Software Architecture and Design Specification

Project: Dynamic Resume Analyzer

Version: 1.0

Authors:

Vishwambhara RH	SRN: PES1UG23CS700
Prerana M P	SRN: PES1UG23CS903
Vineet Anil Sharma	SRN: PES1UG23CS690
Vrishank N A	SRN: PES1UG23CS705

Date: 13-09-2025

Status: Draft

Revision History

Version	Date	Author	Change Summary
1.0	10-09-2025	Vrishank	Introduction, Scope, Stakeholders, and Goals & Constraints sections.
1.0	10-09-2025	Prerana	Component Diagram, Descriptions, Architecture Pattern, and Technology Stack.
1.0	11-09-2025	Vishwambhara	Documented the Risks, Traceability, Security Architecture, and UML Sequence Diagrams.

1.0	12-09-2025	Vineet	API Design, Error Handling, UX Design, and Appendices.

Approvals

Role	Name	Signature/Date
QA Lead	Vishwambhara R H	PES1UG23CS700/ 13-09-2025
Test Engineer	Vrishank N A	PES1UG23CS705/ 13-09-2025
Developer	Prerana M P	PES1UG23CS903/ 13-09-2025
Product Owner	Vineet Anil Sharma	PES1UG23CS690/ 13-09-2025

1. Introduction

1.1 Purpose

This document specifies the architecture and design of the Dynamic Resume Analyzer project.

1.2 Scope

This document covers the high-level architecture and detailed design of the Dynamic Resume Analyzer project, focusing on the core functionalities of resume parsing and validation. It outlines the system's components, their interactions, and the design decisions made to meet the requirements of version 1.0.

1.3 Audience

The intended audience includes developers, quality assurance (QA) engineers, security auditors, instructors, and maintenance teams who need to understand the system's design and implementation details.

1.4 Definitions

- Applicant Tracking System (ATS): Software for managing and filtering job applications.

- JWT: JSON Web Token, a standard for securely transmitting authentication data.
- Natural Language Processing (NLP): AI techniques for understanding and processing human language.
- Optical Character Recognition (OCR): Technology to convert images of text into machine-encoded text.
- Regular Expression (Regex): A sequence of characters defining a search pattern for text processing.
- TLS: Transport Layer Security: A cryptographic protocol that ensures secure communication over a network by encrypting data, verifying integrity, and authenticating parties (commonly used in HTTPS for secure web browsing).
- STRIDE: Security threat modeling (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege).
- WCAG: Web Content Accessibility Guidelines.
- MFA: Multi-factor Authentication: A security process that requires users to provide two or more verification factors (e.g., password + OTP + biometric) to gain access to a system.
- SQL: Structured Query Language: A standardized programming language used to manage and manipulate relational databases. It allows creating, reading, updating, and deleting data (CRUD operations), as well as defining and controlling database structures.

2. Document Overview

2.1 How to use this document

This document provides architectural deliverables, including a high-level overview of the system, a breakdown of its components, and a discussion of key design decisions. It is intended to guide the development and testing of the Dynamic Resume Analyzer by providing a clear blueprint of the system's design.

2.2 Related Documents

SRS: Dynamic Resume Analyzer v1.0
Software Test Plan (STP) - Resume Parsing & Validation
Design Specifications v1.0
Data Protection/GDPR Standards

3. Architecture

3.1 Goals & Constraints

Goals: The system aims for high performance, reliability, and security. Key goals include:

- Performance: Processing resumes (≤ 3 pages) in ≤ 10 seconds and handling 50 concurrent requests.
- Reliability: Achieving 99.5% monthly uptime.
- Usability: Balancing positive and critical feedback, minimizing cognitive overload, and complying with WCAG 2.1 AA accessibility standards.
- Security: Ensuring confidentiality and integrity of resumes, preventing unauthorized access, and maintaining secure audit logs.

Constraints: The system must operate under several constraints:

- Performance: Resume processing time is capped at ≤ 10 seconds for files up to 3 pages.
- Supported Formats: Only .pdf, .docx, and .txt file formats are allowed for uploads.
- Data Handling: Uploaded resumes must be automatically deleted after analysis to ensure privacy and security.
- Security: Compliance with specific security requirements, including TLS 1.2+ encryption, MFA for admin accounts, session timeouts, and hashed passwords.

3.2 Stakeholders & Concerns

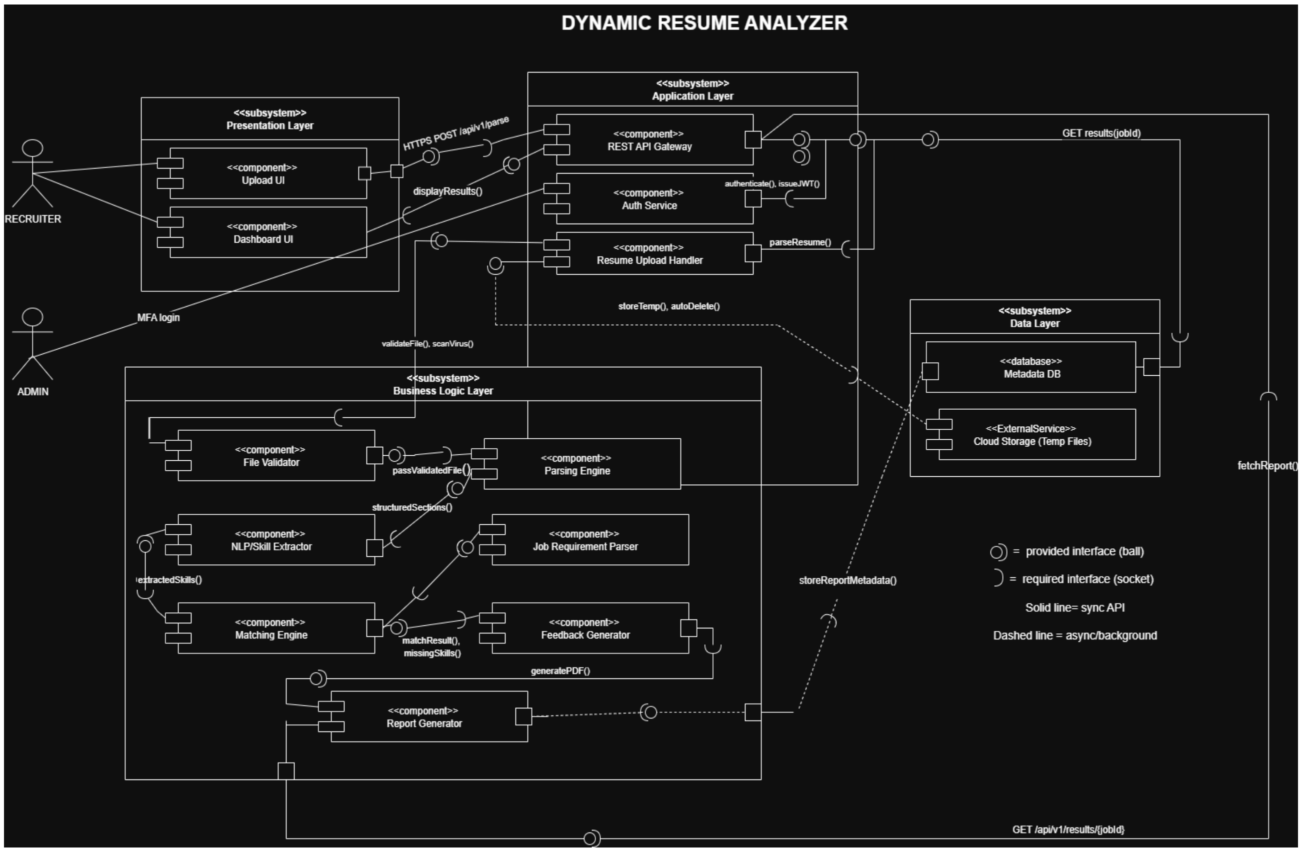
User/Recruiter: Their primary concerns are the accuracy of resume parsing, the quality and usefulness of the generated feedback and reports, and user privacy. They also value features like batch analysis and resume comparison with job descriptions.

Privacy/Security Stakeholder: This stakeholder is concerned with the anonymization of sensitive personal data (email, phone) in reports and the secure handling and auto-deletion of uploaded resumes post-analysis. They are also responsible for ensuring compliance with data protection standards.

Developer/QA Team: Their concerns include the modularity and maintainability of the system, the stability of the build, and the availability of a representative test environment and data.

Product Owner: This stakeholder's main concern is the successful delivery of the core features and the overall readiness of the product for release. They are responsible for approving test results and providing sign-off.

3.3 Component (UML) Diagram


DYNAMIC RESUME ANALYZER
<<subsystem>> Presentation Layer
<<component>> Upload UI
<<component>> Dashboard UI
RECRUITER
ADMIN
MFA login
HTTPS POST /api/v1/parse
displayResults()
<<subsystem>> Application Layer
<<component>> REST API Gateway
<<component>> Auth Service
<<component>> Resume Upload Handler
authenticate(), issueJWT()
parseResume()
storeTemp(), autoDelete()
validateFile(), scanVirus()
GET results(jobId)
<<subsystem>> Data Layer
<<database>> Metadata DB
<<ExternalService>> Cloud Storage (Temp Files)
<<subsystem>> Business Logic Layer
<<component>> File Validator
<<component>> Parsing Engine
passValidatedFile()
structuredSections()
<<component>> NLP/Skill Extractor
<<component>> Job Requirement Parser
extractedSkills()
<<component>> Matching Engine
<<component>> Feedback Generator
matchResult(), missingSkills()
generatePDF()
<<component>> Report Generator
storeReportMetadata()
fetchReport()
GET /api/v1/results/{jobId}
= provided interface (ball)
= required interface (socket)
Solid line= sync API
Dashed line = async/background

3.4 Component Descriptions

Upload UI

Web interface (React) where recruiters upload resumes and job descriptions. Supports drag-and-drop and real-time progress updates.

Dashboard UI

Displays parsed results, missing skills, quality score, and downloadable reports. Provides batch processing overview.

REST API Gateway

Central entry point for frontend requests. Handles authentication, request validation, and routes requests to backend micro-modules.

Auth Service

Handles user login, JWT token management, MFA for admin accounts, and session timeouts.

Resume Upload Handler
Manages file uploads (PDF, DOCX, TXT). Enforces format, file size, and virus scanning. Sends valid files to parsing engine.

File Validator
Ensures files are readable, not corrupted, and conform to allowed formats. Works with antivirus scan before processing.

Parsing Engine
Extracts structured sections (Education, Skills, Work Experience, Projects) using regex + libraries (PyMuPDF, python-docx).

NLP/Skill Extractor
Uses spaCy for entity recognition, keyword extraction, and synonym expansion. Powers skill-gap detection and action-verb suggestions.

Job Requirement Parser
Processes recruiter-provided job descriptions, extracting required keywords/skills for comparison.

Matching Engine
Compares parsed resume data with job requirements. Detects missing sections, inconsistent formats, and employment gaps.

Feedback Generator
Aggregates results into structured feedback. Balances positive and critical comments per UX guidelines.

Report Generator
Produces downloadable reports (PDF/HTML). Anonymizes personal data, adds fit-score, and optionally emails the recruiter.

Database
Stores parsed resume metadata, feedback logs, and minimal user/session data. Ensures no resumes are permanently stored.

Cloud Storage (temp files)
Holds uploaded resumes only during processing. Files are auto-deleted after analysis.

3.5 Chosen Architecture Pattern and Rationale

Pattern Selected: Layered + Modular Monolith (Service-Oriented)

- **Presentation Layer (Frontend):** React SPA for interactive UI.

- **Application Layer (API Gateway):** Flask endpoints for orchestration.
- **Business Logic Layer (Engines):** Parsing, NLP, matching, and feedback as independent modules.
- **Data Layer:** SQLite/PostgreSQL for metadata + temporary cloud storage.

Rationale

- **Simplicity:** Microservices would be overkill for an MVP. A modular monolith with clear separation of concerns is easier to maintain.
- **Extensibility:** Individual engines (e.g., Parsing, NLP, Matching) can later be split into microservices if scaling demands it.
- **Security:** The layered approach ensures proper enforcement of access control and secure data handling at each level.
- **Performance:** Serverless deployment (Vercel/Flask functions) ensures fast scaling for resume uploads without constant server overhead.

3.6 Technology Stack & Data Stores

- **Backend Language & Framework: Python** with the **Flask** framework, deployed as a **Serverless API**.
- **Frontend:** A **React.js** Single Page Application (SPA) for a dynamic and responsive user experience.
- **Database: SQLite** (Note: Serverless environments are often "stateless," so for a Vercel deployment, you might either omit the database or use a free cloud-based option like NeonDB or Vercel's own storage solutions if needed).
- **Resume Parsing Libraries: PyMuPDF** and **python-docx**.
- **NLP Library: spaCy**.
- **Deployment: Vercel**. It will host both the React frontend and the Python serverless backend, providing continuous integration and deployment directly from your code repository.

3.7 Risks & Mitigations

- **Risk: Critical Data Breach and PII Exposure**
  - **Description:** Resumes contain a high volume of Personally Identifiable Information (PII). A security breach could expose sensitive user data, leading to identity theft, reputational damage, and severe legal and financial penalties under data protection regulations (e.g., GDPR).

    - **Mitigation:** A multi-layered security strategy will be implemented. This includes **end-to-end encryption** (TLS 1.3 for data-in-transit, AES-256 for any data-at-rest), a strict data retention policy that ensures resumes are **purged immediately after analysis**, and robust **access control mechanisms** to prevent unauthorized access to the system's infrastructure.
- **Risk: Algorithmic Bias and Inaccurate Analysis**
    - **Description:** The core NLP model could produce biased or unfair results, systematically disadvantaging certain demographics, penalizing non-traditional resume formats, or inaccurately assessing a candidate's qualifications. This poses a significant ethical risk and undermines the tool's utility and credibility.
    - **Mitigation:** The NLP models will undergo **rigorous validation against industry-standard bias detection benchmarks**. The system will incorporate **Explainable AI (XAI)** principles, providing users with justifications for the feedback they receive. Furthermore, the analysis engine will be designed for continuous improvement through regular audits and updates based on performance metrics and user feedback.
- **Risk: Denial of Service (DoS) Attack**
    - **Description:** Malicious actors could target the file upload endpoint, overwhelming the service with high-volume or oversized requests. This would exhaust server resources, leading to service unavailability for legitimate users and potentially incurring high operational costs.
    - **Mitigation:** The API gateway will be fortified with **rate limiting** to prevent abuse from a single source. Strict **file size and type validation** will be enforced at both the client and server levels. The deployment platform's (Vercel) built-in **Web Application Firewall (WAF)** and DoS protection capabilities will be leveraged to provide an additional layer of defense.

3.8 Traceability to Requirements

Requirement	Requirement Description	Architectural Component(s)

ID		
RPRS-1	The system shall allow users to upload resumes in .pdf, .docx, and .txt formats.	React Client (upload UI), File Processing Module (server-side validation).
RPRS-2	The system shall allow users to create an account using their email and a secure password.	React Client (registration form), Flask API (user management endpoint), Database.
RPRS-3	The system shall parse the uploaded resume to extract key information, including contact details, work experience, education, and skills.	NLP Analysis Module (within the Flask API).
RPRS-4	The system shall generate feedback on resume content, including keyword optimization, formatting consistency, and the use of action verbs.	NLP Analysis Module (contains the rule engine and logic for feedback).
RPRS-5	The system shall provide a feature to compare the user's resume against a provided job description, highlighting matching keywords and identifying gaps.	Flask API (dedicated /compare endpoint), NLP Analysis Module.
RPRS-6	Users shall be able to download their analysis report in PDF format.	Flask API (using a report generation library to create the PDF), React Client (to initiate the download).
RPRS-7	The system shall provide a user dashboard to view a history of past analysis reports.	React Client (dashboard UI), Flask API (endpoint to fetch report history), Database (to store report metadata).
RPRS-8	The user interface shall be compliant with Web Content Accessibility Guidelines	React Client (requires semantic HTML, ARIA attributes, and accessible design).

	(WCAG) 2.1 AA standards.	
RPRS-9	The system must complete the analysis of a resume (up to 3 pages) within 10 seconds.	Flask API, specifically the performance of the File Processing and NLP Analysis modules.
RPRS-10	The system shall achieve 99.5% monthly uptime.	Deployment Infrastructure (Vercel), and robust error handling within the Flask API.
RPRS-11	The system must be able to handle 50 concurrent analysis requests without significant degradation in performance.	Deployment Infrastructure (leveraging Vercel's serverless scaling) and the Stateless design of the Flask API.
RPRS-12	All uploaded resumes must be automatically and permanently deleted from the system after the analysis report is generated and delivered to the user.	File Processing Module (implements the transient data handling policy).
RPRS-13	All data transmitted between the client and the server must be encrypted using TLS 1.2 or higher.	Deployment Infrastructure (Vercel, which enforces modern TLS protocols).
RPRS-14	All user passwords must be salted and securely hashed using a modern, strong algorithm.	Flask API (user management logic, using a library like Passlib for hashing).
RPRS-15	User authentication shall be secure, with options for social logins (e.g., Google, LinkedIn) to enhance user experience while maintaining security.	Flask API (integration with an OAuth 2.0 / OIDC compliant service).
RPRS-16	Access to administrative functions shall be protected by Multi-Factor Authentication (MFA).	Flask API (user management logic for admin roles, integrated with an MFA provider).

3.9 Security Architecture

- **Data in Transit:** All communication between the user's browser and the server will be encrypted using **TLS (HTTPS)**, which is provided by default on hosting platforms like Render.

- **Password Storage:** If user accounts are implemented, passwords will not be stored in plaintext. They will be salted and hashed using a strong algorithm like **Argon2** or **bcrypt**, managed via a Python library like passlib.
- **Input Sanitization:** The Flask backend will use libraries and best practices (e.g., Jinja2 templating) to sanitize user inputs, mitigating risks of Cross-Site Scripting (XSS) and other injection attacks.
- **Data Privacy:** As per the project constraints, uploaded resume files will be stored temporarily for analysis and deleted immediately after the session, ensuring user privacy.

4. Design

4.1 Design Overview

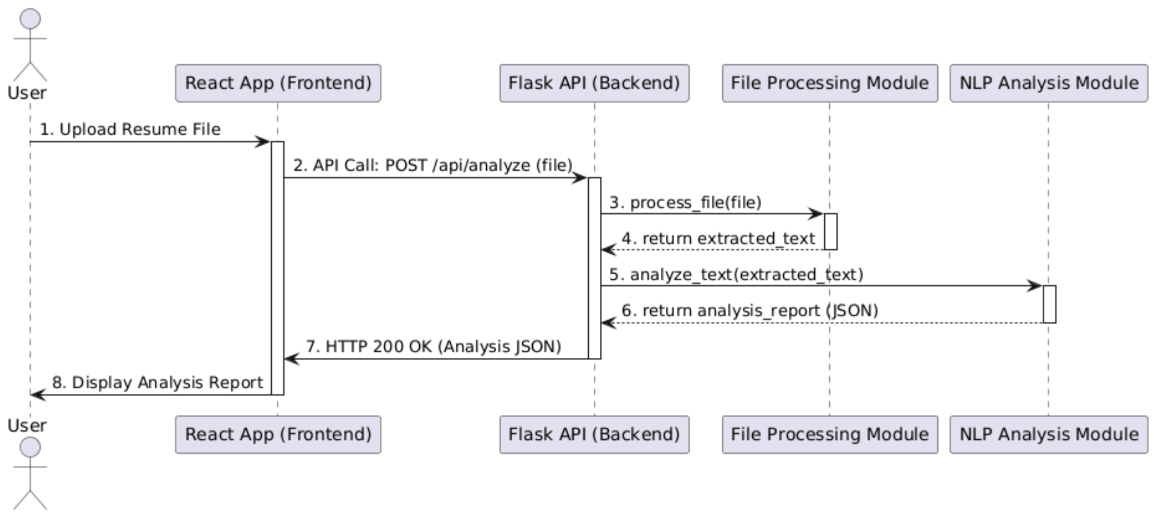
The system is designed as a **monolithic application** using the **Flask web framework**. This design keeps the architecture simple and easy to manage for a project of this scope. The application follows a layered approach:

1. **Presentation Layer (Frontend):** Simple HTML, CSS, and JavaScript pages served directly by Flask. This layer is responsible for capturing user file uploads and displaying the final analysis report.
2. **Business Logic Layer (Backend):** The core Flask application handles HTTP requests, manages file processing, orchestrates the parsing and analysis, and enforces all application rules.
3. **Data Access Layer:** A simple module to interact with the **SQLite** database for any necessary data persistence (e.g., user accounts). Resume files themselves are not persisted in the database.

4.2 UML Sequence Diagrams

At least 2 sequence diagrams covering 2 flows specific to your project

Flow 1: Successful Resume Analysis

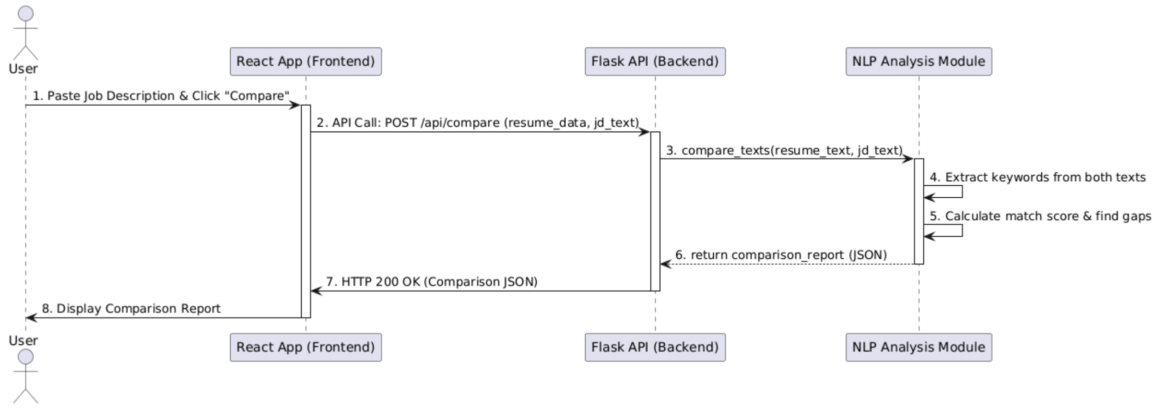

User
React App (Frontend)
Flask API (Backend)
File Processing Module
NLP Analysis Module
1. Upload Resume File
2. API Call: POST /api/analyze (file)
3. process_file(file)
4. return extracted_text
5. analyze_text(extracted_text)
6. return analysis_report (JSON)
7. HTTP 200 OK (Analysis JSON)
8. Display Analysis Report

This diagram illustrates the primary user flow where a user uploads a resume and successfully receives an analysis report.

Description of Interactions:

1. The **User** selects a file and clicks "Analyze" in the browser.
2. The **React App (Frontend)** sends the file in an asynchronous API call (HTTP POST) to the **Flask API (Backend)**.
3. The Flask API receives the request and sends the file to the **File Processing Module**.
4. The File Processing Module extracts the raw text using the appropriate library (e.g., PyMuPDF).
5. The extracted text is then passed to the **NLP Analysis Module**.
6. The NLP module uses the spaCy library to process the text and generate a structured analysis report in JSON format.
7. The Flask API receives the report and sends it back to the React App as the API response.
8. The **React App** receives the JSON data, updates its state, and dynamically renders the analysis for the user to see.

Flow 2: Resume to Job Description Comparison


User
React App (Frontend)
Flask API (Backend)
NLP Analysis Module
1. Paste Job Description & Click "Compare"
2. API Call: POST /api/compare (resume_data, jd_text)
3. compare_texts(resume_text, jd_text)
4. Extract keywords from both texts
5. Calculate match score & find gaps
6. return comparison_report (JSON)
7. HTTP 200 OK (Comparison JSON)
8. Display Comparison Report

This diagram illustrates the process of comparing an uploaded resume against a job description provided by the user.

Description of Interactions:

1. The **User** pastes a job description into a text field and clicks "Compare."
2. The **React App (Frontend)** sends the resume data and the job description text in an API call to the **Flask API (Backend)**.
3. The Flask API receives the request and passes both texts to the **NLP Analysis Module**.
4. The NLP module extracts key skills and terms from both the resume and the job description.
5. It then compares the two sets of data, calculates a match score, and identifies keyword gaps.
6. A structured **Comparison Report** in JSON format is generated and returned to the Flask API.
7. The Flask API sends the JSON report back to the React App in its response.
8. The **React App** receives the data and renders the detailed comparison, showing the match score and keyword analysis to the user.

4.3 API Design

Resume Upload and Parsing API

Endpoint: /api/v1/parse
Method: POST

Purpose: Submit a resume file (PDF, DOCX, or TXT) and optionally a job description for automated analysis.
Request:

- Content-Type: multipart/form-data
- Parameters:
    - file (required): Resume file
    - jobDescription (optional): Text input

Response Example in JSON (202 Accepted):

```
{

  "jobId": "123e4567-e89b-12d3-a456-426614174001",
  "status": "processing",
  "message": "Resume received for analysis."
}
```

Error Codes:

- 400 Bad Request: Invalid or missing file
- 413 Payload Too Large: File size exceeds 10MB
- 415 Unsupported Media Type: File type not supported
- 422 Unprocessable Entity: Corrupted or password-protected file
- 401 Unauthorized: Authentication failure

Parsing Results Retrieval API
Endpoint: /api/v1/results/{jobId}
Method: GET
Purpose: Retrieve the extracted sections, feedback, and match result for a submitted resume.
Authentication:

- Bearer JWT in Authorization header

Response Example in JSON(200 OK):

```
{
  "status": "completed",
  "sections": {
    "education": [...],
    "skills": [...],
    "experience": [...],
    "projects": [...]
  },
  "feedback": {
    "missingSections": ["skills"],
    "incompleteSections": ["projects"],
    "suggestions": [ ... ]
  },
  "score": 82,
  "matchPercentage": 76,
  "reportDownloadUrl": "/api/v1/download/{jobId}"
}
```

Error Codes:

- 404 Not Found: Invalid or expired jobId
- 401 Unauthorized

4.4 Error Handling, Logging & Monitoring

Error Handling:

- API errors follow a standard JSON structure with fields for code, message, severity, and timestamp.
- All client-facing error messages are descriptive and provide guidance for resolution.
- The system is designed for graceful degradation, ensuring that partial results are returned if non-critical parsing steps fail.

Logging:

- All API requests, parsing activities, and data store actions are logged in structured JSON format.
- Sensitive information (such as email, phone, address) is masked or excluded from logs.
- Logs include request IDs to facilitate tracing and are retained for 30 days; security/audit logs are retained for 90 days.

Monitoring:

- Application and infrastructure metrics, including resume processing times, error rates, and system uptime, are continuously monitored via Grafana and Prometheus.
- Alerts are configured for downtime, error spikes, and anomaly detection in traffic or security events.

4.5 UX Design

General Principles:

- The application interface complies with WCAG 2.1 AA accessibility standards, featuring high-contrast color palettes, keyboard navigation, and clear labeling for screen readers.
- Responsive design ensures usability across desktop and mobile browsers; minimum supported width is 320px.
- Interface flow is linear and user-friendly: resume upload, progress feedback, results presentation, and detailed report download.

User Interaction Flow:

- A drag-and-drop area guides the user on supported formats and file size limits.
- Visual progress indicators communicate the analysis status.
- Results dashboard distinguishes between complete, incomplete, and missing sections through clear iconography and color coding.
- Reports are concise and well-structured, featuring a balance of strengths and areas for improvement, downloadable as PDF.

Error Messaging:

- Error notifications are shown inline with clear explanations and, where possible, recommended corrective actions.
- Example: “The uploaded file format is not supported. Please upload a PDF, DOCX, or TXT file (max 10MB).”

4.6 Open Issues & Next Steps

Known Issues and Technical Debt:

- Only English resumes and job descriptions are supported (NLP pipelines for other languages planned).
- No support for scanned/image-based resumes (OCR integration is a roadmap feature).
- ATS integration and semantic gap analysis are not present in MVP.
- Limited industry-specific resume templates; expansion needed for wider use cases.

Next Steps:

- Incorporate spelling/grammar check using dedicated language APIs.

- Expand synonym library for more robust keyword/skill extraction.
- Add support for batch processing in UI (already present in backend).
- Begin work on machine learning–based section detection.
- Explore integration with partner ATS/job board APIs.

5. Appendices

5.1 Glossary

- **ATS:** Applicant Tracking System
- **JWT**: JSON Web Token
- **NLP:** Natural Language Processing
- **OCR**: Optical Character Recognition
- **TLS:** Transport Layer Security
- **MFA:** Multi-factor Authentication
- **WCAG:** Web Content Accessibility Guidelines
- **MFA**: Multi-factor Authentication:
- **SQL:** Structured Query Language
- **RTM:** Requirements Traceability Matrix
- **STP:** Software Test Plan

5.2 References

- Software Requirements Specification (SRS): Dynamic Resume Analyzer v1.0
- Software Test Plan (STP): Dynamic Resume Analyzer v1.0
- IEEE 42010: Systems and Software Engineering - Architecture Description
- OWASP Top 10 Security Practices
- NIST SP 800-160 (Security Engineering)
- RFC 7519: JSON Web Token (JWT)
- WCAG 2.1 AA Guidelines

5.3 Tools

- Architecture and UML: Draw.io, PlantUML
- API Documentation: Swagger/OpenAPI, Postman
- Performance Testing: JMeter, Artillery
- Monitoring and Alerting: Prometheus, Grafana

- Security Testing: OWASP ZAP, Burp Suite
- Document Processing: PyPDF2/pdfplumber, python-docx
- Development: Python 3.9+, Flask/FastAPI, React.js, PostgreSQL, Redis, Docker, AWS/GCP