Resume–JD Matching System — Full Technical & Product Specification

This document explains the complete flow, architecture, libraries, scoring logic, parsing logic, UI behaviour, and writing-quality checks for a resume–JD matching system. The system is built in **React (frontend)** and **Node.js (backend)** with **no LLM** usage in v1. Everything is deterministic, rule-based, explainable, and easy to scale.

---

1. What the System Does

A user pastes a Job Description and uploads a Resume (PDF/DOCX). The system extracts structured information from both, compares them, and returns: - Fit Score - ATS-Likelihood Score - Writing/Grammar Score - Missing Skills, Keywords, Experience - Resume bullet improvements - Tailored summary suggestions - Highlighted matching and missing items

Everything is generated through lightweight NLP + rule-based logic.

---

2. High-Level System Flow

1. **User Input** → JD text + Resume file
2. **Frontend Validation** → file type, size, quick checks
3. **Upload to Backend** → via API (multipart)
4. **Parsing Worker (Node)**:
5. PDF/DOCX text extraction
6. OCR fallback
7. Section detection
8. Skill extraction
9. **JD Parser** → extract required skills, preferred skills, years, title, keywords
10. **Resume Parser** → extract summary, experience, skills, education
11. **Grammar & Spelling Checks** → rule-based
12. **Scoring Engine** → Fit, ATS, Writing Score
13. **Suggestions Engine** → missing items + improved bullets
14. **Response back to frontend**
15. **Frontend UI** → highlight matches/missing + results page

---

3. Technology Choices (React + Node, No LLM)

These are the exact libraries for each component.

3.1 PDF/DOCX Text Extraction (Node)

PDF

- **pdf-parse** → clean, fast text extraction for most PDFs
- **pdfjs-dist** → optional if positional data is needed

DOCX

- **mammoth** → DOCX → HTML → clean text
- **docx** → plain-text extraction

OCR Fallback (Scanned PDFs)

- **tesseract.js** Used only when pdf-parse returns near-empty or low-confidence output.

3.2 Resume Section Detection

- Pure **regex + rule-based logic**
- Optional helpers:
- **compromise** (lightweight NLP)
- **natural** (tokenization, stemming)

3.3 Skill Extraction & Fuzzy Matching

- **string-similarity** for fuzzy matching
- **fast-fuzzy** for faster comparisons
- Internal **skills taxonomy JSON** stored in the repo
- Synonym mapping JSON (K8s → Kubernetes)

3.4 JD Parsing (Node)

Everything rule-based: - Regex patterns for: “required”, “must have”, “good to have”, “3+ years”, commas lists - **compromise** for noun-phrases - Title normalization via internal JSON mapping

---

3.5 Grammar, Spelling & Style Checks (Node, No LLM)

Spelling

- **nodehun** (Hunspell dictionaries)
- or **simple-spellchecker**

Grammar (Rule-Based)

- POS tagging using **wink-pos-tagger**
- Custom grammar rules:
- subject–verb mismatch
- incorrect tense patterns
- repeated words
- sentence fragments

Style

Custom checks: - Bullets longer than X words - Bullets missing metrics - Bullets starting with weak verbs (Responsible for…) - Passive voice detection (POS + regex) - First-person pronouns detection (“I”, “my”, “we”)

3.6 Scoring Engine (Node)

Pure deterministic JS logic.

Fit Score Weights

- Skill match: 50%
- Core skill coverage: 20%
- Experience relevance: 15%
- Keyword coverage: 10%
- Formatting readability: 5%

ATS Score Weights

- Keyword match: 50%
- Parseability: 25%
- Format cleanliness: 15%
- Length/density: 10%

Writing Score Weights

- Spelling accuracy: 30%
- Grammar correctness: 30%
- Bullet quality: 25%
- Readability: 15%

3.7 Backend (Node)

- **Express.js** → API server
- **Multer** → file uploads
- **BullMQ** or **RabbitMQ** → task queue

- **AWS SDK (S3)** → temp file storage
- **PostgreSQL** → DB
- **Redis** → caching + queue

3.8 Frontend (React)

- **React** + **Vite/Next.js**
- **TailwindCSS** → styling
- **react-dropzone** → file upload UI
- **react-query / SWR** → API data management
- **diff-match-patch** → highlight match/miss
- **react-highlight-words** → highlighting

---

4. JD Parsing — Detailed Logic

JD text is converted into structured data: - title_hint - seniority_range - required_skills[] - preferred_skills[] - years_experience - responsibilities[] - keywords[]

4.1 How required/preferred skills are detected

Regex lines containing: - "Required", "Must have", "Mandatory", "You must" - Bullets directly under "Requirements" - Skills inside brackets or comma-separated lists

Preferred skills from: - "Nice to have", "Preferably", "Bonus"

4.2 Experience extraction

Patterns: - `3+ years` - `5 years minimum` - `3-5 years`

---

5. Resume Parsing — Detailed Logic

Extract the following: - Summary - Experience entries - Skill list - Education - Projects (optional)

5.1 Section detection

Based on heading keywords: - Experience - Work History - Summary - Skills - Education

Using regex + backup via NLP noun-chunking.

5.2 Bullet extraction

- Detect lines starting with -, •, *, or indentation

5.3 Duration calculation

- Extract start/end dates
- Normalize into months of experience

5.4 Formatting flags

- Columns detection via repeated position patterns (if pdfjs-dist used)
- Images detection → if PDF has no text blocks for specific regions

---

6. Spelling, Grammar & Style Checks

6.1 Spelling

All words checked via Hunspell dictionary.

6.2 Grammar

Rule-based checks: - subject–verb agreement - verb tense consistency - repeated words - overly long sentences

6.3 Style

Custom: - No metrics - Weak action verbs - Bullets too long - Passive voice

Each issue includes location + suggested fix.

---

7. Scoring Engine (Full Explanation)

7.1 Fit Score Components

- Skill match = JD skills that appear in resume
- Core skill coverage = required JD skills found
- Experience relevance = years match + domain keyword match
- Keyword coverage = resume mentions JD keywords
- Formatting penalty = columns, images, tables

7.2 ATS Score Components

- Keyword match
- ATS parseability → no images, no tables, no columns
- Format cleanliness → paragraph size, font issues
- Length/density

7.3 Writing Score Components

- Spellcheck score (error count)
- Grammar issues (count + severity)
- Bullet quality (action verbs, metrics)
- Readability

---

8. Suggestion Engine

Suggestions are rule-based: - If a core skill missing → suggest adding under Skills + a bullet template - If experience insufficient → suggest stronger action verbs + context - If grammar/spelling errors → corrected strings - If long bullets → rewrite template

Also provide a JD-tailored summary generated via templates.

---

9. API Endpoints

POST /analyze

Multipart input: - jd_text - resume_file

Returns: job_id

GET /status/:job_id

Returns full payload: - scores - structured JD & resume - missing items - suggestions - spell/grammar/style issues

---

10. UI/UX Flow

1. Landing: paste JD + upload resume
2. Parsing preview: show extracted sections
3. Results dashboard:

4. Fit / ATS / Writing scores
5. Missing skills + explanations
6. Bullet improvement suggestions
7. Highlighted matches/missing tokens
8. Export improved content

---

11. Storage, Security & Privacy

- Files stored temporarily in S3
- TTL auto-delete (default 24h)
- HTTPS everywhere
- Malware scanning for uploads
- GDPR delete endpoint

---

12. Testing Strategy

- Unit tests for regex, date parser, skill extraction
- Integration tests for resume parsing with 50+ sample resumes
- End-to-end JD+resume → scoring tests
- Accuracy benchmarking vs recruiter-labelled dataset

---

13. Development Roadmap

Phase 1 (MVP)

- JD parser
- Resume parser
- Fit/ATS/Writing score
- Spelling/grammar/style rules
- Suggestions
- Basic UI + results

Phase 2

- Inline resume editor
- PDF export
- User accounts + history
- Multiple resumes per JD

Phase 3

- Improved skill taxonomy

- Chrome extension
- Optional LLM-based fine rewriting

If you want, I can also generate: - The API code structure in Node - The exact folder architecture for backend - The skills taxonomy starter list - Example regex patterns used in parsing