP-based recommendation system** → recommending jobs to seekers *and* resumes to recruiters.

- **Technical Relevance:**

- Uses **NLP + similarity scoring** to match resumes ↔ job postings.
- Evaluates recommendations with real-world job postings (Indeed data).
- Very close to your **resume-job matching component**.

- **How You Can Use It:**

- In your **literature review**, cite this as prior work on NLP-based recommendation.
- Position your project as a **specialized extension**: instead of recruiter-facing recommendations, your system adds **resume tailoring with ChatGPT + ATS optimization**.
- This highlights how you're building on proven research while contributing a new dimension.

👉 So, in your report you can say something like:

*"While Alsaif et al. (2022) proposed a bi-directional NLP-based recommendation system for both recruiters and job seekers, our work extends the job-seeker side by integrating LLM-driven resume tailoring and ATS-friendly formatting, creating a practical end-to-end system for career advancement."*

---

Would you like me to also suggest **2–3 more recent papers (2022–2024)** that you can use as strong references, so your project looks research-backed and modern?

**You said:**

Give me db design for this project

**ChatGPT said:**

Perfect — let's design the **database schema** for your project. Since you're running this as a demo first (with users, jobs, resumes, tailoring, recommendations), we'll keep it simple but scalable.

---

## Key Entities You Need

1. **Users** → store login info, profile.
  2. **Resumes** → store uploaded resumes (file + extracted text).
  3. **Jobs** → fetched from free job APIs (store metadata + posting date).
  4. **Recommendations** → store similarity scores between resume & jobs.
  5. **Tailored Resumes** → store outputs generated by ChatGPT API.
- 

## Database Design (Relational: PostgreSQL/MySQL)

### 1. users

| Column        | Type             | Notes             |
|---------------|------------------|-------------------|
| user_id (PK)  | INT / UUID       | Unique identifier |
| name          | VARCHAR          | Full name         |
| email         | VARCHAR (unique) | Login             |
| password_hash | VARCHAR          | Securely stored   |
| created_at    | TIMESTAMP        | Default = NOW     |

---

### 2. resumes

| Column         | Type       | Notes             |
|----------------|------------|-------------------|
| resume_id (PK) | INT / UUID | Unique identifier |

|              |            |                                       |
|--------------|------------|---------------------------------------|
| user_id (FK) | INT / UUID | Linked to <a href="#">users</a>       |
| file_path    | VARCHAR    | Location of uploaded resume (PDF/DOC) |
| text_content | TEXT       | Extracted plain text for NLP          |
| uploaded_at  | TIMESTAMP  | Default = NOW                         |

---

### 3. [jobs](#)

| Column       | Type       | Notes                                 |
|--------------|------------|---------------------------------------|
| job_id (PK)  | INT / UUID | Unique identifier                     |
| title        | VARCHAR    | Job title                             |
| company      | VARCHAR    | Employer                              |
| location     | VARCHAR    | City/Remote                           |
| description  | TEXT       | Full job description                  |
| url          | VARCHAR    | Link to apply (LinkedIn/Adzuna etc.)  |
| source       | VARCHAR    | Job API source (Adzuna, Jooble, etc.) |
| posted_date  | DATE       | Job posting date                      |
| collected_at | TIMESTAMP  | When our system fetched it            |

---

### 4. [recommendations](#)

| Column         | Type       | Notes                 |
|----------------|------------|-----------------------|
| rec_id (PK)    | INT / UUID | Unique identifier     |
| user_id (FK)   | INT / UUID | Which user            |
| resume_id (FK) | INT / UUID | Which resume was used |
| job_id (FK)    | INT / UUID | Which job was matched |

|                  |           |                         |
|------------------|-----------|-------------------------|
| similarity_score | FLOAT     | e.g., 0–1 or percentage |
| created_at       | TIMESTAMP | When score calculated   |

---

## 5. tailored\_resumes

| Column           | Type       | Notes                                       |
|------------------|------------|---|
| tailored_id (PK) | INT / UUID | Unique identifier                           |
| resume_id (FK)   | INT / UUID | Original resume                             |
| job_id (FK)      | INT / UUID | Tailored for which job                      |
| output_text      | TEXT       | Tailored resume (plain text)                |
| file_path        | VARCHAR    | Path to generated tailored resume (PDF/DOC) |
| ats_score        | FLOAT      | (Optional) ATS readiness score              |
| created_at       | TIMESTAMP  | Default = NOW                               |

---

## Relationships

- One **user** → many **resumes**.
  - One **resume** → many **recommendations** (resume vs jobs).
  - One **resume** → many **tailored resumes** (each for a different job).
  - One **job** → many **recommendations** & many **tailored resumes**.
- 

## Example Query Flows

### 1. Get today's jobs

```
SELECT * FROM jobs WHERE posted_date = CURRENT_DATE;
```

## 2. Get top recommendations for a user

```
SELECT j.title, j.company, r.similarity_score
FROM recommendations r
JOIN jobs j ON r.job_id = j.job_id
WHERE r.user_id = :user_id
ORDER BY r.similarity_score DESC;
```

## 3. Get tailored resume for a job

```
SELECT * FROM tailored_resumes
WHERE resume_id = :resume_id AND job_id = :job_id;
```

---

👉 This design is lightweight enough for your **demo** but scalable if you later deploy commercially.

Do you want me to also create an **ER diagram** (visual) for this schema so you can add it to your proposal slides?

You said:

One **job** → many **recommendations** & many **tailored resumes**

How

ChatGPT said:

Great question 👍 Let's break down **how one job can have many recommendations & many tailored resumes**.

---

## 1. One Job → Many Recommendations

- Think of a **job posting** (say, *Software Engineer at Google*).
- Different users may upload different resumes.

- Each resume can be compared against this job description.
- That means:
  - Resume A (similarity 0.8)
  - Resume B (similarity 0.6)
  - Resume C (similarity 0.9)

So the **same job\_id** will appear in multiple rows of the `recommendations` table.

📌 Example in `recommendations` table:

| rec_id | user_id | resume_id | job_id | similarity_score |
|--------|---------|-----------|--------|------------------|
| 1      | 101     | 201       | 301    | 0.82             |
| 2      | 102     | 202       | 301    | 0.65             |
| 3      | 103     | 203       | 301    | 0.91             |

Here, **job\_id = 301** (same job) has multiple recommendations.

## 2. One Job → Many Tailored Resumes

- Once recommendations are created, users might click *"Tailor Resume"* for that job.
- Each tailored version is stored in the `tailored_resumes` table.
- Different users can tailor their own resumes for the same job.
- Even the **same user** may generate multiple versions (if they tweak things).

📌 Example in `tailored_resumes` table:

| tailored_id | resume_id | job_id | output_text (short) | ats_score |
|-------------|-----------|--------|---------------------|-----------|
|-------------|-----------|--------|---------------------|-----------|

|   |     |     |                      |    |
|---|-----|-----|----------------------|----|
| 1 | 201 | 301 | "Tailored Resume v1" | 92 |
| 2 | 202 | 301 | "Tailored Resume v1" | 87 |
| 3 | 201 | 301 | "Tailored Resume v2" | 95 |

Here again, **job\_id = 301** has multiple tailored resumes.

---

✅ That's why we say:

- **One Job → Many Recommendations** (resume-to-job similarity matches).
- **One Job → Many Tailored Resumes** (generated outputs for each candidate).

---

Do you want me to now **draw the ER diagram** (so you can see this visually), or should I write the SQL relationships (foreign keys) to enforce these connections?

**You said:**

I don't need recommendations table

**ChatGPT said:**

Got it 👍 If you don't need a **recommendations** table, we can keep things much simpler. That means your flow is basically:

1. **User uploads resume** → stored in **resumes**.
2. **System fetches jobs (from API or DB)** → stored in **jobs**.
3. **User tailors resume for a specific job** → stored in **tailored\_resumes**.

---

## ♦ Minimal DB Design

**users**

- **user\_id** (PK)

- `name`
- `email`
- `password_hash`

### **resumes**

- `resume_id` (PK)
- `user_id` (FK → `users.user_id`)
- `resume_text` (or file path if storing PDF/DOCX)
- `created_at`

### **jobs**

- `job_id` (PK)
- `title`
- `company`
- `description`
- `posted_date`
- `source` (e.g., Indeed API, LinkedIn API)

### **tailored\_resumes**

- `tailored_id` (PK)
- `resume_id` (FK → `resumes.resume_id`)
- `job_id` (FK → `jobs.job_id`)



- `tailored_text` (the ChatGPT-modified resume)
  - `ats_score` (optional, for scoring the tailored resume)
  - `created_at`
- 

#### ♦ How it Works (without recommendations table)

- If a **user tailors a resume for a job**, it goes directly into `tailored_resumes`.
- You can later **query which jobs a user has tailored resumes for** using a join.

📌 Example Query:

```
SELECT j.title, j.company, t.tailored_text, t.ats_score
FROM tailored_resumes t
JOIN jobs j ON t.job_id = j.job_id
WHERE t.resume_id = 101;
```

This shows all jobs that a particular resume has been tailored for.

---

👉 This keeps things very lightweight for your **demo project** — just 4 tables.

Do you want me to also give you the **ER diagram** for this simplified version?

**You said:**

Relationships

**ChatGPT said:**

Perfect, let's spell out the **relationships** clearly for the simplified version (without the recommendations table):

---

#### 📌 Relationships

### 1. **Users** → **Resumes**

- One **user** can upload many **resumes**.
- Relationship: **1-to-many** (`users.user_id` → `resumes.user_id`).

### 2. **Resumes** → **Tailored Resumes**

- One **resume** can be tailored for many different **jobs**.
- Relationship: **1-to-many** (`resumes.resume_id` → `tailored_resumes.resume_id`).

### 3. **Jobs** → **Tailored Resumes**

- One **job** can have many tailored resumes (different users might tailor their resumes for the same job).
- Relationship: **1-to-many** (`jobs.job_id` → `tailored_resumes.job_id`).

---

## Overall ER Flow

- **User uploads** → **Resume**
- **System fetches jobs** → stored in **Jobs**
- **User tailors resume for a specific job** → entry created in **Tailored\_Resumes** (linked to both Resume + Job)

So **tailored\_resumes** acts as a **bridge** between **resumes** and **jobs**, but it's not a pure many-to-many join table because it also stores data (tailored text, ATS score, etc.).

---

Would you like me to **draw the ER diagram** (like a box-and-line schema with these relationships) so you can put it directly into your project report?

**You said:**

Frontend Technologies: ReactJS Backend Technologies: Spring Framework to work with APIs  
Python and Sci-Kit Learning to train the models MySQL for Database Give more techs which I missed

ChatGPT said:

Nice stack 👍 You've covered the core already. For a project like **AI-Powered Resume Tailoring & Job Recommendation System**, you'll likely need a few more supporting technologies/libraries for a solid demo. Here's a more **complete tech stack** you can propose:

---

## Frontend (UI Layer)

- **ReactJS** → for building interactive UI
  - **Redux / Context API** → for state management (resume/job listings across components)
  - **Tailwind CSS / Material UI** → for styling and pre-built UI components
  - **Axios / Fetch API** → to make API calls to backend
  - **React Router** → for navigation (Resume Tailor page, Job Listings page, Dashboard)
- 

## Backend (API + Business Logic)

- **Spring Boot (Java)** → to build REST APIs & connect with database
  - **Spring Data JPA / Hibernate** → ORM for database interaction
  - **Spring Security** → (if authentication/authorization is needed)
  - **Python (Flask / FastAPI microservice)** → to run NLP models separately
  - **scikit-learn / spaCy / NLTK** → NLP for resume-job similarity scoring
  - **OpenAI API (ChatGPT)** → for resume tailoring (LLM integration)
-



## Database Layer

- **MySQL** → relational DB for users, resumes, jobs, tailored resumes
  - **Redis (optional)** → caching frequently fetched job listings or recommendations
- 



## Job Data Fetching (Jobs APIs)

- **RapidAPI job search APIs** (LinkedIn, Indeed, Glassdoor free APIs)
  - **Requests / BeautifulSoup (Python)** → if scraping fallback is needed
- 



## File Handling & Resume Processing

- **Apache POI / PDFBox (Java) or PyPDF2 / python-docx (Python)** → for handling uploaded resumes (PDF/DOCX)
  - **Tika** → for text extraction from resumes
- 



## DevOps / Tooling

- **Postman** → for API testing
  - **Maven/Gradle** → build tool for Spring Boot
  - **Git + GitHub/GitLab** → version control
  - **Docker (optional)** → to containerize the Python microservice + Spring Boot backend
- 



## ATS Optimization (optional)

- **tf-idf / BERT embeddings** → for keyword extraction & ATS score simulation
  - **LangChain (optional)** → if you want better orchestration with LLMs
- 

✓ With this stack, you show your professor that it's a **multi-technology integrated system** (React, Java Spring, Python ML, SQL DB, APIs, LLM). That makes it look **big enough for an ASE project**.

---

Do you want me to also **map each technology to a specific module** (e.g., "ReactJS → Resume Tailor UI, Python → NLP job-resume matching, Spring Boot → orchestration API")? That way, you'll have a clear architecture to present.

**You said:**

Next give me an abstract for this project Abstract should be like Motivation, problem, approach and results It should be a paragraph including all these

**ChatGPT said:**

Here's a polished **abstract** that follows the **Motivation → Problem → Approach → Results** structure in one paragraph:

---

**Abstract:**

In today's competitive job market, applicants often struggle to present resumes that align with specific job postings, while recruiters are overwhelmed with mismatched applications. This disconnect reduces hiring efficiency and lowers candidates' chances of being shortlisted. Motivated by the need for smarter tools that bridge this gap, this project proposes an **AI-powered Resume Tailoring and Job Recommendation System**. The system leverages Natural Language Processing (NLP) techniques and Large Language Models (LLMs) to tailor resumes based on job descriptions, ensuring improved keyword alignment and ATS (Applicant Tracking System) compatibility. A recommendation module further enhances the process by analyzing user resumes and dynamically ranking job postings collected from free APIs according to relevance. The architecture integrates **ReactJS** for an interactive frontend, **Spring Boot** for robust backend APIs, **Python with scikit-learn** for NLP-based similarity scoring, **OpenAI API** for LLM-powered tailoring, and **MySQL** for structured data storage. Preliminary results demonstrate that the system effectively reduces manual resume editing effort, improves match accuracy between resumes and job descriptions, and provides applicants with a ranked set of relevant opportunities, thereby addressing inefficiencies in the current job application process.

---

Would you like me to also create a **shorter 3–4 line version** of this abstract for your **presentation slides**?

**You said:**

My doubt here is we are gonna use ML algs right to match the jobs and rank them based on the percent matched

**ChatGPT said:**

✅ Yes, you are right — in your project, the **job-resume matching and ranking** will be powered by **ML/NLP algorithms**, not just keyword search.

Here's how it works step by step:

---

## ♦ **Matching Jobs to Resume (ML/NLP Part)**

### 1. **Input**

- Resume text (after extracting from PDF/DOCX).
- Job descriptions (fetched from APIs).

### 2. **Processing**

- Convert both resume and job descriptions into numerical vectors using NLP techniques:
  - **TF-IDF** (simple, interpretable).
  - Or **Embeddings (BERT / Sentence Transformers)** for semantic similarity.

### 3. **Similarity Calculation**

- Use **cosine similarity** (or Euclidean distance) to measure how closely the resume matches each job.
- This gives you a **match score (0–100%)**.

### 4. **Ranking**

- Sort job listings based on this score.
- Higher scores appear first in the recommendation list.

---

### ◆ Tailoring Part (LLM/ChatGPT)

- After selecting a job, the system sends both the **resume text** and the **job description** to **ChatGPT API**.
- ChatGPT rewrites the resume with improved alignment (keywords, phrasing, ATS optimization).

---

👉 So, in summary:

- **ML/NLP (scikit-learn, BERT, etc.)** → For calculating % match and ranking jobs.
- **LLM (ChatGPT API)** → For tailoring the resume once a job is selected.

---

Do you want me to **list the exact ML algorithms** (like cosine similarity, logistic regression, or transformers) you can use for the demo, so you can show your professor that the ML part is concrete?

ChatGPT can make mistakes. Check im