Architecture Documentation

DEVision – Job Applicant Subsystem

Squad Hercules

28 Nov 2025

Team Members

Le Trung Kien - s3878260
Nguyen Hoang Son – s3990627
Nguyen Tan Phat - s3911749
Ung Xuan Dat - s3932156
Chau Tung Nguyen - s3976069

Table of Contents

1. Introduction 2
1.1 Purpose of This Document 3
1.2 Key Features of the JA Subsystem 3
1.3 Architecture Style Overview 3
2. Squad Data Model 3
2.1. Data ModelData Model Diagram 3
1.1.1. How the Data Model Supports the Core Features in the SRS 3
1.1.2. Applicant Registration & Login 3
1.1.3. Applicant Profile Management 3
1.1.4. Job Search & Matching 4
1.1.5. Job Application Submission 4
1.1.6. Premium Subscription & Payment Data 4
1.1.7. Cross-Team Data Exchange with Job Manager 5
1.1.8. Support for Microservices and Sharding (Ultimo) 5
2.2. Sharding 5
3. Squad Container Diagram 6
3.1. Job Applicant Frontend Container Diagram 6
3.1.1. Frontend Technology Stack 6
3.1.2. User Context 6
3.1.3. Core Containers and Responsibilities 6
3.1.4. Feature Modules 7
3.1.5. Frontend–Backend Communication 8
3.2. Squad Backend Container Diagram 8
3.2.1. Backend Technology Stack 8
3.2.2. API Gateway 9
3.2.3. Purpose of Core Backend Containers (Business Microservices) 9
3.2.4. External Providers 10
3.2.5. Backend Communication with Other Subsystems 11
4. Backend Component Diagram 11
4.1. Authentication Microservice 12
4.2. Authorization Microservice 13
4.3. Admin Microservice 14
4.4. Notification Microservice 15
4.5. Applicant Microservice 16
4.6. Premium Applicant Subscription Microservice 17

4.7. Application Microservice 19
5. Frontend Component Diagram 20
5.1. Login Component Module (Job Applicant) 21
5.2. Registration Component Module (Job Applicant) 21
5.3. Notification Component Module (Job Applicant) 22
5.4. Profile Component Module 22
5.5. Application Dashboard Module 23
5.6. Application Form Module 23
5.7. Job Listing Module 23
5.8. Account Management Module 24
5.9. Payment Module 24
6. Architecture Rationale 25
6.1. Data Model Justification 25
6.1.1. Maintainability 25
6.1.2. Extensibility 25
6.1.3. Resilience 26
6.1.4. Scalability 26
6.1.5. Security 27
6.1.6. Performance 27
6.2. Frontend Architecture Justification 28
6.2.1. Maintainability 28
6.2.2. Extensibility 28
6.2.3. Resilience 29
6.2.4. Scalability 29
6.2.5. Security 29
6.2.6. Performance 30
6.3. Backend Architecture Justification 30
6.3.1. Maintainability 30
6.3.2. Extensibility 31
6.3.3. Resilience 31
6.3.4. Scalability 31
6.3.5. Security 32
6.3.6. Performance 32
7. Conclusion 33
8. References 33

1. Introduction

1.1 Purpose of This Document

This document provides an overview of the Job Applicant (JA) subsystem within the DEVision platform, detailing its system design, data model, and key architectural decisions. It includes C4 diagrams, a detailed data model with entities and relationships, and outlines the API contracts and sharding strategies. This serves as a reference for stakeholders and future developers to understand the subsystem's structure and design choices.

1.2 Key Features of the JA Subsystem

The JA subsystem supports the following key features:

- User Registration & Authentication (Email/SSO login)
- Profile Management (Personal details, education, work experience, skills)
- Resume & Portfolio Management (Upload and manage CV, cover letter, media)
- Job Search & Application (Search job posts and apply)
- Subscription Management (Premium services)
- Payment Processing (Integration with Payment Gateway)
- Real-Time Notifications (Kafka-based alerts)
- Admin Controls (Admin login and management)

1.3 Architecture Style Overview

Our system adopts a React component-based architecture on the frontend, interacting with backend services through REST APIs. On the backend, we employ a microservice architecture built with Spring Boot, supported by Kafka for asynchronous communication, Redis for caching and token management, and MongoDB (with GridFS for media storage) with country-based sharding to ensure scalable and efficient data storage.

2. Squad Data Model

2.1. Data ModelData Model Diagram

[Complete System ERD.png]

1.1.1. How the Data Model Supports the Core Features in the SRS

Our data model is designed to fully support all Simplex, Medium, and Ultimo features described in the DEVision Job Applicant and Job Manager SRS. Instead of listing each entity, this section explains how the data stored in the model enables the key business capabilities required for both subsystems and how the two teams exchange data.

1.1.2. Applicant Registration & Login

Our data model is designed to support all core features of the Job Applicant (JA) and Job Manager (JM) subsystems described in the SRS, while also aligning with the architectural boundaries defined by our microservices. For Applicant Registration and Login, the model stores all essential information needed to support both normal and SSO-based authentication flows. Core profile fields such as email, password hash, and country allow the system to enforce unique email constraints, validate user credentials, and immediately determine the correct database shard for persistence. Authentication-related attributes further support the generation of encrypted JWE tokens and the management of refresh or revocation states through Redis. Storing the country value at registration is particularly important, as it directly satisfies the Ultimo requirement for region-based sharding and ensures that newly registered users are placed into the correct shard from the very beginning. Altogether, these elements allow the system to perform secure, efficient, and scalable login and registration operations.

1.1.3. Applicant Profile Management

The data model also enables rich Applicant Profile Management by organizing profile data into well-defined groups that mirror real-world applicant information. Education histories, work experience entries, technical skills, and optional portfolio media form the core of the Applicant's structured professional profile. These entities collectively allow the frontend to render a comprehensive and organized view of each applicant. More importantly, the structure supports the Applicant Search feature in the Job Manager subsystem, where companies retrieve Applicant profile information through API usage rather than maintaining a duplicate copy. The model further includes fields that allow the system to emit real-time Kafka events whenever an applicant updates skills or changes their country, supporting the Ultimo-level requirements for instant profile-update notifications and cross-shard migration. All profile-related data is owned by the Applicant Service, which keeps data responsibilities cleanly separated across microservices.

The Applicant subsystem stores portfolio images and videos using MongoDB GridFS, while the Applicant Profile document keeps only the GridFS reference ID and basic metadata (path, MIME type, timestamps). This avoids MongoDB document size limits and keeps large files efficient to retrieve. All applicant media remains in the applicant's regional shard to maintain locality and ensure fast access for both JA and JM via API calls.

1.1.4. Job Search & Matching

To support Job Search and Matching features, the data model integrates with the Job Manager subsystem without duplicating its authoritative data. Instead of storing Job Posts internally, the JA subsystem uses a global Skill dictionary and shared reference structures to interpret Job Manager data consistently. Applicant search queries rely on Job Posts obtained directly from the JM API, following the SRS requirements for API Usage. The JobPost structure is maintained entirely on the JM side, which contains information such as job titles, employment types, salary ranges, required skills, and country, which are critical for Full-Text Search, filtering, and location-based retrieval. Although this data is not stored within the JA database, the JA data model supports the storage of job applications referencing Job Posts by ID. This enables the system to maintain a clean boundary where Job Posts remain the property of the JM subsystem, while still allowing JA to support application flows that reference external Job Post IDs.

1.1.5. Job Application Submission

The data model is also structured to fully support Job Application Submission. Application records within the JA subsystem reference both the Applicant and the external Job Post via an ID obtained from the JM API. This design allows the system to maintain internal consistency without duplicating Job Post attributes. The Application model contains fields such as cover letter references, CV file references, timestamps, and status values that allow Applicants to track their application progress while enabling JM to retrieve application information via JA's API Provision endpoints. Because applications are stored in the Applicant's shard, searches and retrieval operations remain efficient, and the data remains localized alongside the Applicant's profile. This arrangement satisfies the SRS requirement that Job Applications be persisted by JA while ensuring that JM reads these records only through well-defined APIs.

1.1.6. Premium Subscription & Payment Data

Premium Subscription and Payment features are supported through subscription-related data structures that handle subscription status, payment transactions, and personalized Job Search Profiles. The Subscription Service microservice owns these entities, allowing Applicants to define their preferred skill tags, salary expectations, employment types, and locations. These preferences are used by the system's Kafka-based matching engine to identify relevant Job Posts in real time. When new Job Posts are created or updated on the JM side, the Applicant's preferences allow the system to compute matches instantly and send

notifications to eligible Premium users. All subscription-related data remains localized inside the Applicant's shard to ensure fast retrieval and efficient matching computation, especially under high-load scenarios.

1.1.7. Cross-Team Data Exchange with Job Manager

Cross-team data exchange between the JA and JM subsystems is carefully designed to avoid duplication and maintain consistent ownership. The JA subsystem consumes Job Manager data such as Company public profiles, Job Posts, and payment authorization responses through JM's APIs, as required by the SRS API Usage specification. Conversely, JM consumes Applicant data such as profile summaries, application records, and authorization validation through JA's API Provision endpoints. This ensures that JA remains the authoritative source for applicant-centric data, while JM remains the authoritative source for company and job-related data. Applications form the only direct connection point between the two subsystems, linking an Applicant to a Job Post without ever replicating the Job Post's attributes inside the JA database. This clear ownership model prevents circular dependencies, maintains data integrity, and ensures the subsystems stay loosely coupled but fully interoperable.

The Job Manager subsystem uses MongoDB GridFS to store Company Media, with documents containing only GridFS references and metadata. This prevents large promotional images or videos from bloating documents and supports efficient retrieval through signed URLs. Their media is stored in the JM-owned MongoDB cluster rather than applicant shards, ensuring clean ownership and preventing cross-domain duplication. [1]

1.1.8. Support for Microservices and Sharding (Ultimo)

Finally, the overall structure of the data model is aligned with the microservice architecture and the sharding strategy defined for the project. Each microservice owns its corresponding entities Authentication for credentials, Applicant Service for profile data, Application Service for job applications, Subscription Service for premium features, which ensuring clean domain separation. All user-owned and region-dependent data is stored in shards determined by the country shard key, enabling high scalability and resilience. Kafka events include country and entity identifiers, allowing consumers to route operations to the correct shards instantly. This sharded, microservice-aligned data model achieves high scalability by distributing regional load, high resilience by isolating shard failures, and high performance through localized data access, all while maintaining minimal duplication and efficient communication across services.

2.2. Sharding

We apply sharding in our backend to satisfy the Ultimo requirement for scalable and location-based search. Our system uses **Country** as the shard key for both Job Applicant and Job Manager subsystems. This aligns with the SRS, which requires all Applicant and Job Post searches to be routed by country for performance. During registration, each Applicant or Company provides a **country** value. This value determines the shard where their profile, applications, and subscription data will be stored.

Our system shards all user-owned and region-dependent data to ensure scalable performance. This includes the full Applicant profile such as education history, work experience, skill tags, and uploaded media, as well as all job applications and their associated file references. Subscription information and payment records belonging to each Applicant are stored within the same shard so that all related data stays localized. On the Job Manager side, Company profiles and Job Posts are also sharded by the company's country, allowing both subsystems to follow a consistent partitioning strategy. Only global reference data, such as the master skill dictionary or other shared lookups that do not depend on user location, remain unsharded.

To select the correct shard, the backend uses the Applicant's or Company's country as the routing key. For authenticated operations, the country value is embedded in the encrypted JWE token issued during login,

allowing the backend to automatically route each request to the appropriate shard. For search operations, the frontend sends an explicit country filter, and when none is provided, the backend defaults to Vietnam as required by the SRS. A simple Shard Router component maps the country to its corresponding database shard, ensuring that all queries and updates are performed only within the correct partition.

When an Applicant changes their country, the system performs a controlled migration of their records from the original shard to the new shard. This keeps all Applicant owned data aligned with the chosen shard key and maintains consistency with Ultimo requirements for country based data distribution.

3. Squad Container Diagram

3.1. Job Applicant Frontend Container Diagram

[Frontend Container.png]

This container diagram illustrates how the front-end of the system is modularized to support scalability, maintainability, and team-based development. It defines major UI containers, their responsibilities, interactions, user entry points, and integration with the back-end system.

3.1.1. Frontend Technology Stack

The frontend is implemented using a modern JavaScript stack:

- Framework: React
- Language: TypeScript (for type safety and maintainability)
- Routing: React Router (Browser Router)
- State Management: Redux Toolkit and React Context (for global and cross-cutting state)
- HTTP Client: Axios (wrapped by a custom HttpUtil layer)
- UI Layer: Component-based React UI with a shared library of reusable components (buttons, inputs, modals, tables, etc.)
- Styling: CSS Modules / SCSS (and/or a utility-first CSS framework such as TailwindCSS, if adopted)
- Form Handling & Validation: React Hook Form / custom hooks

This stack is chosen to support modular development, strong typing, predictable state flow, and clear separation between presentation, logic, and data access.

3.1.2. User Context

The architecture supports two major user groups:

- **Applicant** – explores job opportunities, submits applications, updates personal profiles, and monitors application status.
- **Admin** – manages job postings, user accounts, and operational oversight.

These actors access the system through browser-based interfaces connected to dedicated feature modules.

3.1.3. Core Containers and Responsibilities

Container	Responsibilities
Browser Route	manages high-level navigation across the application. It maps URL paths to specific modules, enabling page transitions, role-based access, protected routes, and deep linking.

Main	initializes the application environment, loads global settings, injects shared providers, and establishes the user interface foundation. It orchestrates routing, shared state hydration, and overall application rendering
Page	Pages act as structural wrappers for visual screens. They consolidate relevant UI components, manage page-level logic, and define presentation layouts consumed by feature modules.
Layout	centralizes visual consistency across the platform. It manages navigation bars, headers, footers, menus, and UI frameworks to ensure a cohesive user experience.
State Management	maintains shared application state through Redux, Context API, or equivalent libraries. It stores authentication status, user data, UI settings, and transient business objects, enabling predictable data flow and synchronized updates across modules.
Assets	stores static resources including images, icons, fonts, and style sheets. This promotes maintainability and reduces redundancy across feature modules.
Security	enforces access control and session validation. It manages authentication checks, route guarding, token verification, and unauthorized access redirection before users reach protected modules.
Reuseable Components	share UI component library provides standardized interface elements such as buttons, modals, form controls, tables, and alerts. This ensures brand consistency, reduces duplication, and accelerates development.
URL config	centralizes application routing paths and back-end service endpoints. It improves resilience against breaking changes and simplifies refactoring and environment switching.
HttpUtil	manages HTTP requests, token injection, response transformation, error handling, and retry logic—ensuring secure, reliable, and standardized API consumption.

Table: Core Containers and Responsibilities of Job Applicant Front-end Container

3.1.4. Feature Modules

The frontend follows a modular, component-based architecture structured around domain feature modules. Each feature module (e.g., Job Listings, Profile, Application Dashboard) is implemented using a standardized component-level pattern:

- **UI (Presentation Components):** Responsible for rendering the user interface, handling user interactions, and displaying data.
- **Hooks (Logic Layer):** Custom React hooks encapsulate stateful logic, orchestration, side effects, and interaction with services (e.g., useJobListings, useProfile).
- **Services (Data Access Layer):** Service objects or functions encapsulate all API calls to the backend via HttpUtil, handling endpoints, request/response mapping, and error handling.

This structure supports high cohesion within each module and low coupling between modules, making the codebase easier to maintain, test, and extend.

Feature Module	Description
Registration	supports new account creation and identity onboarding.
Login	authenticates system users through credential validation.
Notification	delivers system alerts, application updates, and communication prompts.
Profile	enables applicants to manage personal and professional information.

Account Management	provides administrative oversight of system users and permissions.
Job Listings	displays available employment opportunities and supports search and filtering.
Application Form	facilitates job application submission processes.
Application Dashboard	presents applicant progress, statuses, and historical records.
Payment	manages subscription or transactional activities.

Table: Feature Modules of Component Container

3.1.5. Frontend–Backend Communication

The frontend communicates with the backend through HTTP(S) requests to RESTful endpoints:

- **Request Construction**
  - Feature modules call their corresponding service layer functions (e.g., jobService.getJobs(), authService.login()).
  - These services internally use HttpUtil to send HTTP requests to the configured backend base URL and endpoints defined in URL Config.
- **Authentication and Authorization**
  - After successful login, the backend issues tokens (or secure cookies) which are stored and managed by Security / State Management.
  - For protected operations, HttpUtil attaches the appropriate token (e.g., in the Authorization header) so that the backend can authorize requests based on user identity and role (Applicant, Admin).
- **Data Flow**
  - The backend responds with JSON payloads containing domain data (jobs, profiles, applications, notifications, etc.).
  - Service functions parse and normalize the responses, then update State Management via dispatched actions or state setters.
  - Hooks consume this state and pass the data to UI components for rendering.
- **Error Handling and Feedback**
  - HttpUtil interceptors capture errors (e.g., 401, 403, 500) and can trigger global behaviors such as redirecting to login, displaying error toasts, or logging out the user.
  - Feature modules display contextual feedback to the user based on the response.

3.2. Squad Backend Container Diagram

[Backend Container.png]

3.2.1. Backend Technology Stack

Core application stack

- Language & runtime: Java (JDK 17)
- Framework: Spring Boot (REST APIs, dependency injection, configuration)
- Security: Spring Security with JWT & OAuth2 client (for Google SSO)
- Data access: Spring Data JPA / Hibernate
- Databases: Relational DBMS per service (MongoDatabase (with GridFS for media storage), “database-per-service” pattern)

Infrastructure & integration

- API Gateway: Spring Cloud Gateway (or equivalent) for routing, cross-cutting concerns.
- Service Discovery: Spring Cloud Netflix / Eureka (or equivalent) for dynamic service registry.
- Message broker: Apache Kafka for asynchronous, event-driven communication. [2]
- Caching / session store: Redis cluster for tokens, sessions, and frequently accessed data. [3]
- External providers:
    - Google Auth (OAuth2 / OpenID Connect) for SSO
    - VNPay / NAPAS / other PSPs for payments
    - Email API Provider (e.g., SendGrid / Mailgun) for outbound email

3.2.2. API Gateway

The API Gateway functions as the unified entry point for all client interactions. It performs request routing, authentication checks, rate limiting, logging, and load distribution. It abstracts backend complexity from client applications and ensures consistent access control and policy enforcement across the system.

3.2.3. Purpose of Core Backend Containers (Business Microservices)

Job Manager Subsystem (Orange)

Service Component	Purpose
Authentication Service	Handles login, registration, token issuance (JWT), refresh mechanics, session management, and Google SSO integration.
Authorization Service	Validates permissions and enforces access rules for protected resources across microservices.
Notification Service	Sends email or system notifications about job matching for premium Company account.
Profile Service	Maintains employer identity information, company details, recruiter settings, and organizational metadata used throughout the employer domain.
Job Service	Manages the complete lifecycle of job postings, including creation, updates, activation/deactivation, and removal. Ensures that all job-related metadata is validated and stored consistently.
Premium Company Subscription Service	manages the lifecycle of company subscriptions, including upgrading to premium plans
Applicant Search Service	discover and filter Application opportunities based on their preferences, profile data, and desired roles.
Mactching Service	matches both the Job Applicant (application) and Job Manager (Job post) experiences by providing automated, skill-based matching.
Payment Service	Coordinates with the Payment Service to process premium plans for companies. Validates employer entitlements and synchronizes premium status of Job Manager and Job Applicant Account.

Table: Core Backend Container of Job Manager Subsystem

Job Applicant Subsystem (Blue)

Service Component	Purpose
Authentication	Handles login, registration, token issuance (JWT), refresh mechanics, session management, and Google SSO integration.

Authorization	Validates permissions and enforces access rules for protected resources across microservices.
Notification	Sends email or system notifications about job matching for premium applicant account.
Applicant	Stores and manages core applicant information including personal details, skills, experience, and preferences. Acts as the primary identity container for candidate-side functionality.
Application	Handles the end-to-end job application process—submitting applications, tracking status, updating applicant dashboards, and syncing decisions from employers. Supports both regular and premium application flows.
Premium Applicant Subscription	Provides value-added features such as notification for job matching with the skill set of the applicant
Admin	view and deactivate any Job Applicant account or Company account based on their ID or email, include a search function that allows administrators to find Job Applicant, Company, and Job Post by name and be able to view and delete any Job Post from the system.

Table: Core Backend Container of Job Applicant Subsystem

Infrastructure Containers

Infrastructure Containers	**Purpose**
Service Discovery	Maintains the registry of active microservices and enables dynamic routing through the API Gateway.
Kafka Cluster [2]	Provides asynchronous communication via domain events (e.g., ApplicationSubmitted, JobCreated, UserRegistered, PaymentSucceeded).
Redis Cluster [3]	Offers in-memory caching for session tokens, rate limits, OTPs, hot data, and precomputed queries.
Databases (MongoDB)	Each microservice maintains its own isolated relational database to ensure loose coupling and independent scaling.

Table: Infrastructure Containers of Job Applicant

3.2.4. External Providers

External providers complement the DEVision backend by delivering specialised capabilities that are more efficient or secure to outsource to third-party platforms. Each provider integrates with the backend through well-defined interfaces, ensuring modularity, security, and maintainability.

Google OAuth Provider (Authentication SSO): enables secure Single Sign-On (SSO) for users registering or logging in with their Google accounts. This offloads identity verification to a trusted external identity provider, improving user convenience and security.

Backend Interaction:

- The Authentication Service sends token-exchange requests to Google.
- Google returns identity claims (email, name, profile ID), which the backend uses to create or authenticate user accounts.

Payment Providers (VNPay, NAPAS, etc.): providers process financial transactions for premium services purchased by applicants or employers. Instead of managing sensitive financial information internally, the backend delegates payment processing to licensed payment gateways.

Backend Interaction:

- The Payment Service initializes a secure transaction request.
- The user completes payment on the provider's secure portal.
- The provider sends a callback to the backend with the final transaction status.
- The backend updates premium entitlements for both Job Applicant and Job Manager subsystems.

Email API Provider: delivers outbound communication such as as notification for job matching with the skill set of the applicant.

Backend Interaction:

- The Notification Service constructs message payloads.
- Sends HTTP requests to the email provider's API endpoint.
- Receives delivery status updates for monitoring or retry mechanisms.

3.2.5. Backend Communication with Other Subsystems

Communication with Frontend

- Protocol: HTTP/HTTPS + REST + JSON.
- The frontend calls the API Gateway, which forwards the request to the appropriate microservice.
- Authentication tokens (JWT) are attached in headers; the gateway and downstream services validate them via the Authentication/Authorization services.

Communication Between Microservices

- Synchronous:
  - REST calls over HTTP (e.g., Job Applicant service fetching job details from Job Manager; Payment service verifying user identity with Authentication service).
- Asynchronous:
  - Event-driven messaging via Kafka.
  - Examples: PaymentSucceeded → consumed by Job Manager / Job Applicant to enable premium features.

Communication with External Providers

- Google Auth: OAuth2/OpenID Connect redirects; backend exchanges authorization codes for tokens and user profile data.
- Payment Providers (VNPay/NAPAS/etc.): Backend initiates payment sessions, then receives callbacks/webhooks to confirm payment status.
- Email API Provider: Notification Service calls an external email API over HTTPS to send transactional emails.

Communication with Data Stores & Caches

- Databases: Each microservice interacts only with its own database via JPA/Hibernate, enforcing clear data ownership.
- Redis [3]: Accessed by Authentication and other services that need fast state lookups or caching (e.g., sessions, token blacklists, hot job listings).

4. Backend Component Diagram

4.1. Authentication Microservice

[Authentication Service Component.png]
Figure: Authentication Service Component Diagram (Job Applicant)

The Authentication Microservice is responsible for user identity verification, credential management, token issuance, session handling, and external identity provider integration.

Layer	Description
API & Interface Layer	Exposes authentication endpoints and DTOs used by external clients and other microservices. This layer validates requests, normalises responses, and ensures a stable contract for consumers.
Security Layer (Filter & Security Configuration)	Enforces cross-cutting security concerns such as request filtering, token validation, and access rules for protected endpoints.
Business Logic Layer (Authentication Module)	Contains controllers, services, repositories and domain models that implement core authentication use cases (login, logout, registration, token refresh, SSO flows).
Infrastructure & Integration Layer	Includes Kafka configuration, Redis configuration, and data access components that connect the microservice to external infrastructure such as the relational database, Redis cache, Kafka cluster, and Google SSO provider.

Table: Internal Structure (High-Level) of Authentication Microservice

Incoming Communication	Description
API Gateway / Frontend Clients	Receives HTTP(S) requests for login, logout, registration, Google SSO callback, token refresh, and token validation.
Internal Microservices	May call authentication endpoints (e.g., token introspection or user identity lookup) via internal APIs.
Service Discovery / Platform Runtime	Registers the Authentication Microservice instance and health checks it to maintain service availability.

Table: Incoming Communication of the Authentication Microservice

Outgoing Communication	Description
API Gateway / Other Microservices	- Returns authenticated responses including JWT access/refresh tokens, user identity information, and error messages for failed authentication - Provides validated identity and claims that downstream services (e.g., Authorization, Applicant) use for authorization and domain logic.
Database	Reads and writes persistent authentication data such as user accounts, encrypted passwords, roles, SSO bindings, and audit records.
Redis Cluster	Stores and retrieves short-lived data including session information, token blacklists, OTPs, and rate-limiting counters to improve performance and security.
Kafka Cluster	Publishes authentication-related events (e.g., UserRegistered, LoginSucceeded, LoginFailed) so that other microservices can react asynchronously (auditing, notification, analytics).

Google SSO Provider	Initiates and completes OAuth 2.0 Connect flows for Google login, exchanging authorization codes for tokens and retrieving basic profile information.
Service Discovery / Monitoring	Sends registration, heartbeat, and health information so the platform can route traffic, monitor availability, and scale instances.

Table: Outgoing Communication of the Authentication Microservice

4.2. Authorization Microservice

[Authorization Service Component.png]
Figure: Authorization Service Component Diagram (Job Applicant)

This diagram represents the authorization microservice within the backend microservices system. It indicates the requests sent through API Gateway, secure by security filters, and directly process with Authorization Module, which related with Authentication, Notification, Premium Applicant Subscription and Applicant Services, Redis, Kafka and Database. The Service is split as Internal and External Components to safely proceed and connect with APIs.

Layer	Description
API & Interface Layer	Exposes authentication endpoints and DTOs used by external clients and other microservices. This layer validates requests, normalises responses, and ensures a stable contract for consumers.
Security Layer (Filter & Security Configuration)	Enforces cross-cutting security concerns such as request filtering, token validation, and access rules for protected endpoints.
Business Logic Layer (Authorization Module)	Contains controllers, services, repositories and domain models that implement core authorization use cases (send request to get applicant's data, premium applicants can have access to exclusive features)
Infrastructure & Integration Layer	Includes Kafka configuration, Redis configuration, and data access components that connect the microservice to external infrastructure such as the relational database, Redis cache, Kafka cluster

Table: Internal Structure (High-Level) of Authorization Microservice

Incoming Communication	Description
API Gateway / Frontend Clients	The API Gateway gets client requests, validates, and navigates to respective services.
Internal Microservices	Backend services perform business core logic and data processing. They communicate with Authorization Service to gain access to application before executing any operation
Service Discovery / Platform Runtime	Register service name and Ips, update Kafka, ensure the system can scale and cope with modification in service

Table: Incoming Communication of the Authorization Microservice

Outgoing Communication	Description
API Gateway / Other Microservices	Serve as entry point for client and between services communication. Initially request JWT token to the authorization service for validation and access control

Database	Store persistent data such as token type, token ID, auth ID
Redis Cluster	Execute fast, in-memory caching for session tokens. Reduce latency and manage token expiry
Kafka Cluster	Enable asynchronous event-driven communication
Service Discovery / Monitoring	Sends registration, heartbeat, and health information so the platform can route traffic, monitor availability, and scale instances.

Table: Outgoing Communication of the Authorization Microservice

4.3. Admin Microservice

[Admin Service Component.png]
Figure: Admin Service Component Diagram (Job Applicant)

This diagram handles the role of the Admin, where they can access Job System. The Admin Service is responsible for managing the entire system, service configuration and controlling the application from both Job Applicant and Job Manager, can view and remove any Job Post from system.

Layer	Description
API & Interface Layer	Exposes authentication endpoints and DTOs used by external clients and other microservices. This layer validates requests, normalises responses, and ensures a stable contract for consumers.
Security Layer (Filter & Security Configuration)	Enforces cross-cutting security concerns such as request filtering, token validation, and access rules for protected endpoints.
Business Logic Layer (Admin Module)	Contains controllers, services, repositories and domain models that implement core admin use cases (get applicant profile and detail, update or delete profiles)
Infrastructure & Integration Layer	Includes Kafka configuration, Redis configuration, and data access components that connect the microservice to external infrastructure such as the relational database, Redis cache, Kafka cluster

Table: Internal Structure (High-Level) of Admin Microservice

Incoming Communication	Description
API Gateway / Frontend Clients	The API Gateway receive requests, validates and navigates from Admin Dashboard to execute business logic of Admin Service
Internal Microservices	Backend services such as Applicant Service, Job Service, Profile Service,… They directly interact with Admin Service to provide data or receive updating actions from Admin
Service Discovery / Platform Runtime	Register service name and Ips, update Kafka, this component allows the Admin Service can communicate with other services

Table: Incoming Communication of the Admin Microservice

Outgoing Communication	Description
API Gateway / Other Microservices	The Admin Service sends and receives data back via API Gateway to frontend clients. It can connect with other services to perform CRUD
Database	Store persistent data such as adminId, email, passwordHash
Redis Cluster	Store session tokens, cached admin queries. Reduce latency and manage token expiry
Kafka Cluster	Enable asynchronous event-driven communication

Service Discovery / Monitoring	Sends registration, heartbeat, and health information so the platform can route traffic, monitor availability, and scale instances.

Table: Outgoing Communication of the Admin Microservice

4.4. Notification Microservice

[Notification Service Component.png]

Figure: Notification Service Component Diagram (Job Applicant)

This diagram handles receive events from Matching Service get from Job Manager and deliver notification whenever found a matched job by sending to Premium Account Subscription Service. The purpose of using Kafka and Redis and Security package for handling caching and storing message for performance, maintaining and minimizing security risks across internal and external components.

Layer	Description
API & Interface Layer	Exposes authentication endpoints and DTOs used by external clients and other microservices. This layer validates requests, normalises responses, and ensures a stable contract for consumers.
Security Layer (Filter & Security Configuration)	Enforces cross-cutting security concerns such as request filtering, token validation, and access rules for protected endpoints.
Business Logic Layer (Notification Module)	Contains controllers, services, repositories and domain models that implement core notification use cases (notify premium applicants when found a matched job, confirm successful upgrade to Premium Account)
Infrastructure & Integration Layer	Includes Kafka configuration, Redis configuration, and data access components that connect the microservice to external infrastructure such as the relational database, Redis cache, Kafka cluster

Table: Internal Structure (High-Level) of Notification Microservice

Incoming Communication	Description
API Gateway / Frontend Clients	API Gateway forwards requests to Notification Service , including JWT Token for authentication. The Filter handles validating these tokens before gaining access to the core logic performance
Internal Microservices	Services such as Matching Service, Premium Applicant Subscription,… they create events triggering Notification. These event are streamed via Kafka.
Service Discovery / Platform Runtime	Register service name and Ips, update Kafka, this component allows the Notfication Service can communicate with other services

Table: Incoming Communication of the Notification Microservice

Outgoing Communication	Description
API Gateway / Other Microservices	The Notification Service sends and receives data back via API Gateway to frontend clients. It also connects with other related services to send notification
Database	Store persistent data such as notificationId, message, jobPostId
Redis Cluster	Store session tokens, cached notification queries. Reduce latency and manage token expiry
Kafka Cluster	Publish notification events to Kafka topics.
Email API Provider	Send the notification via external Email API to users.
Service Discovery / Monitoring	Update service status and health metrics to the platform. Ensure Notification is discoverable and easily tracking performance for scaling and fault tolerant.

Table: Outgoing Communication of the Notification Microservice

4.5. Applicant Microservice

[Applicant Service Component.png]

Figure: Applicant Service Component Diagram (Job Applicant)

The diagram illustrates the **Applicant Microservice**, which manages applicant account operations and applicant-related functionalities within the DEVision system. All incoming requests are routed through the API Gateway, validated through security layers, and then processed by the internal modules of the microservice.

Layer	Description
API & Interface Layer	Exposes authentication endpoints and DTOs used by external clients and other microservices. This layer validates requests, normalises responses, and ensures a stable contract for consumers.
Security Layer (Filter & Security Configuration)	Enforces cross-cutting security concerns such as request filtering, token validation, and access rules for protected endpoints.
Business Logic Layer (Authentication Module)	Contains controllers, services, repositories and domain models that implement core authentication use cases (Applicant can do basic function of CRUD, can be view and deactivated by Admin, can be searched by Company).
Infrastructure & Integration Layer	Includes Kafka configuration, Redis configuration, and data access components that connect the microservice to external infrastructure such as the relational database, Redis cache, Kafka cluster.

Table: Internal Structure (High-Level) of Applicant Microservice

Incoming Communication	Description
API Gateway / Frontend Clients	The API Gateway receives applicant-related requests from the client, validates authentication tokens, and routes the requests to the internal

	components of the Applicant Service to execute the corresponding business logic.
Internal Microservices	Backend services such as **Admin Service, Application Service, Job Service, Authorization Service** interact directly with the Applicant Service to provide required data, validate applicant actions, or receive updates related to applicant information.
Service Discovery / Platform Runtime	Registers the Applicant Service name and IPs, updates service metadata, and maintains health checks. This component ensures the Applicant Service can reliably communicate with other services within the platform.

Table: Incoming Communication of the Applicant Microservice

Outgoing Communication	Description
API Gateway / Other Microservices	- Returns applicant-related responses such as applicant profiles, application status, validation results, and error messages when processing fails. - Provides verified applicant information and request context for downstream services (e.g., Application Service, Admin Service, Authorization Service) to perform authorization checks and execute their business logic.
Database	Reads and writes persistent applicant data including applicant profiles, contact details, application histories, premium status flags, and audit records related to updates or changes made within the microservice.
Redis Cluster	Stores and retrieves short-lived data including session information, token blacklists, OTPs, and rate-limiting counters to improve performance and security.
Kafka Cluster	Publishes applicant-related events (e.g., ApplicantUpdated, ApplicantCreated, PremiumStatusChanged) so other microservices (Application, Notification, Search) can react asynchronously for workflows such as indexing, sending notifications, or updating dependent records.
Admin / Job / Application / Authorization Services	Communicate with the Applicant Microservice to receive updated applicant information, verify applicant identity, validate permissions, or synchronize data for job applications and account management.
Service Discovery / Monitoring	Sends registration, heartbeat, and health information so the Applicant Microservice can route traffic, monitor availability, and scale instances.

Table: Outgoing Communication of the Applicant Microservice

4.6. Premium Applicant Subscription Microservice

[Premium Applicant Service Component.png]

Figure: Premium Applicant Service Component Diagram (Job Applicant)

The diagram illustrates the **Premium Applicant Subscription Microservice**, which manages premium plan purchases and subscription status for applicants. All incoming requests pass through the API Gateway, are validated by security layers, and then processed by the internal subscription module.

Layer	Description
API & Interface Layer	Exposes authentication endpoints and DTOs used by external clients and other microservices. This layer validates requests, normalises responses, and ensures a stable contract for consumers.
Security Layer (Filter & Security Configuration)	Enforces cross-cutting security concerns such as request filtering, token validation, and access rules for protected endpoints.
Business Logic Layer (Authentication Module)	Contains controllers, services, repositories, and domain models that implement core premium subscription use cases (Applicants can purchase or renew premium plans, subscription status can be validated and updated, and premium information can be accessed by other services such as Authorization or Notification).
Infrastructure & Integration Layer	Includes Kafka configuration, Redis configuration, and data access components that connect the microservice to external infrastructure such as the relational database, Redis cache, Kafka cluster.

Table: Internal Structure (High-Level) of Premium Applicant Subscription Microservice

Incoming Communication	Description
API Gateway / Frontend Clients	The API Gateway receives subscription-related requests from frontend clients, validates authentication tokens, and routes them to the Premium Applicant Subscription Service to execute premium purchase and subscription management logic.
Internal Microservices	Backend services such as **Applicant Service, Authorization Service, Payment Service, Notification Service** interact directly with the Premium Subscription Service to provide required data, validate payment status, or receive updates when a user becomes a premium applicant.
Service Discovery / Platform Runtime	Registers the Premium Applicant Subscription Service name and IPs, updates service metadata, and performs health checks. This component ensures the service remains discoverable and can reliably communicate with other platform services.

Table: Incoming Communication of the Premium Applicant Subscription Microservice

Outgoing Communication	Description
API Gateway / Other Microservices	- Returns applicant-related responses such as applicant profiles, application status, validation results, and error messages when processing fails. - Provides verified applicant information and request context for downstream services (e.g., Application Service, Admin Service, Authorization Service) to perform authorization checks and execute their business logic.

Database	Reads and writes persistent subscription data such as premium status, plan details, payment timestamps, subscription history, and audit logs related to subscription activities.
Redis Cluster	Stores short-lived data including cached subscription details, temporary payment verification values, and frequently accessed premium flags to improve performance and reduce database operations.
Kafka Cluster	Publishes subscription-related events (e.g., PremiumPurchased, PremiumActivated, SubscriptionUpdated) so that other microservices (Notification, Applicant) can react asynchronously, such as sending confirmation messages or updating applicant benefits.
Payment / Notification / Authorization Services	-Validates the payment transaction and confirms successful purchase. -Sends confirmation messages, premium activation alerts, or failure notifications to users. -Updates access permissions and premium-related entitlements when an applicant activates or renews a premium subscription.
Service Discovery / Monitoring	Sends registration, heartbeat, and health information so the Premium Applicant Subscription Microservice can route traffic, monitor availability, and scale instances.

Table: Outgoing Communication of the Premium Applicant Subscription Microservice

4.7. Application Microservice

[Application Service Component.png]

Figure: Application Service Component Diagram (Job Applicant)

The diagram illustrates the **Application Microservice**, which handles job application operations in the DEVision system. All incoming requests are routed through the **API Gateway**, validated by **Security Filters**, and then processed by the internal logic of the microservice.

Layer	Description
API & Interface Layer	Exposes authentication endpoints and DTOs used by external clients and other microservices. This layer validates requests, normalises responses, and ensures a stable contract for consumers.
Security Layer (Filter & Security Configuration)	Enforces cross-cutting security concerns such as request filtering, token validation, and access rules for protected endpoints.
Business Logic Layer (Authentication Module)	Contains controllers, services, repositories, and domain models that implement core application-related use cases. These include creating and updating applications, validating application data, managing application status changes, and providing application information to other services such as Job, Authorization, Applicant, or Notification.

Infrastructure & Integration Layer	Includes Kafka configuration, Redis configuration, and data access components that connect the microservice to external infrastructure such as the relational database, Redis cache, Kafka cluster.

Table: Internal Structure (High-Level) of Premium Applicant Subscription Microservice

Incoming Communication	Description
API Gateway / Frontend Clients	The API Gateway receives incoming requests related to application operations and forwards them to the Application Microservice. Requests go through the External Interface, then into the Controller layer for processing.
Internal Microservices	Other services such as Job Service, Applicant Service, Authorization Service, and Admin Service communicate with the Application Microservice to validate data, trigger application workflows, or retrieve application-related information through the Internal Interface.
Service Discovery / Platform Runtime	The service registers its name and instances, updates metadata, and responds to health checks so that it can be discovered and called reliably by the API Gateway and other microservices.

Table: Incoming Communication of the Premium Applicant Subscription Microservice

Outgoing Communication	Description
API Gateway / Other Microservices	The Application Microservice returns processed results (application updates, validation results, workflow statuses, error messages) back to the API Gateway. It also exposes updated application information to other microservices when requested.
Database	The Repository layer reads/writes persistent application data such as application details, statuses, history, and audit records, using the Model layer for data structure.
Redis Cluster	Through Redis Config, the microservice caches frequently accessed application data or temporary values to reduce database load and improve performance.
Kafka Cluster	Using Kafka Config, the service publishes application-related events (e.g., ApplicationCreated, ApplicationUpdated, ApplicationStatusChanged) so other microservices like Notification or Analytics can process them asynchronously.
Authorization / Applicant / Job / Notification Services	These services pull or receive application-related data, check permissions, trigger notifications, or perform job-related logic based on the state changes processed within the Application Microservice.
Service Discovery / Monitoring	Sends registration, heartbeat, and health information so the Application Microservice can route traffic, monitor availability, and scale instances.

Table: Outgoing Communication of the Premium Applicant Subscription Microservice

5. Frontend Component Diagram

5.1. Login Component Module (Job Applicant)

[Login Component.png]

Figure: **Login Component Module (Job Applicant)**

The Login module enables users to authenticate using traditional credentials or third-party identity providers.

Purpose of Internal Containers

- Login Form (UI Component) presents the login interface, captures credentials, and handles input validation feedback.
- Social Login (UI Component) provides alternative authentication mechanisms such as Google-based login.
- Login Hook manages authentication logic, including form state, validation, and API invocation. It updates the global state upon successful authentication.
- Login Service performs backend authentication requests, using URL Config to determine endpoints and HTTP Util for secure transmission.

Component Interaction

When users submit login credentials, the Login Form or Social Login component triggers the Login Hook, which validates input and calls the Login Service. The service communicates with backend authentication endpoints via HTTP Util. Upon receiving a valid token and user profile, the Login Hook updates State Management, enabling authenticated routing, UI personalization, and session persistence. Visual and functional elements displayed during the process are sourced from Assets and Reusable Components.

5.2. Registration Component Module (Job Applicant)

[Registration Component.png]

Figure: Registration Component Module (Job Applicant)

The Registration module facilitates secure account creation for new users, including social identity onboarding.

Purpose of Internal Containers

- Social Register (UI Component) renders registration forms and supports social identity-based sign-up.
- Register Hook coordinates input validation, asynchronous submission, error messaging, and post-registration flow control.
- Register Service issues registration requests to backend services through HTTP Util, using URLs defined in URL Config.

Component Interaction

When a new user initiates registration, input collected by Social Register is processed by the Register Hook, which performs validation and delegates data submission to the Register Service. The service communicates with backend registration endpoints and receives confirmation responses. If registration is successful, Register Hook may update State Management to initiate auto-login or redirect users to the Login module. Throughout the process, UI consistency is maintained through Assets and Reusable Components.

5.3. Notification Component Module (Job Applicant)

[Notification Component.png]

Figure: Notification Component Module (Job Applicant)

The Notification module provides real-time communication of system updates, alerts, and user-specific messages.

Purpose of Internal Containers

- Notification UI displays notification lists, icons, badges, and alert interfaces embedded within application pages.
- Notification Hook retrieves, filters, and manages notification data, including marking notifications as read.
- Notification Service interacts with backend notification endpoints to fetch, acknowledge, or update notification records using HTTP Util.

Component Interaction

Upon page load or triggered events, the Notification UI requests data from the Notification Hook, which calls the Notification Service to retrieve notification details from backend APIs. The Hook synchronizes retrieved data with State Management to ensure visibility across different areas of the application. Notification UI components then render updated information using Reusable Components and visual Assets. URL Config ensures that the correct notification endpoints are invoked consistently.

5.4. Profile Component Module

[Profile Component.png]
Figure: Profile Component Module (Job Applicant)

The Profile Module handles displaying profile specifications for users.

Purpose of Internal Containers.

- Profile Detail, Profile Update and Profile Avatar handles rendering user's profile information such as name, email and personal data, editing or deleting and displaying the avatar
- The Hook components play a role in loading logics for user profile data, call the APIs, error handling, manage switching between states, validate form, submission logic, moreover, it abstracts asynchronous operations and error handling.
- The Service directly interacts with backend functions to fetch and update, delete user's avatar, detail or profile via HTTP Utils.

Component Interaction

Besides rendering to the Page, the Profile Detail, Profile Update and Profile Avatar get data from the Hook by calling the API from backend. By implementing State Management and Reusable, it retrieves data and displays on multiple dashboards where components rely on. URL config stores all backend endpoints path, and it ensures to locate API routes correctly.

5.5. Application Dashboard Module

[Application Dashboard Component.png]
Figure: Application Dashboard Component Module (Job Applicant)

The Application Dashboard renders the specific details of job applicants and handles data management structure for the dashboard.

Purpose of Internal Containers.

- Applied Jobs, Saved Jobs, Notifications, Profile Card displays the specific data called from the Hook for the Application Dashboard page.
- The Hook components handle loading for data related to UI components, manage state transitions, update information for job saving, and job applying.
- Dashboard Service mainly interacts with backend to get the data, handling performance via connecting with HTTP Utils.

Component Interaction
When the users wish to access the Application Dashboard, all the UIs are loaded and displayed via the Page. The State Management and Reuseable handles load data from the Hooks, where it directly connects with Dashboard Service and continuously executes via URL Config to call the API from backend.

5.6. Application Form Module

[Application Form Component.png]
Figure: Application Form Component Module (Job Applicant)

The Application Form displays the details of a form which applicant uses to apply the job.

Purpose of Internal Containers

- Job Description, Applicant Information, Upload Resume and Cover Letter UI render the data to the Page.
- The Application Form Hook handles loading the data for the UI. Only one Hook demanded 4 UIs since every UI component must stick together to perform the perfect Application Form.
- The Application Form Service directly interacts with backend to get the necessary data through HTTP Utils.

Component Interaction
The Application Form UIs displays on Frontend by loading the Page. The State Management and Reuseable connect with the Application Form Hook to load the essential data rendered for the Form. Implementing URL Config to confirm successful connection with backend through the Service.

5.7. Job Listing Module

[Job Listing Component.png]
Figure: Job Listing Component Module (Job Applicant)

The Job Listing module displays available job posts with essential details such as title, company, location, salary, and employment type.

Purpose of Internal Containers

- The internal UI components: JobCard, SearchBar, Filter, and Pagination, render the job data on the page.
- A single Job Listing Hook handles all data loading and state updates for these components.
- The Job Listing Service communicates with the backend through HTTP utilities to fetch job posts and apply filters.

Component Interaction

- When the Job Listing Page loads, components retrieve data via the Hook, which calls the Service to get job posts from the backend.
- Reusable components update the UI based on search, filter, and pagination actions.
- URL Config ensures proper routing and correct backend connection.

5.8. Account Management Module

[Account Management Component.png]
Figure: Account Management Component Module (Job Applicant)

The Account Management module allows administrators to view user account information and deactivate accounts when necessary.

Purpose of Internal Containers

- The internal UI components consist of Account Info and Account Deactivation, which display account details and provide the deactivation action.
- A single Account Management Hook loads account data and updates component states.
- The Account Management Service communicates with the backend via HTTP utilities to retrieve account details and process deactivation requests.

Component Interaction

- When the Account Management Page loads, both UI components receive data from the Hook, which calls the Service to fetch account information.
- The Hook also triggers the deactivation workflow and returns updated status to the UI.
  URL Config ensures stable routing and proper backend connection during these operations.

5.9. Payment Module

[Payment Component.png]

Figure: Payment Component Module (Job Applicant)

The Payment Module manages the full subscription payment flow, including selecting a plan, submitting payment information, viewing payment history, and handling success or error states.

Purpose of Internal Containers

- The internal UI components: Payment Plan Card, Payment Form, Payment History, Payment Status, Payment Cancel, and Payment Error, render all payment-related data and user actions.
- All components receive their state through a centralized Payment Hook, which loads and synchronizes payment status, history, and form inputs.
- The Payment Service interacts directly with the backend through HTTP utilities to initiate payments, confirm transactions, and retrieve payment records.

Component Interaction

- When the Payment Page loads, the UI components obtain data via the Payment Hook, which calls the Payment Service to fetch the required information.
- State Management and Reusable UI elements support consistent updates across the payment screens.
- URL Config ensures correct routing and stable backend connectivity for all payment requests.

6. Architecture Rationale

6.1. Data Model Justification

The Data Model is a core component of the Job Applicant Subsystem and has been designed to support the modular and distributed nature of our microservices architecture. Below, we provide justifications for how the data model supports the maintainability, extensibility, resilience, scalability, security, and performance of the system.

6.1.1. Maintainability

The data model follows a modular design, with each microservice managing its own domain and associated data. For example, the Applicant Service owns the Applicant data, while the Application Service handles job applications. This clear ownership allows each service to be independently maintained without risk of breaking other parts of the system. Each service has its own database schema (MongoDB (with GridFS for media storage)) and interacts with others through well-defined APIs and Kafka events.

Advantages

- Clear service ownership of data, meaning that only relevant microservices need to know about and update certain data entities.
- Isolated changes: New features or bug fixes in one service's data model don't require changes in other services. For instance, adding a new field to the Application entity doesn't affect the Payment Service or Notification Service.
- Improved testing: Each service can be tested independently, as its own data and database, making unit tests more straightforward.

Disadvantages:

- Cross-service debugging and tracking changes across multiple services can become more challenging as the number of services grows.

6.1.2. Extensibility

The data model has been designed with future-proofing in mind. Each service is decoupled from the others, and new services or features can be added with minimal disruption to existing ones. For instance, adding a new Search Profile for applicants or a Job Posting feature can be done by adding new services, with minimal changes to existing services.

Advantages

- Independent scaling: As the system grows, new features such as advanced search for applicants or job matching can be added by simply introducing new microservices and expanding the data model to support them.
- Modular architecture: New fields or relationships can be added to existing data models without affecting the functionality of other services. For example, a new Rating system for applicants can be added in the Applicant Service without requiring changes to other services.
- Loose coupling: Each service communicates via APIs and events, which reduces the complexity of adding new features.

Disadvantages

- Introducing new services requires careful coordination to ensure API contracts and data consistency are maintained across all services.
- API versioning can become complex if backward compatibility is not carefully handled.

6.1.3. Resilience

The data model supports resilience by utilizing event-driven communication and asynchronous processing via Kafka. This allows services to be fault-tolerant and handle failures gracefully. If a service like Authentication Service fails, the Applicant Service can continue operating, as it will receive Kafka events indicating state changes in a fault-tolerant way. Similarly, data consistency is ensured by ensuring eventual consistency with Kafka and Redis caches for high-priority data.

Advantages

- Eventual consistency: Kafka allows services to be resilient to temporary failures by replaying missed events. This ensures that eventual consistency is maintained between services.
- Fault isolation: Each service manages its own data, meaning that failures in one service (the Profile Service) do not affect others (the Notification Service).
- Redundancy: The system can retry operations on failure and leverage eventual consistency through Kafka, ensuring that updates are processed even if one service temporarily fails.

Disadvantages

Eventual consistency might introduce slight delays in the propagation of updates between services (a profile update might not immediately appear across all services). This can be managed with event retry mechanisms and circuit breakers to ensure better reliability.

6.1.4. Scalability

The data model is optimized for scalability by employing sharding and independent scaling of each microservice. For example, the Applicant Service can be scaled independently to handle high loads of resume uploads, while other services such as the Payment Service can remain unaffected. Sharding by country ensures that data is distributed effectively across different database clusters, reducing load and latency.

Advantages

- Horizontal scaling of services means that individual microservices can scale based on demand. For example, the Notification Service can handle more notifications without affecting the Application Service.

- Sharding improves performance by limiting the amount of data each service needs to interact with. For example, applicants from different regions (countries) are placed in separate database shards, ensuring fast and scalable access.
- Event-driven scaling: Kafka allows you to scale the event consumers dynamically, adding more event listeners as the system grows.

Disadvantages

- Sharding introduces complexity in managing and coordinating data across different database clusters.
- Cross-shard operations can be complex to handle, especially in global queries (querying all applicants across countries).

6.1.5. Security

The data model is secured through role-based access control (RBAC) implemented in the Authorization Service. Sensitive fields such as passwordHash or ssoId are never exposed to external systems or the frontend, ensuring that data breaches are minimized. Additionally, API Gateway and JWT tokens are used for secure communication and authorization across services.

Advantages

- Fine-grained access control ensures that only authorized users or services can access specific data. For example, an Admin can access all applicant data, while an Applicant can only view their own profile.
- Sensitive data protection is ensured by not exposing sensitive attributes like passwordHash or ssoId in API responses or external DTOs.
- JWT tokens are used to securely authenticate and authorize users across services, ensuring that only trusted users can access protected resources.

Disadvantages

- Managing JWT expiration and revocation can become complex as the system scales.
- Distributed security management across microservices requires additional infrastructure and careful handling of tokens and keys.

6.1.6. Performance

Caching and event-driven architecture significantly improve performance by reducing redundant operations. Redis is used to cache frequently accessed data, such as applicant profiles, and Kafka ensures low-latency messaging between services. The system is designed to minimize blocking operations, and service-specific databases ensure that each service is optimized for its data needs.

Advantages

- Caching with Redis significantly reduces latency, especially for frequent read operations such as profile lookups.
- Kafka allows asynchronous processing, meaning services can work in parallel without waiting on each other, improving system throughput.
- Service isolation enables independent optimization of each service, so slow services like Search Profiles do not impact core services like Applicant Service.

Disadvantages

- Cold starts in services or cache misses in Redis can result in a temporary performance lag.
- Event-driven systems can introduce latency if not properly optimized (especially with large event queues).

6.2. Frontend Architecture Justification

The DEVision frontend adopts a modular, component-based architecture built with React, TypeScript, reusable UI libraries, centralized state management, and structured communication layers (hooks and services). This architectural approach was intentionally selected to support long-term sustainability, team collaboration, and system evolution. The following sections justify the architecture based on key software quality attributes.

6.2.1. Maintainability

The frontend architecture emphasizes separation of concerns—UI, hooks, and services exist as independent layers within each feature module. Shared utilities such as HTTP Util, URL Config, and Reusable Components further reduce duplication.

Advantages

- Easier debugging and onboarding: Developers can quickly locate issues—for example, login validation errors are isolated within the Login Hook rather than spread across UI components.
- Low refactoring effort: Styling changes can be made in UI components without modifying service logic.
- Improved testability: Services and hooks can be unit tested independently, enabling automated regression testing.

Disadvantages

- Higher initial structural complexity: Junior developers may find the layered organization overwhelming.
- Requires strong coding conventions: Without consistent documentation and reviews, modules may diverge in structure.

DEVision Context Example: Future developers modifying the Notification system only need to update the Notification Hook or Service rather than scanning multiple modules.

6.2.2. Extensibility

Feature modules in DEVision (Login, Registration, Job Listings) operate independently and follow a repeatable architectural pattern. New features can reuse shared infrastructure.

Advantages

- Future functionality can be added easily: For example, implementing LinkedIn OAuth would only require adding a new SocialLogin UI component and extending the existing Login Hook and Service.
- Minimal ripple effects: Existing authentication or routing logic remains untouched.
- Supports iterative, Agile development: Teams can develop independent modules in parallel.

Disadvantages

- Risk of module proliferation: Without governance, too many modules can increase maintenance overhead.

- Cross-module data sharing may require additional state management rules.

DEVision Context Example: A future "Company Analytics Dashboard" can be introduced without refactoring Login, Registration, or Notification modules.

6.2.3. Resilience

Although frontend resilience heavily depends on backend stability, the chosen architecture contributes to fault tolerance through centralized error handling and state synchronization.

Advantages

- HTTP Util provides unified error handling: Ensures that API failures trigger consistent UI responses (toast messages, retries).
- Graceful degradation: UI can continue functioning using cached or previously stored state.
- Network failures contained: Hooks can implement retry strategies or fallback UI.

Disadvantages

- Limited ability to recover from critical backend outages.
- Client-side caching must be managed carefully to avoid displaying stale information.

DEVision Context Example: If the Notification Service API fails, only the notification UI is affected, not Login, Registration, or Job Listings.

6.2.4. Scalability

The component-based architecture enables responsive scaling as the user base and functional requirements grow.

Advantages

- Supports UI-level scaling: Components can be lazy-loaded to reduce initial bundle size.
- Team scalability: Multiple developers can independently build separate modules.
- Performance scaling through state optimization: Redux selectors prevent unnecessary re-renders.

Disadvantages

- Overuse of global state may lead to performance bottlenecks.
- Too many nested components can increase rendering complexity.

DEVision Context Example: As DEVision grows to thousands of applicants, modular state slices prevent the entire UI from re-rendering due to a single notification update.

6.2.5. Security

Security responsibilities are shared across frontend and backend, but the frontend architecture includes secure client-side access control.

Advantages

- Route protection via Security container: Prevents unauthorized access to protected screens.
- Token handling centralized in HTTP Util: Reduces vulnerability to inconsistent authentication logic.
- Limited exposure of sensitive logic: Business rules remain on the backend.

Disadvantages

- Tokens stored on client-side remain vulnerable if not secured via HTTP-only cookies.
- Frontend can only enforce superficial access control - backend validation is still mandatory.

DEVision Context Example: Applicants cannot access administrative dashboards because routing and state-based authorization restrict access at the UI level.

6.2.6. Performance

The frontend architecture optimizes rendering and API communication efficiency.

Advantages

- Selective component re-rendering: Hooks and memoization reduce unnecessary UI updates.
- Shared HTTP Util reduces redundant requests: Common caching, throttling, or debouncing can be applied globally.
- Code splitting: Large modules can be loaded only when needed, improving initial load speed.

Disadvantages

- Increased architectural overhead if not used wisely: Excessive abstraction may slow development.
- Client-side performance still depends on network reliability.

DEVision Context Example: Login Form loads instantly, while features like Notification fetching occur asynchronously after authentication, improving perceived performance.

6.3. Backend Architecture Justification

The Backend of DEVision strictly follows the flow of Microservice Architecture developed by Java Spring Boot with components supporting the operation of the system. This architecture helps in maintaining and scaling the system for future development. With usage of external systems such as Google Auth, Email API and connected with Redis Cluster, Kafka, MongoDB (with GridFS for media storage), it can support the enhancement of the DEvision system. The following specifications clearly indicate their outcomes and drawbacks of chosen architecture.

6.3.1. Maintainability

With the chosen microservice architecture, the system now can be updated, debugged or improved through development. The architecture advocates isolating modification on other services when a service confronts error without affecting the whole system.

Advantages

- Easily checking and isolating faults
- Each person can work independently on each microservice component
- Each microservice is small and clear, easier to understand and maintain

Disadvantages

- Cross-service debug is more complicated compared with Modular Monolith
- Requires strong DevOps since Microservice runs on external system: Kafka, Redis
- More management parts, such as more Repositories, Services, therefore, take more time to maintain

DEVision Context Example: Notification Service needs a new feature to send SMS via email. The system can update without touching other services such as Authorization Service or Application Service. By applying Microservice, the system shall remain visible, straightforward debugging and minimizing code complexity

6.3.2. Extensibility

By generating Microservice Architecture, it supports the ability to add new features, services and integration with minimal risks.

Advantages

- Add new microservice without affecting other microservices
- External API such as Google Auth or Email API can be swappable since it is loosely coupled
- Allow dynamic behaviour by developing service discovery

Disadvantages

- API and DTO across service might complex
- If there is no connection, it might cause duplication since each service develops independently
- Cooperation and updating interface are demanded when adding features to enlarge multiple services since each service connect independently via API contract

DEVision Context Example: Suppose the Recruiter Service that allows companies to post jobs and view applicants. The system can build this feature independently, connect it through API Gateway and locate it via Service Discovery. Since the service can be easily added new features, services, however, API versioning between services must be carefully followed.

6.3.3. Resilience

The backend system can easily be rebuilt and recovered from malfunction through applying Microservice Architecture.

Advantages

- Isolation service prevents the failure of whole system
- Redis and Kafka handles fault-tolerant solutions for caching and messaging
- Health checks can be implemented for each service

Disadvantages

- Late responses between services might affect user experience
- Demand fallback logic and circuit breakers to handle partial service failure
- Eventual consistency can cause short-term mismatching on data

DEVision Context Example: If an external provider such as Google Auth fails to connect, the Notification Service can handle recall messages in Kafka or Redis until Google Auth normally works again. According to Service independence, Redis caching and service discovery, resilience can be significant, however, handling on partial service failure and eventual consistency demand replaced strategies.

6.3.4. Scalability

With Microservice Architecture, the system can scale independently, therefore, a service can improve without affecting others.

Advantages

- Redis and Kafka support on horizontal scaling
- API Gateway and Service Discovery allow dynamic routing and load balancing
- Service can scale independently based on load

Disadvantages

- The service is fragmented; however, shared infrastructure can cause bottlenecks. For example: Redis is overload, memory full.
- A request runs through each service; therefore, it requires distributed tracing monitor to help tracking performance flow
- Scaling many services demands excessive cost and complexity

DEVision Context Example: During peak hiring season, Applicant Service might be overloaded due to thousands of Resume uploads. The system can horizontally scale that service while other services remain unchanged. Scalability is extraordinarily strong in Microservice Architecture; however, when a shared infrastructure such as Redis Server confronts overloading, it might cause bottlenecks and decrease the system loading.

6.3.5. Security

With Microservice, it protects data and accesses between services via external interface and secure communication

Advantages

- API Gateway filters and validate token requests prior to accessing business logic
- External APIs are isolated and hardened, for example, Google Auth, Email Provider API
- Authentication via JWT token enhance consistent access control

Disadvantages

- Each service will be served and secured independently
- More area for hacking due to multiple endpoints.
- Too many tokens might be difficult when managing and recalling

DEVision Context Example: When a user logs in via Email Provider API, the Authentication Service creates a JWT, then the Authorization Service checks if the user has already gained permission to access the platform. Unauthorized users are blocked at the API Gateway before reaching the dashboard. Security is enforced by validating JWT tokens, API Gateway filtering, however, it might be complex on managing token and secure multiple endpoints

6.3.6. Performance

Building the system with Microservice Backend logic improves fast and efficiently for the system. With Redis caching, Kafka config and service isolation, they decrease redundancy and latency of the system

Advantages

- Redis caching and Kafka reduce latency and improve system performance
- Service can be optimized individually. For example, each database handles each service, as a result, a query might run faster and not affect on other databases.

- By calling External API, those providers such as Google Auth, Email Provider API take care of heavy tasks and improve the flow of the system

Disadvantages

- Microservice might cause network overhead since it calls through many services inside that component
- Cold starts and cache misses might slow down the initial requests if a service instance is not running
- Adjusting performance demands deep knowledge of other external services. For example, if the system encounters errors, it might be difficult to trace the issue root.

DEVision Context Example: The Applicant Service caches always accessed profile in Redis. Instead of querying DB every time, the system calls it from Redis. This decreases the latency and enhances response. Performance in Microservice might be helpful with Redis caching, Kafka messaging and database independence, thus, the latency might happen if inter-service network calls and cache misses are not handled properly.

7. Conclusion

In Conclusion, the Job Applicant subsystem of DEVision is offered as an extensively featured, scalable, and security-conscious platform that helps applicants go through the entire recruitment process, including registration and creation of profiles, job search, job application, and premium real-time job matching. The system makes use of the latest architectural trends like dynamically patterned monoliths or layers at the lower levels and microservice, sharded databases, Redis, and Kafka at higher levels. These have good extensibility, maintainability, security and scalability benefits which make the platform scalable to accommodate growing user demands.

But there are also significant challenges that are brought by the architecture of the system. The engineering discipline required on multi-layered structures, module strictness, SSO integrations, tokens, and cross shard data migration is complex. Also, the use of Kafka, Redis, and distributed deployment leads to increased overhead and risk of operations. These shortcomings notwithstanding, the system is still strong and quite compatible with the current software engineering practice. It provides a potent and future-proof platform in the DEVision ecosystem when unified appropriately to bridge applicants and employers.

8. References

[1] ChatGPT. (2025), OpenAI. Accessed: Nov 15, 2025. [Online]. Available: https://chatgpt.com/

[2] DataCamp. (2025). “Kafka Streams Tutorial: Introduction to Real-Time Data Processing”. Accessed: Nov. 23, 2025. [Online]. Available: https://www.datacamp.com/tutorial/kafka-streams-tutorial?utm_cid=19589720824&utm_aid=186331390509&utm_campaign=230119_1-ps-other~dsa~tofu_2-b2c_3-apac_4-prc_5-na_6-na_7-le_8-pdsh-go_9-nb-e_10-na_11-na&utm_loc=1028581-&utm_mtd=-c&utm_kw=&utm_source=google&utm_medium=paid_search&utm_content=ps-other~apac-en~dsa~tofu~tutorial~kafka&gad_source=1&gad_campaignid=19589720824&gbraid=0AAAADQ9WsE3I1TwpBHV-6WDO18x-SJyE&gclid=CjwKCAiAraXJBhBJEiwAjz7MZShS9QCxQ3qRcbwk-oec7vrZGky5iMAwLGwUy8sw6D6fzRQwpyBpjhoCEEMQAvD_BwE

[3] GeeksforGeeks. (2025). “Complete Guide to Redis (Java) – System Design”. Accessed: Nov. 23, 2025. [Online]. Available: https://www.geeksforgeeks.org/system-design/complete-guide-to-redis-java/

[4] Cybeready. (2025). “The 7 Pillars for Zero Trust: An In-Depth Guide”. Accessed: Nov. 26, 2025. [Online]. Available: https://cybeready.com/the-7-pillars-for-zero-trust-an-in-depth-guide/