My Resume Portfolio – Product Requirements Document

Overview

The AI-Powered Resume Building System is a web application that helps users improve and tailor their resumes using artificial intelligence. Users can upload their existing resume in various formats (PDF, DOCX, or plain text), and the system will parse the content into structured data (e.g., name, experience, skills, education). This data is stored securely in a PostgreSQL database for further processing. An integrated AI chatbot then interacts with the user to refine and enhance the resume based on a target job role provided by the user. The AI suggests improvements in wording, emphasis, and structure without fabricating any information, ensuring all enhancements remain truthful to the user's real experience. Once refinements are complete, the system formats the content into a polished resume using a predefined professional template. The conversational AI is powered by the Verbal AI SDK for managing dialogue, and an MCP Server may be utilized for advanced document parsing if required. The end result is a well-structured, role-targeted resume ready for download in PDF or DOCX format.

Features & Functionalities

User Input & Parsing

Users can begin by providing their current resume through a convenient upload interface or via direct text input:
• File Upload: The system accepts resumes in PDF, DOCX, or plain text (TXT) formats. Upon upload, the document's text content is extracted. For PDFs and Word documents, the system uses parsing libraries or an external service (MCP Server) to reliably convert the file into raw text.
• Text Parsing: After extraction, the resume text is parsed into structured fields. The parser identifies key sections such as Personal Details (name, contact information), Work Experience (job titles, companies, dates, and descriptions of responsibilities/achievements), Education (degrees, institutions, years), Skills, and other relevant sections (certifications, projects, etc.). This may involve detecting section headings and using natural language processing to classify each part of the text.
• Error Handling: If the document parsing encounters issues (e.g. an unrecognized format or irregular layout), the system will either fall back to a simpler text extraction method or prompt the user to verify and correct any ambiguities. Users also have the option to input or edit text manually if the automated parsing misses something.
• Data Storage: The parsed resume data is then saved into a PostgreSQL database under the user's profile (see Database & Data Management). Storing structured data allows the AI to easily retrieve and analyze specific parts of the resume during the enhancement process.

Database & Data Management

The system employs a PostgreSQL relational database to store user information and parsed resume content in a secure and organized manner:
• Schema Design: The database is structured to link Users with their Resumes. For example, a users table stores account details (user ID, name, email, hashed password, etc.), and a resumes table stores the resume content fields (foreign key to user, summary, work_experience, education, skills, etc. as separate columns or JSON data). This design enables a user to manage multiple resumes or versions if needed, and ensures data normalization (avoiding duplication of user info across resumes).

• CRUD Operations: Users can create a new resume entry by uploading a file, update their resume content through the chatbot or manual edits, and retrieve or delete their resume data. Each update (whether from user edits or AI suggestions) is written to the database, so the latest resume state is always saved. Versioning could be considered (e.g., saving previous versions or change history), although initially the system may simply overwrite the resume entry with the latest content for simplicity.
• Data Management: The database ensures integrity of the data (using transactions for updates) and supports queries for analytics or admin oversight (for example, counting how many resumes have been processed, or filtering resumes by target role for future feature insights). All personal data in the database is handled with care (see Security Considerations for encryption and protection measures).
• Scalability: PostgreSQL can handle numerous users and resume records; if usage grows, the schema can be optimized with indexing (e.g., index on user_id in resumes table) to maintain quick access. In future, sharding or read-replica setups can be used if the volume becomes very large.

AI Chatbot Assistance

A core feature of the system is the AI-driven chatbot that guides users in improving their resume content for a specific job role:
• Interactive Conversation: After parsing, the user enters a target job role or title (e.g. "Software Engineer", "Marketing Manager"). The AI chatbot (leveraging the Verbal AI SDK) then engages in a conversation about tailoring the resume to that role. It may start by summarizing the resume's strengths and suggesting areas to focus on for the given role.
• Role-Based Suggestions: The AI analyzes the user's experience and skills in light of the target role. It provides concrete suggestions such as highlighting certain skills, rephrasing bullet points to use strong action verbs, or reordering sections to prioritize relevant experience. For example, if the target role is Data Scientist and the user has a relevant project buried in the resume, the AI might suggest bringing that project into a prominent position or elaborating on it with technical details.
• User Guidance and Input: The chatbot maintains a helpful, conversational tone. It might ask the user questions to clarify details or gather more information. (e.g., "Can you provide more details about your role at XYZ Corp, so we can emphasize your achievements there?"). The user can answer these questions or skip them. All new information provided by the user during chat is incorporated into the resume content, but the AI will not invent any details on its own. It strictly uses the user's input and the existing resume data to make enhancements.
• Fact-Based Enhancements: The AI ensures that all suggestions remain factual and based on the user's actual background. For instance, it may suggest quantifying an achievement ("increased sales by 20%") only if the user has indicated that figure. If the user has not provided such data, the AI might encourage them to think of a real statistic or leave that point as-is. This guardrail prevents the creation of false or misleading resume content.
• Multi-Turn Editing: The conversation can continue for multiple rounds. The user might say, "I'm not happy with the wording of my summary," and the AI will then propose alternative phrasing. The user can accept the suggestion, ask for another option, or refine it further. This iterative loop continues until the user is satisfied with each section of the resume. The Verbal AI SDK manages the context of the conversation, so the AI remembers earlier parts of the discussion and the current state of the resume.
• User Control: At all times, the user has control over which suggestions to accept. The chatbot might offer a change, and the user can approve it (which triggers the system to update that field in the database) or reject it. Nothing is permanently altered until the user confirms, ensuring the user remains the final editor of their resume content.

Resume Enhancement & Formatting

Once the content has been refined via the AI assistance, the system handles the final assembly and formatting of the resume:

• Automated Enhancements: Before formatting, the system can perform some automated polishing. This includes checking for consistent tone and tense, scanning for any typos or grammatical issues that might have been introduced, and ensuring that the content fits within typical resume length guidelines (e.g., one or two pages). These checks can be done by the AI or additional language tools, and the user is informed of any potential issues (for example, if the resume is overly long, the AI may suggest which sections could be shortened).

• Predefined Template: The resume content is then injected into a predefined template that has a professional design. The template defines the layout – for example, section headings style, font choices, margin sizes, and how the information is organized on the page. Using a template ensures consistency and saves the user from worrying about design. The template chosen will be ATS-friendly (Applicant Tracking System compatible), meaning it avoids overly fancy formatting that could confuse automated resume screeners. It will, however, be visually clean and modern.

• Formatting Engine: The system's formatting engine takes the structured data (now updated with all the user-approved AI suggestions) and generates a properly formatted document. If implemented on a web stack, this might involve generating an HTML/CSS representation of the resume and then converting it to PDF. Alternatively, a library or service (such as a PDF generation library, or using the MCP server in a conversion mode) could be used to create a DOCX and PDF output. Key formatting tasks include:

• Ensuring the user's name and contact info appear clearly at the top.

• Listing sections in a logical order (often Summary, Experience, Education, Skills, etc., unless the user specified a different arrangement).

• Applying consistent bullet point styles, fonts, and spacing.

• Handling text overflow gracefully (e.g., adjusting font size or spacing if content is slightly too long for one page, or flowing to a second page in a controlled manner).

• Download Options: The final formatted resume is made available for the user to download. Common download formats will be PDF (for a ready-to-send version that preserves layout) and DOCX (so the user can do any further editing in Word or similar tools if they desire). The system may also allow the user to preview the formatted resume on screen before downloading, to ensure satisfaction.

• Template Customization (Future): Initially, the system will use a single well-designed template. In future versions, we may allow users to choose from multiple templates or customize elements (like accent color or font). This is noted here as a potential extension, though the first version focuses on one default template to simplify the experience and ensure quality.

System Architecture & Technology Stack

The resume builder is designed with a modern web architecture, divided into front-end, back-end, and integrated AI services:

• Frontend: A responsive web application (built with React.js or Next.js) provides the user interface. This includes pages/components for uploading the resume, the chat interface for AI conversation, and preview/download page for the final resume. Using React or Next allows for a dynamic user experience and easy integration of interactive elements like file pickers and chat windows. Next.js could be used if server-side rendering or SEO for a landing page is needed, but since this is primarily an app for logged-in users, a client-side React SPA is also suitable. The front-end communicates with the back-end via

RESTful API calls (or GraphQL endpoints) for all operations (upload, chat, etc.).
• Backend: The server-side application will be implemented in a robust runtime such as Node.js or Python. The choice can depend on team expertise or integration needs:
• If Node.js is chosen, frameworks like Express (or NestJS for a more structured approach) can be used to build the API endpoints.
• If Python is chosen, a framework like Flask or Django (with Django REST Framework) can provide similar capabilities.
The back-end exposes endpoints for uploading resumes, retrieving/saving parsed data, handling chat messages (which involve AI calls), and serving the final formatted resume file. It contains the business logic for orchestrating the parsing, AI suggestion integration, and formatting process.
• AI Service (Verbal AI SDK): The AI chatbot functionality is powered by the Verbal AI SDK integrated on the backend. This SDK likely communicates with a cloud-based AI model or service. The back-end will use it to send the resume data and user inputs to the AI and receive suggestion responses. Because it's an SDK, it might manage context (conversation state) and possibly provide utilities for formatting the AI's answers or controlling the style of the conversation. The integration is done server-side to keep API keys and logic secure (the front-end never directly talks to the AI service; it only talks to our backend).
• Document Parsing (MCP Server): For parsing resumes, especially in PDF or complex formats, the system may utilize an external MCP Server (Model Context Protocol server). This server can be a microservice specialized in document parsing: the backend would send the raw file or text to the MCP service which returns structured data (possibly in JSON format). Using an MCP server can improve parsing accuracy by leveraging advanced algorithms (such as OCR for scanned PDFs or intelligent content extraction). However, for simplicity and speed, the initial implementation might handle most parsing in-process (using libraries like PDFBox, textract, or python-docx) and only call the MCP server for difficult cases or as a future enhancement. The architecture is designed to accommodate this modularly.
• Database: PostgreSQL serves as the primary data store. The database is hosted on a secure server and the backend communicates with it via an ORM or direct SQL queries. PostgreSQL was chosen for its reliability, relational capabilities (important for linking users to resumes and possibly normalizing resume sub-data if needed), and its JSON support (which could be useful if we store some sections or AI feedback in JSON columns). The backend enforces SQL queries or ORM calls to be parameterized to prevent injection attacks (see Security).
• Infrastructure & Deployment: The system will likely be deployed on a cloud platform. The architecture could follow a microservice approach (with separate services for the main backend, the AI integration if offloaded, and the parsing server) or a simpler monolithic approach initially (the Node/Python backend handling API + calling out to external API/SDK for AI). Containerization (Docker) can be used to encapsulate the application, and orchestration (Kubernetes or a cloud service like AWS ECS) for scaling if needed. The front-end can be deployed as a static bundle or served via Node/Next server.
• Technology Stack Summary:
• Backend: Node.js (Express/Nest) or Python (Flask/Django), Verbal AI SDK (for AI interactions), HTTP/REST API design.
• Database: PostgreSQL 13+ (with appropriate tables and possibly extensions for full-text search or JSON handling as needed).
• Frontend: React.js/Next.js, HTML5/CSS3, possibly a UI library (Material-UI, Ant Design, etc.) for consistency in design.
• Parsing: PDF and DOCX parsing libraries, with optional integration to an MCP server for enhanced

parsing and conversion.
• AI Integration: Verbal AI SDK (which under the hood might use large language models to generate suggestions), requiring internet connectivity to the AI service if cloud-based.
• APIs: All interactions defined clearly (upload endpoint, chat endpoint, etc.), using JSON as the data format for communication. Authentication tokens will be handled in API headers for security.

With this tech stack, the system is positioned to be scalable, maintainable, and easy to extend (for example, swapping in a different AI service or adding new features) in future iterations.

Diagrams & Documentation

Below are diagrams and models illustrating the system architecture, user flow, database schema, and key API interactions for the AI-powered resume builder.

Architecture Diagram

The following is a high-level architecture diagram showing the major components of the system and their interactions:

flowchart LR
subgraph Client (Browser)
UI[Web Application (React/Next.js)]
end
subgraph Server (Backend)
API[Backend Service (Node.js/Python)]
DB[(PostgreSQL Database)]
end
subgraph External Services
AI[Verbal AI SDK/Service]
Parser[MCP Parsing Server]
end

```
UI -- Upload/Chat API calls --> API
API -- Read/Write --> DB
API -- Parse Request --> Parser
Parser -- Parsed Data --> API
API -- AI Queries --> AI
AI -- Suggestions --> API
API -- Formatted Resume --> UI
```

Explanation: The user interacts with the UI in their web browser. When a resume file is uploaded, the UI sends it via an API call to the Backend Service (API). The backend may delegate file content to the MCP Parser for detailed parsing; once parsed, structured data is stored in the PostgreSQL DB. During the resume refinement conversation, the UI sends the user's messages and target role to the API, which in turn calls the Verbal AI Service to get improvement suggestions. The suggestions flow back from the AI to the backend and then to the UI. Finally, when the user is done, the backend formats the resume (using template logic and data from the DB) and the UI allows the user to download the formatted

resume. All components are modular – for example, the Parser service could be optional or replaced, and the AI service is encapsulated via the SDK.

User Flow Diagram

The diagram below outlines the end-to-end user journey through the system, from starting with an existing resume to obtaining an enhanced version:

```
flowchart TD
Start((Start)) --> UploadResume[User uploads resume (PDF/DOCX/TXT)]
UploadResume --> ParseResume[System parses resume content]
ParseResume --> InputRole[User specifies target job role]
InputRole --> ChatbotAssist[AI chatbot provides improvement suggestions]
ChatbotAssist --> Iterate{Changes needed?}
Iterate -- Yes --> Refine[User & AI refine content further]
Refine --> ChatbotAssist
Iterate -- No --> FormatResume[System formats the final resume]
FormatResume --> DownloadResume[User downloads the formatted resume]
DownloadResume --> End((End))
```

Explanation: The flow begins at Start, where the user has an initial resume. The first step is Upload Resume, after which the System Parses the content. The user then provides a Target Role they are aiming for. The AI Chatbot then kicks in to suggest improvements tailored to that role. The user and AI go through an iterative loop (Changes needed? – Yes) where the user can apply suggestions or ask for further tweaks. This loop continues until the user is satisfied (answer No to Changes needed?), at which point the system proceeds to Format Resume with the template. Finally, the user Downloads the enhanced resume and the process Ends. This ensures a guided, user-in-control process from raw resume to improved outcome.

Database Schema

The system uses a relational database schema to keep track of users and resumes. A simplified Entity-Relationship diagram is shown below:

```
erDiagram
USER ||--o{ RESUME : owns
USER {
int user_id PK
varchar name
varchar email
varchar password_hash
// ... other user fields
}
RESUME {
int resume_id PK
int user_id FK
text summary
text experience
```

text education
text skills
text other_sections
// ... other resume fields
}

Explanation: Each User can own multiple Resume entries (though in many cases a user might work with one primary resume at a time). The USER table holds account information (with a primary key user_id and fields like name, email, login credentials, etc.). The RESUME table holds the actual resume content fields, including a foreign key user_id linking to the owner. In this model, key sections of the resume (summary, experience, education, skills, etc.) are stored in separate fields or as large text blocks. This structure makes it easy to update individual sections as the user accepts AI suggestions. Additional tables could exist for normalized data (for example, an EXPERIENCE table with one row per job and a foreign key to RESUME), but for the MVP a single resume table as shown is sufficient. All references are properly keyed to maintain referential integrity. For instance, deleting a user can cascade to delete their resumes, and queries joining user and resume tables can retrieve full information for a user's resume editing session.

API Interaction Models

This section describes the key API endpoints and interaction patterns between the front-end and back-end (and onward to external services):

sequenceDiagram
participant U as User
participant UI as Frontend UI
participant BE as Backend API
participant PAR as MCP Parser Service
participant AI as Verbal AI Service

```
U ->> UI: Upload resume file (PDF/DOCX/TXT)
UI ->> BE: **POST** /resumes (resume file)
BE ->> PAR: Parse document content
PAR -->> BE: Parsed data (JSON fields)
BE ->> DB: Store parsed resume data
BE -->> UI: Return parse result (success/failure)

U ->> UI: Enter target job role
UI ->> BE: **POST** /chat (target role info)
BE ->> AI: Send resume data + target role
AI -->> BE: Return initial suggestions
BE -->> UI: Deliver AI suggestions (chat message)

loop Refinement
    U ->> UI: Send feedback or request changes
    UI ->> BE: **POST** /chat (user message)
    BE ->> AI: Send context + user message
    AI -->> BE: Return follow-up suggestion
```

```
    BE -->> UI: Deliver AI response
end

U ->> UI: Confirm final version ready
UI ->> BE: **GET** /resumes/{id}/formatted
BE ->> DB: Retrieve latest resume content
BE ->> UI: Return formatted resume file (PDF/DOCX)
U ->> UI: Download file
```

Explanation: The sequence diagram above shows interactions in a typical use case. First, the user triggers a file upload (POST /resumes), which the Frontend UI sends to the Backend. The backend calls the Parser service to get structured data, saves it in the DB, and responds to the UI confirming the resume was parsed. Next, the user provides the target role and possibly initiates the chat session (POST /chat). The backend includes the relevant resume data and target role in a request to the Verbal AI Service, which returns suggestions that are relayed back to the UI. The loop of refinement is represented where the user and AI exchange further messages via repeated calls to the chat API. Finally, when the user is satisfied, they (implicitly or explicitly) request the final formatted resume (GET /resumes/{id}/formatted or a similar endpoint). The backend then fetches the updated resume content from the database, generates the formatted document, and returns it to the UI for download. All these API endpoints are secured (only the authenticated user can access their resume endpoints) and use standard HTTP methods for clarity. The above model ensures a clear separation of concerns: the front-end handles user interactions and display, the back-end handles logic and integrations, and external services handle specialized tasks (AI and parsing).

Implementation Plan

The development of the AI-powered resume builder will be carried out in phases, ensuring core functionality is delivered first and enhancements are layered on subsequently. Below is a phased implementation plan:

1. Phase 1: Core System Setup – Establish the foundational infrastructure for the application.

• Backend & Database: Set up the PostgreSQL database and develop the basic backend API (skeleton of endpoints for upload, chat, etc.). Decide on the back-end framework (Node.js or Python) and implement a simple health-check and user authentication module. Integrate the Verbal AI SDK in a basic form (e.g., a test endpoint that returns a dummy AI response) to verify connectivity. Also, configure the environment for the MCP Parser (or an equivalent library) — for now, maybe just ensure the server can accept a file and run a simple parse (extract plain text).

• Frontend: Initialize the React/Next.js app with a basic layout. Implement a simple file upload component and a text display area for results. At this stage, the UI may not do much beyond sending the file to the backend and showing a success message.

• Goal: By end of Phase 1, the system architecture is in place: a user can upload a file and the file's text is stored in the database via the backend. The skeleton for chat is in place (though not fully functional), and the AI service connectivity is tested. Essentially, the "plumbing" is done, allowing for feature development in subsequent phases.

2. Phase 2: Resume Parsing & Chatbot Functionality – Implement the resume parsing logic and the interactive chatbot feature.

• Resume Parsing: Develop robust parsing for resumes. This could mean writing parsing functions to

split the raw text into sections and fields, or integrating the MCP Server fully at this point for an advanced parse. Ensure the parsed data is correctly saved into the structured database schema. Add logic to handle various edge cases in resumes (missing section titles, unusual formatting). Also, create an interface or log for developers to review parse accuracy (for later improvement).

• AI Chatbot: Build the conversation flow using the Verbal AI SDK. Define conversation scripts or prompts for the AI so it knows how to handle the context (for example, provide the AI with a system prompt that includes the user's parsed resume and an instruction to act as a resume coach). Implement the front-end chat UI: a chat window where the user sees the AI's suggestions and can type responses or requests. The backend chat endpoint will send user messages to the AI service and return the AI's reply.

• Integration: Tie parsing results to the chatbot – once parsing is done (from upload), automatically initiate the chatbot with a greeting and maybe a summary of parsed info or a question about target role. Ensure that the chat can fetch and update the resume data in the DB as suggestions are accepted (for example, if AI suggests a new summary and user says "Yes, apply that", then update the summary field in the DB).

• Testing at this stage: By end of Phase 2, a user can go through the full cycle of uploading a resume, having it parsed, and engaging in a basic dialogue with the AI to get improvement suggestions. The changes might not yet reflect in a downloadable formatted resume, but the system now achieves its primary interactive functionality. Conduct tests with a variety of resumes and roles to fine-tune the AI prompts and parsing reliability.

3. Phase 3: Resume Enhancement & Formatting Engine – Implement the content enhancement application and the formatting output.

• Apply Suggestions: Expand the chatbot functionality to not just suggest but also apply changes. Develop a mechanism in the UI for the user to accept or reject each suggestion. If accepted, update the displayed resume content and save it to the DB. Possibly allow the user to edit text directly in a form as well (some users might prefer direct editing in addition to chat – if in scope). Ensure the AI suggestions are reflected in the stored structured data.

• Formatting Engine: Develop the template layout for the resume and the code to generate the final document. This may involve creating an HTML template and using a headless browser or PDF library to render PDF. Or using a library like docx-templater for Word format. Implement this on the backend, triggered when the user is done editing. Create the API endpoint for downloading the formatted resume.

• Frontend for Download: In the UI, provide a "Preview" or "Download" button once the chat is concluded. If a preview feature is added, show a snapshot (perhaps a PDF viewer or an image) of the formatted resume so the user can see it before downloading.

• Polish & Refinement: This phase also includes improving the AI's output tone and consistency now that end-to-end flow is working. Work on any remaining issues from Phase 2 (parsing errors or awkward AI suggestions). Ensure the template format looks good for various lengths of content. If multiple page resumes are possible, handle page breaks appropriately.

• Goal: By end of Phase 3, the system will be fully functional: Users can upload a resume, converse with AI to enhance it, and obtain a newly formatted resume document. Essentially, the product's core value proposition is delivered.

4. Phase 4: Testing, Security, and Deployment – Rigorously test the system and prepare for production release with strong security and stability.

• Testing: Perform comprehensive testing including unit tests for parsing logic, integration tests for the upload-to-download workflow, and user acceptance testing with a small group. Test with different

resume formats (various layouts, lengths, and file types) to ensure parsing is robust. Also test the AI chatbot with different user inputs to ensure it handles them gracefully (for instance, irrelevant answers, or attempts to make it add false info – the AI should politely refuse or redirect).
• Performance Optimization: If needed, optimize the system for performance. This could involve caching the parsed result so the AI doesn't need to repeatedly fetch from the DB, or batching database writes. Ensure that the PDF generation is not too slow for large resumes (maybe generate asynchronously if needed and provide a loading indicator).
• Security Audits: Review the system for security vulnerabilities. Ensure that user authentication tokens are properly validated on each API call. Guard against common web vulnerabilities: SQL injection (use parameterized queries/ORM), XSS (sanitize any data that might be rendered in UI, though most data is user-provided text), CSRF (if using cookies, implement CSRF tokens or same-site cookie attributes). For file uploads, ensure that only allowed file types are accepted and they are scanned or handled in a way that no malicious content can execute on the server. Apply data encryption where appropriate (for instance, encrypt resume text in the database or rely on database encryption at rest).
• Deployment: Containerize the application (if using Docker) and deploy to a cloud environment. Set up the database in a managed cloud database service or secure VM. Configure environment variables for any secrets (database credentials, AI API keys). Use HTTPS for all client-server communication (obtain TLS certificates). Also, set up monitoring and logging – for example, track errors, response times, and usage metrics to quickly identify issues and understand usage patterns post-launch.
• Documentation & Training: Write user documentation or an onboarding guide explaining how to use the system (this might be a help section in the UI). Also document the system for future developers (architecture docs, how to run the system, etc., some of which are covered by this PRD and subsequent technical design docs). Train any internal staff (if this will be handed to a support or operations team).
• Launch: After confirming everything is stable and secure, proceed with a production launch. This may include a beta period with monitored performance. Collect user feedback for future improvements (like additional features or template choices).
• Goal: Phase 4 ensures the product is not only feature-complete but also reliable, secure, and user-ready. By the end of this phase, the AI-powered resume builder can be released to end users with confidence in its stability and security.

Security Considerations

Security and privacy are paramount given the system will handle personal resume data (which can include sensitive personal information and career details). The design will incorporate the following measures:
• Data Encryption: All sensitive data stored in the database will be protected. Passwords are hashed (using strong hashing algorithms like bcrypt or Argon2) and never stored in plain text. The resume content and personal details may be encrypted at the database level or application level to prevent exposure even if the database is compromised. At the very least, database encryption at rest will be enabled (most cloud DB services provide this by default). Backups of the database will also be encrypted. In transit, all data exchanges between front-end and back-end use HTTPS to encrypt data over the network, preventing eavesdropping on resume content or user credentials.
• Authentication & Authorization: Users will authenticate (likely via email/password or single sign-on if integrated later). The system will enforce session management best practices – e.g., using secure, HttpOnly cookies or JWT tokens for API auth. Each API request will be authorized: the backend ensures that the user making the request has the rights to the resource (for example, user A cannot fetch or

modify user B's resume data by changing an ID). Direct object reference attacks are mitigated by scoping queries to the authenticated user's context. If an admin interface exists, admin privileges are separately managed and logged. After a period of inactivity, user sessions may expire to reduce risk of stolen tokens.

• Secure API & Input Handling: All inputs (file uploads, text fields, chat messages) are validated and sanitized. For file uploads, the system checks the file type and size against acceptable limits. It processes the files in memory or in a sandboxed environment to avoid malicious file exploits. The parsing component (especially if using an external MCP service) is treated carefully – any code execution or scripts embedded in documents (like macros in a DOCX) are not executed, only textual content is extracted. The APIs have rate limiting or other abuse protections to prevent denial-of-service or brute force attacks (e.g., too many login attempts triggering a lockout). Communication with external services (the AI and parser) is done securely: API keys or credentials for these are stored server-side (never exposed to the client) and are kept in secure config storage.

• Privacy and Data Protection: The system will adhere to privacy best practices. Users' resume data is their own – it will not be shared with third parties without consent. If using the AI service involves sending resume content to a third-party AI API, users should be informed (via terms of service or a prompt) that their data is being processed by the AI provider for the purpose of getting suggestions. All such interactions will comply with data protection regulations (for example, GDPR if applicable, meaning we'd allow a user to delete their data permanently upon request, etc.). Audit logs can track who accessed data and when, which is useful for security auditing.

• Penetration Testing & Reviews: Before launch, and periodically thereafter, the system will undergo security reviews. This includes code reviews focusing on security, dependency vulnerability scans (to catch any known issues in libraries), and possibly third-party penetration testing. Any findings will be addressed promptly.

• Continuous Monitoring: Once live, implement monitoring for unusual activities (e.g., many failed logins could indicate a brute force attempt, which can trigger IP blocking). Also, ensure that any error logs do not inadvertently log sensitive information (like not logging full resume content on an error, to avoid it being stored in plaintext logs).

By taking these measures, the resume builder will protect user data and maintain trust. Security will be an ongoing commitment throughout the product's life cycle, not just a one-time setup.

———

End of Document.