CV Scanner

ARCHITECTURAL REQUIREMENTS SPECIFICATION

Version: 4.0.0

Date: September 28th, 2025

Status: Final

Document Type: Architectural Requirements

Project: CV Scanner - Automated CV Analysis System

Team Members

Marcelo Parsotam

Unaisah Hassim

Talhah Karodia

Abdullah Pochee

Ronan Smart

Table of Contents

1. Architectural Design Strategy

We have Chosen Design Based on Quality Requirements as the strategy of our system.

Our system has clearly defined quality goals that drive architectural decisions. For example:

- **Modifiability**: The Python CV scoring engine implements the Strategy Design Pattern to allow dynamic switching between different ranking strategies (e.g., ranking by experience, education, or skill match). This design supports our modifiability goal by enabling developers to add new ranking logic without modifying existing code. Strategies are encapsulated in individual classes and plugged into the engine at runtime.

- **Usability**: A clean interface, interactive tooltip tutorial, and click-minimized workflows were designed early based on usability requirements.

- **Security**: Access control and bcrypt password hashing were architectural priorities.

- **Reusability**: Shared components like the registration form and API endpoints are used in multiple contexts (user self-registration and admin-created users).

- **Interoperability**: Support for multiple CV file formats (.pdf, .doc/.docx) and normalization of data was architected into the processing pipeline.

1.1 Why we chose this design strategy:

By designing based on quality requirements, we ensure the final architecture delivers on what matters most for the system and its users, not just functionality, but experience, extensibility, and robustness.

2 Architectural Strategies

We have a 3-Layered Architecture for our system to work.

Our system follows a 3-tier separation:

1. **Presentation Layer**: The presentation layer internally follows the MVC pattern, where state (model), display logic (view), and user event handling (controller) are separated. This is implemented using Reacts component model and hooks/state management.

2. **Application Layer (Business Logic)**: Java Spring API manages auth, routing, user roles, and orchestrates CV handling.

3. **Data Layer**: Manages database interaction and persistence. As well as AI Parsing and storing of data.

This clear separation enables:

- **Independent development** of frontend, backend, and services.
- **Modifiability**: Changes to one layer don't affect others.
- **Security and Role Control**: Isolated in the Application layer.
- **Interoperability**: Easy integration of third-party or native services.

3 Architectural Quality Requirements

3.1 User Friendliness (Usability)

The system must be easy to use and understand for all user roles by providing a clean, consistent, and role-specific interface with minimal training required. Tasks such as viewing CVs, uploading new documents, and managing users should require no more than 3 clicks. Tooltips and feedback messages must be available to guide users during their tasks.

Justification

A clean and intuitive interface lowers the learning curve and minimizes user errors. By keeping interactions within **3 clicks** and providing tooltips/feedback, users (Admins, Editors, and Viewers) can efficiently perform tasks with little or no training. This reduces support costs, increases adoption, and improves user satisfaction.

3.2 User Management (Security)

The system must restrict access to the User Management page so that only users with the admin role can access it. This includes Adding, Editing and Removing users from the system of Editor and Viewer Role.

The system shall securely store user passwords using the bcrypt hashing algorithm with salting. This ensures passwords are not stored in plaintext and protects against brute-force and rainbow table attacks.

Justification

Restricting access to the User Management page ensures **role-based access control**, preventing unauthorized changes to sensitive system data. Using **bcrypt with salting** guarantees secure password storage, which is critical to prevent credential theft in case of a data breach. These measures collectively uphold confidentiality, integrity, and compliance with security best practices.

3.3 Different CV document types (Interoperability)

The system must support uploading and parsing of CVs in both .pdf and .doc/.docx formats. Regardless of format, all parsed data must be normalized into a common internal structure to ensure consistent downstream processing. The parser must extract key fields such as Name, Contact Information, Work History, Skills, and Education from both file types.

Justification

Supporting multiple formats (.pdf, .doc/.docx) ensures that users can upload CVs in the most common industry-standard formats, improving accessibility and adoption. Normalizing parsed data into a common structure avoids downstream inconsistencies, allowing all processing modules (scoring, ranking, searching) to work uniformly regardless of input type. This increases interoperability and robustness.

3.4 Reusing Register Logic (Reusability)

The system must promote component and logic reusability. For example, the user registration form and backend endpoint must support both self-registration by users and admin-initiated user creation. This reduces code duplication and simplifies maintenance.

Justification

Reusing the same registration logic for both **self-registration** and **admin-initiated creation** prevents code duplication and inconsistencies in business rules (e.g., password policies, validation checks). This reduces maintenance overhead, ensures uniform behaviour across the system, and simplifies testing since only one logic flow needs to be verified.

4. Architectural Design and Pattern

This section outlines the fundamental architectural patterns selected for our system, explaining the design decisions behind each choice. We examine how these patterns address specific quality attributes and support our system's functional requirements through deliberate implementation strategies.

Pattern 1: Three-Layer Architectural Pattern

Motivation and Rationale

The three-layer architecture was chosen to establish clear separation of concerns across distinct functional domains. This separation directly supports maintainability and enables parallel development efforts across specialized teams.

Quality Attributes Supported

- **Modifiability**: Isolated layers allow changes in one domain without impacting others
- **Testability**: Each layer can be tested independently with appropriate mocking strategies
- **Scalability**: Individual layers can be scaled based on specific resource requirements
- **Team Productivity**: Enables frontend, backend, and data specialists to work concurrently

Implementation Approach

Our implementation structures the system into three well-defined layers:

- **Presentation Layer**: React-based user interface handling all client-side interactions
- **Application Layer**: Spring Boot services containing business logic and workflow orchestration
- **Data Layer**: Azure SQL Database combined with AI processing services for data persistence and transformation

Pattern 2: Transaction Script Pattern

Motivation and Rationale

The Transaction Script pattern was selected for its alignment with our system's primarily procedural business processes. Each CV processing operation follows a clear, sequential workflow that maps naturally to discrete transaction scripts.

Quality Attributes Supported

- **Simplicity**: Reduces architectural complexity for straightforward CRUD operations
- **Performance**: Minimal abstraction overhead for frequently executed operations
- **Understandability**: Linear code flow matches business process documentation
- **Rapid Development**: Accelerates implementation of new business transactions

Implementation Approach

In our Spring Boot controllers, each endpoint encapsulates a complete business transaction:

- Authentication and authorization workflows in AuthController
- End-to-end CV processing pipelines in CVController
- Each method maintains full responsibility from input validation through response generation

Pattern 3: Model-View-Controller (MVC) Pattern

Motivation and Rationale

The MVC pattern provides structured organization for our React frontend, separating data management, user interface rendering, and user interaction handling. This separation aligns with modern frontend development practices and component-based architecture.

Quality Attributes Supported

- **Maintainability**: Isolated concerns prevent ripple effects from UI changes
- **Reusability**: Component-based architecture enables UI element reuse
- **Testability**: Views, controllers, and models can be tested independently
- **Developer Experience**: Familiar pattern reduces cognitive load for frontend developers

Implementation Approach

Our React implementation clearly separates:

- **Models**: State management through React hooks and context API
- **Views**: Presentational components for CV dashboard, upload interfaces, and admin panels
- **Controllers**: Event handlers and effect hooks managing business logic and API interactions

Pattern 4: Gateway Pattern

Motivation and Rationale

The Gateway pattern centralizes external service communication, providing a unified interface for API interactions. This abstraction simplifies error handling, request transformation, and service configuration management.

Quality Attributes Supported

- **Consistency**: Uniform approach to external service communication
- **Configurability**: Centralized management of API endpoints and credentials
- **Reliability**: Consistent error handling and retry mechanisms across all external calls
- **Security**: Single point for implementing security protocols and headers

Implementation Approach

Our custom api.ts gateway provides:

- Unified apiFetch() and aiFetch() methods for all external communications
- Automatic header management and content-type negotiation
- Environment-based configuration for different deployment targets

- Consistent error handling and logging across all API interactions

Pattern Integration and Synergy

Complementary Pattern Relationships

These patterns work together to create a cohesive architecture:

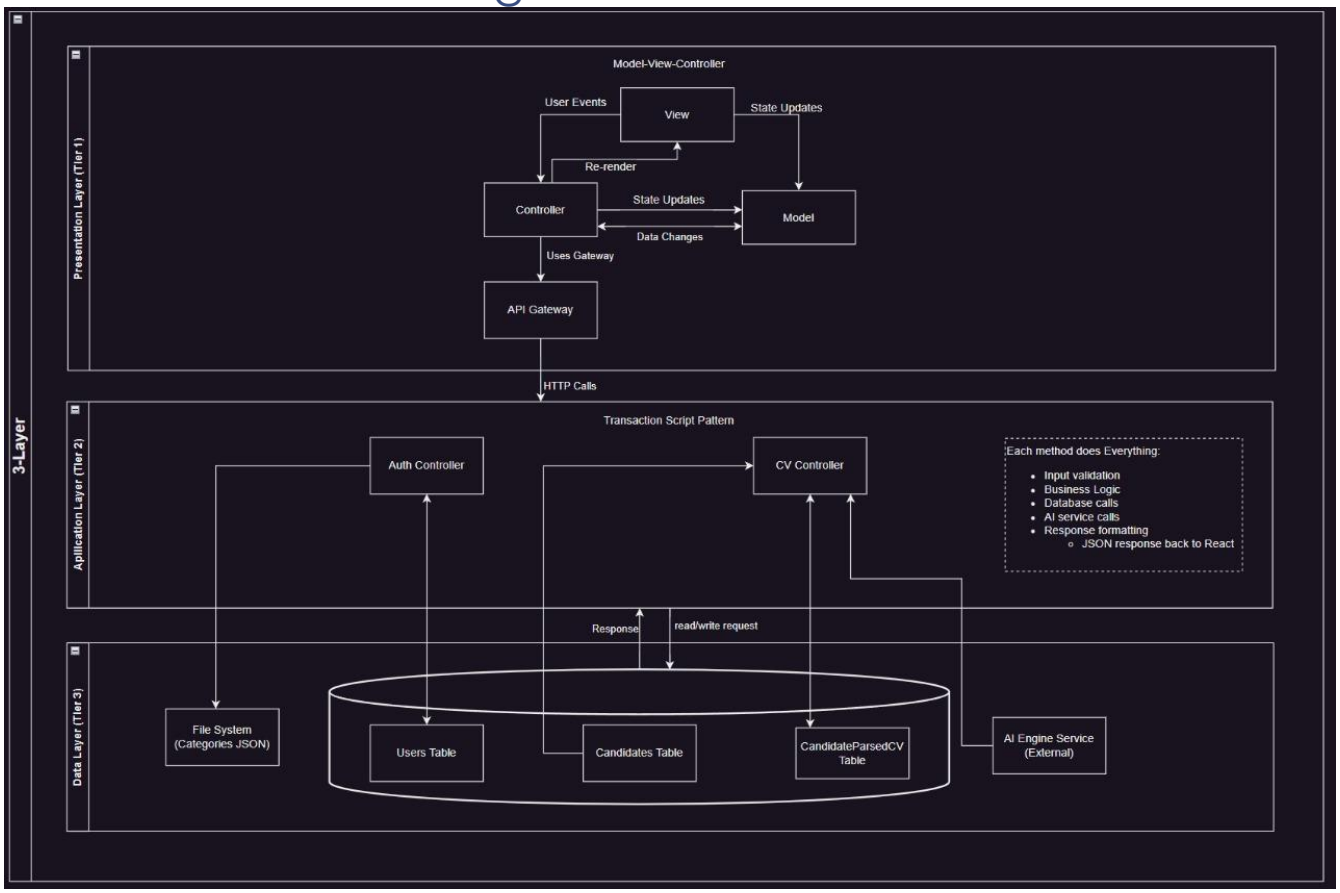
- **Three-Layer** provides the structural foundation
- **Transaction Script** organizes business logic within the application layer
- **MVC** structures the presentation layer implementation
- **Gateway** manages cross-layer communications consistently

Quality Attribute Trade-offs

While these patterns provide significant benefits, we acknowledge certain trade-offs:

- **Transaction Script** may require refactoring if business logic complexity increases significantly
- **Three-Layer** introduces some communication overhead between layers
- **MVC** in React requires discipline to maintain clear separation in component design

4.1 Architectural Design



4.2 Overview

This architectural diagram illustrates our 3-layer system where the React frontend (MVC pattern) communicates through an API gateway to Spring Boot controllers (Transaction Script pattern), which orchestrate complete business transactions by coordinating between Azure SQL database and AI processing services, with clear separation of concerns ensuring that user interface, business logic, and data processing remain independently maintainable while working together through well-defined communication channels.

5. Architectural Constraints

The Constraints followed below is the needs the client have given us to develop with and must be followed throughout the whole production phases; they are as follows:

- System must be hostable on:
  - IIS
  - Azure App Services
- Must use:
  - Azure SQL Database
  - .NET or Java Spring APIs
  - Modern JS framework (Angular, React, or Vue)

6. Technology Requirements

The technologies we have chosen work inline to the requirements that are expected by our client and some work depending on each other's performances, they are as follows:

6.1 Java Spring API

- **Pros**
  - Java Spring has robust security.
  - It is great for scalability and performance for large applications.
  - It has a variety of libraries that allow a lot of writability and has community support.
  - It has comprehensive testing framework support.
  - It has Cross-platform compatibility and JVM optimization.
- **Cons**
  - Memory consumption is higher than lighter frameworks.
  - Configuration may be complex as the system grows.

- **Justification**
    - Choice for this was to follow the client's needs being .NET and Java Spring, so we chose Java Spring as learning .NET would take a lot more time.

6.2 React Framework for UI

- **Pros**
    - It has a Component-based architecture which promotes Code Reusability.
    - A virtual DOM for efficient rendering and performances.
    - Many third-party libraries that can make the interface more writable.
    - Flexible and can be integrated with various backends.
- **Cons**
    - Rapid development pace can lead to frequent breaking changes.
    - Requires additional libraries for complete functionality, (e.g. routing, state management).
    - Can become complex for large applications without proper architecture.
- **Justification**
    - Choice for React was between 3 frameworks (React, Angular, Vue). We chose React because it will be a lot easier to work with separated data and makes loading a lot easier for the system as it loads by components.

6.3 Azure Database

- **Pros**
    - Built in Security features including encryption and threat detections.
    - It is very Scalable.
    - Has Integrated backup and disaster recovery solutions.

    - Integration with other Microsoft Azure services which need to be adhered to.

- **Cons**
    - Cost is high for large-scale operations.
    - Potential latency issues depending on geographic location.
- **Justification**
    - This is a requirement from the client to be used. Specified in our Technology Requirements.

6.4 Python based AI

- **Pros**
    - Extensive machine learning libraries.
    - Simple and readable syntax.
    - Cross-platform capability.
    - Rich ecosystem for AI/ML development.
- **Cons**
    - Slower execution speeds compared to compiled languages.
    - High memory consumption for large datasets.
    - Dependency management can become complex.
- **Justification**
    - AI in python simplifies a long process, making production a lot easier to manage, and with the aid of the C++ Engine, the speed is compensated using the writability of the language.