

v

# **EEET2582/ISYS3461 | Software Engineering: Architecture and Design**

**School of Science, Engineering & Technology**

 RMIT  
UNIVERSITY

## **Architecture Documentation**

**DEVision - Job Manager Subsystem**

### **Hercules Squad**

#### **Team Members**

**Member 1** - Nguyen Trong Khoa - S3978477

**Member 2** – Duong Phu Dong - S3836528

**Member 3** – Ho Quang Huy - S3924729

**Member 4** – Nguyen Hong Anh - S3924711

**Member 5** – Vu Thien Minh Hao - S3938011

**Teacher: Dr Tri Huynh**

**Submission Due Date: 6:00 PM 28<sup>th</sup> November 2025**

"We declare that in submitting all work for this assessment, I have read, understood, and agree to the content and expectations of the Assessment declaration."

## Table of Contents

<b>1. Introduction .....</b>	<b>4</b>
<b>2. Key Features Supported .....</b>	<b>4</b>
2.1. Company registration, activation, and secure login .....	4
2.2. Company profile management .....	4
2.3. Job post lifecycle management.....	4
2.4. Applicants search and filtering .....	4
2.5. Subscription and payment .....	4
2.6. Real-time notifications .....	4
2.7. Microservice-based backend with Redis token revocation .....	4
2.8. Sharded databases based on country .....	5
2.9. Integration with the Job Applicant subsystem .....	5
<b>3. Squad Data Model.....</b>	<b>5</b>
3.1. Company onboarding, authentication and access control.....	5
3.2. Company profile management .....	5
3.3. Job Posting.....	5
3.4. Applicant searching .....	6
3.5 Premium subscription, headhunting, payment and billing.....	6
3.6 Notifications and cross-service events .....	6
3.7 Cross-team data exchange (JM >< JA) .....	6
<b>4. Sharding Strategy.....</b>	<b>7</b>
4.1. Sharding Goal.....	7
<b>5. Squad Container Diagrams .....</b>	<b>7</b>
<b>6. Backend Component Diagrams .....</b>	<b>9</b>
6.1. Component List.....	9
6.2. General Microservice Flow for Job Manager .....	11
<b>7. Incoming/Outgoing Communications.....</b>	<b>13</b>
7.3.1 Authentication Microservice .....	13

7.3.2 Payment Microservice.....	13
7.3.3 Matching Service .....	14
7.3.4 Subscription Service .....	14
7.3.5 Applicant Search Service .....	15
7.3.6 Authorization Microservice .....	15
7.3.7 Notification Microservice.....	16
7.3.8 Job Post Service .....	17
7.3.9 Profile Service: .....	17
8. Frontend Component Diagrams .....	18
8.1. Auth components – Login Component & Registration Component.....	18
8.2. Profile components Profile components.....	19
8.3. Job Post components .....	20
8.4. Applicant Search components .....	21
8.5. Payment components .....	23
9. Architecture Rationale .....	24
9.1. Data model.....	24
9.1.1. Maintainability .....	24
9.1.2. Extensibility.....	24
9.1.3. Resilience.....	25
9.1.4. Scalability.....	25
9.1.5. Security .....	26
9.1.6. Performance .....	27
9.2 System Architecture .....	27
10. Conclusion .....	29

## 1. Introduction

This report presents the Data Model, System Architecture, and associated design justifications for the DEVision Job Manager subsystem. In details, we dedicate how core data entities, backend microservices, frontend components, and supporting infrastructure are defined and integrated to satisfy the functional, quality, and scalability requirements specified in the Squad Software Requirements Specification (SRS). However, the scope of this report aims directly and only to the Job Manager side of the DEVision platform, including interfaces with the Job Applicant subsystem, external payment and Authentication API Provider services. In brief, the report is intended to serve as a single, authoritative reference for project stakeholders, engineering teams, and future maintainers when implementing, reviewing, or extending the Job Manager subsystem. And to ensure

## 2. Key Features Supported

The Job Manager subsystem provides the following core capabilities to support company-side recruitment workflows:

### 2.1. Company registration, activation, and secure login

The subsystem allows companies to register accounts, verify their identity through activation flows, and authenticate using secure mechanisms such as JWS/JWE tokens and supported Single Sign-On (SSO) providers.

### 2.2. Company profile management

Authorised company users can maintain a structured organisation profile, including contact information, branding assets, and descriptive content that is exposed to potential applicants via the Job Applicant subsystem.

### 2.3. Job post lifecycle management

Recruiters can create, edit, tag, publish, and archive job posts. Each job post records key attributes such as title, description, salary range, employment type, required skills, and expiry date to support search and matching.

### 2.4. Applicants search and filtering

The subsystem supports searching for applicants using filters (e.g., location, education, experience), full-text search, and technical skill tags. These capabilities enable recruiters to efficiently identify candidates that match their requirements.

### 2.5. Subscription and payment

Companies can purchase and manage premium plans via integration with third-party payment gateways. Subscription status is tracked to unlock additional features such as advanced applicant search and real-time matching.

### 2.6. Real-time notifications

Using Kafka as the messaging backbone, the subsystem generates and consumes events (e.g., job post updates, new applications, profile changes) to deliver timely notifications to company users.

### 2.7. Microservice-based backend with Redis token revocation

The backend is decomposed into microservices, each responsible for a specific business capability. Redis is used as a central token revocation store to support secure logout and session invalidation across services.

## 2.8. Sharded databases based on country

Persistent data for companies and job posts is partitioned by country, enabling the platform to scale horizontally and reduce query latency for region-specific operations.

## 2.9. Integration with the Job Applicant subsystem

The Job Manager subsystem exposes and consumes APIs to exchange job posts, company public profiles, subscription status, and application-related data with the Job Applicant subsystem, ensuring a consistent end-to-end experience across the platform.

## 3. Squad Data Model

To ensure all the required features from the Software Requirement Specifications (SRSs) can be implemented correctly and consistently, we need to figure out how the data will be stored and connected to our system so that it can support the core functionalities of our application. Thus, an organized and sophisticated Entity Relationship Diagram (ERD) capturing all relationships between each data model is needed. To align with the design of our system architecture, the data models are organized around their respective microservice, as shown in the above figure. Overall, for the JM side, there are 9 microservices, each responsible for a specific feature of the application, including: Company Service, Job Post Service, Applicant Search Service, Premium Company Subscription Service, Matching Service, Payment Service, Notification Service, Authentication Service, and Authorization Service. This keeps the model consistent with the container diagram and supports independent deployment of each service. Due to the size of the diagram, it is difficult to insert and observe the entire system in detail. Thus, the high-resolution version of the Data Model diagram can be viewed in the submission folder via this link: [Data Model link](#). The detailed implementation and explanation of each service will be discussed in the sections below.

### 3.1. Company onboarding, authentication and access control

The Authentication & Authorization services store company credentials, SSO identities and login tokens. Together with the core Company record, they support features such as company registration, login, account locking and permission checks on all management screens. Only opaque tokens and a safe “public company” projection are ever exposed outside these services, which satisfies SRS requirements for security and privacy while still letting other services identify the current company via companyId.

### 3.2. Company profile management

The Company / PublicProfile / CompanyMedia entities supports all employer-branding features in our application. Company stores the stable identity and account state of each organisation; PublicProfile holds the public information that applicants can view while looking for job (“about us”, “who we are looking for”, industry, location); and CompanyMedia adds URLs and metadata for logo and gallery images. This structure lets the Job Manager UI build a company profile page without mixing media details into the core account object. The same public profile data is exposed to the Job Applicant subsystem via REST APIs so that applicants see consistent company information in search results and job-detail views.

### 3.3. Job Posting

The Job Post service implements the main recruitment features: creating, publishing, managing and maintaining job posts. The job posts aggregate store all structured information, including title, description, employment types (full-time, part-time, fresher, internship, contractual), salary type with min/max ranges, and status flags for draft/published/archived. Required skills for each job are expressed through a separate set of records that refer to a shared SkillTag catalogue between two subsystems. Using the same skillId and naming convention as the Applicant side allows both

subsystems to communicate about required skills in a consistent way and enables easier skill-based matching between job requirements and applicant profiles.

### 3.4. Applicant searching

To support the searching for applicants feature, the Applicant Search service keeps the flags for each (company, applicant) pair. These flags help the company to mark their target applicant “FAVOURITE” or “WARNING” and will be considered when search results and application views functions are used. The underlying applicant summaries, skills and resume are pulled from the Applicant subsystem over APIs, so the Job Manager data model only stores what it needs to remember per company.

### 3.5. Premium subscription, headhunting, payment and billing

Premium features in the SRS (headhunting profiles saved for real-time matching, enhanced job visibility) are backed by three main groups of data models:

- The Subscription aggregate tracks the company’s plan (Freemium/Premium), billing period, current status and the last successful payment.
- The SearchProfile model stores reusable “headhunting profiles” defined by the company, including preferred country, salary range, education level, employment types and target skill tags. These records are evaluated whenever applicant-update events arrive from the other subsystem to decide which premium companies should receive a match.
- The PaymentTransaction records capture each subscription payment with amount, currency, gateway and outcome, and are used for billing history and subscription validation.

The Company model also contains a deformatized flag indicating whether the company is premium account or not, so permission checks in other services (for example extra job posting options, access to advanced or automatic search) can be done efficiently without reading the full subscription history.

### 3.6. Notifications and cross-service events

The Notification service stores in-app and email notifications sent to companies, including registration confirmations, subscription reminders, payment transaction and real-time applicant matches. It consumes events from both Job Manager and Job Applicant services (for example “applicant updated”, “application submitted”, “subscription expiring”) and persists a simplified notification payload that the UI can render. This facilitates timely alerts without tightly coupling notification delivery to the source systems.

### 3.7. Cross-team data exchange (JM <> JA)

The data model also supports data exchange with the Job Applicant subsystem. Job Manager exposes:

- public company profile information (from Company/PublicProfile/CompanyMedia),
- public job information and skill requirements (from the Job Post model and SkillTag), and
- payment/receipt information via the Payment service.

In return, our Job Manager will consume these data from the Applicant side:

- applicant summaries, skills and resumes for searching and matching against SearchProfiles, and application submissions that reference companyId and jobId
- Additionally, Applicant-update events used to trigger notifications, and premium matches will also be consumed.

By keeping these shared concepts aligned (IDs, enums and SkillTag structure) while still separating schemas by microservice, our Job Manager data model ensure we support all core features in the SRS—company onboarding, profile management, job posting, applicant search, premium matching, subscription management, payments and notification.

## 4. Sharding Strategy

### 4.1. Sharding Goal

- To scale high-traffic entities and support regional data compliance.

### 4.2. Shard Key

- country from the **Company** entity.
- Rationale:
  - Ensure companies and their job posts stay in the same shard.
  - Supports geographic data locality and reduces cross-region latency.

### 4.3. What Is Sharded

- Company-related data: **Company**, **CompanyProfile**, **CompanyMedia**, **JobPost**, **JobPost\_Skill**, **CompanySearchProfile**, **Subscription**, etc.
- These entities contain `company_id`, which maps to a shard via the country field.

### 4.4. How Backend Determines Which Shard to Query

- Extract `company_id` from the API request.
- Read the company's country from the global index table (or cache).
- Route all queries to the appropriate shard.
- Cross-shard operations aggregate results from all shards.

## 5. Squad Container Diagrams

Though Job Applicant and Job Manager are separated into two container groups, our squad decided to include both businesses domain into one container diagram. The main reason for this is the software system for the application uses one API gateway and Service Discovery for the easier maintaining and configuring of both backend and frontend deployments.

In the diagram, the colours have been chosen to represent separately for the domains within the platform: orange for the Job Manager; and blue for the Job Applicant. The Job Manager subsystem – the focused domain in the report, is implemented using a microservice-based architecture, where each service is responsible for a specific business function. The services work together to deliver the core functionalities defined earlier in the Key Features Supported in section 2.

- **Company Registration and Login:** The Authentication Service is responsible for managing company account creation and login. It integrates with identity providers via Single Sign-On (SSO) and generates secure JWT/JWE tokens for user authentication. The Authorization Service checks the company's roles and verifies token validity (as shown in the arrows between Authentication Service and Authorization Service). This is further supported by a Redis-based token revocation mechanism, ensuring secure session management across all services.
- **Company Profile Management:** The Profile Service manages company profile data, such as contact information, branding assets, and organisational descriptions. This service interacts with the database container specific to the Job Manager cluster and ensures scalability and isolation of profile data. Additionally, the company profile data is exposed to potential applicants via the Job Applicant subsystem, with the Matching Service helping to ensure the profile's visibility for the right applicant matches.
- **Job Post Lifecycle Management:** The Job Service handles the complete lifecycle of job posts, including creation, editing, tagging, publishing, and archiving. Key attributes such as job title, description, salary range, and required skills are captured by this service. The Job

Service communicates with the database container dedicated to Job Manager to store job post data and is responsible for updating the Applicant Search Service with job tags and relevant search metadata. Job Post updates are published to Kafka as events to notify interested systems (such as the Applicant Search Service).

- **Applicant Search and Filtering:** The Applicant Search Service enables recruiters to search for applicants using various filters, including location, education, skills, and work experience. The Matching Service works closely with the Applicant Search Service to match applicants to job posts based on the defined filters and criteria. While applicant data resides in the Job Applicant subsystem, the Applicant Search Service consumes applicant information through APIs provided by the Job Applicant subsystem, allowing for a seamless integration between the two subsystems.
- **Subscription and Payment:** The Subscription Service manages the lifecycle of company subscriptions, including upgrading to premium plans. The Payment Service integrates with third-party payment gateways (e.g., VNPAY, NAPAS) to handle subscription payments. Once payment is successfully processed, the Premium Company Subscription Service updates the company's subscription status to enable premium features, such as enhanced applicant search capabilities and priority job post visibility.
- **Real-Time Notifications:** The Notification Service listens to events published by Kafka (e.g., job post updates, application submissions, profile changes) and sends timely notifications to company users. The notifications ensure that recruiters are immediately alerted about key events that affect job posts or applications, facilitating real-time engagement and decision-making.
- **API Integration with Job Applicant Subsystem:** The Job Manager subsystem interacts with the Job Applicant subsystem through defined REST APIs. These APIs facilitate the exchange of job post information, applicant profiles, application statuses, and subscription data. As shown in the diagram, the Matching Service and Applicant Search Service consume applicant-related data from the Job Applicant subsystem via these APIs, ensuring that job seekers' information is properly integrated into the recruiting workflows of companies using the Job Manager system.
- **Matching Service:** The Matching Service plays a crucial role in matching applicants to job posts. It acts as a middle layer, receiving job post details and applicant data, and uses algorithms or criteria to ensure that the most suitable applicants are recommended to recruiters. The Matching Service works closely with the Applicant Search Service to refine search results and match applicants with the best-fit positions. It is an integral part of the real-time job application process and is critical for providing high-quality matches that benefit both applicants and companies.

For Applicant subsystem (blue) and Admin (yellow):

- **Applicant Service:** The Applicant Service manages applicant profiles, including personal details, resumes, and cover letters. This service allows for CRUD operations on applicant data and facilitates matching with job posts. The Admin Service has the capability to view and manage applicant information, including the ability to access applicant profiles, resumes, and cover letters, as shown by the arrows pointing from Admin Service to Applicant Service.
- **Premium Applicant Subscription Service:** This service manages the lifecycle of premium subscriptions for applicants, providing access to exclusive features such as advanced job matching and priority visibility. Admin Service can interact with this service to update the subscription plan status for premium applicants, as indicated by the arrow from Admin Service to Premium Applicant Subscription Service.
- **Authentication Service:** The Authentication Service is responsible for the login and registration process, primarily using Google OAuth as an identity provider. It generates JWT

tokens and validates authentication requests, ensuring that only authorised users can access the system. The Admin Service interacts with the Authentication Service for user authentication when creating or managing applicant accounts, as shown by the arrow linking Admin Service to Authentication Service.

- **Authorization Service:** The Authorization Service verifies the roles and permissions of users, ensuring that each request is properly authorised before access is granted. This service works closely with the Admin Service, which needs permission checks for accessing and managing applicant data, job posts, and subscription statuses, as shown by the arrows pointing from Admin Service to Authorization Service.
- **Notification Service:** The Notification Service handles real-time notifications to applicants or companies about job posts, applications, and premium subscription updates. The Admin Service can trigger notifications to applicants for successful payments or job matches, as illustrated by the arrows pointing from Admin Service to Notification Service.
- **Applicant service:** Responsible for providing Applicants' CVs and Cover letters, supporting for tasks relative to the Applicant Service.

The Admin Service plays a central role in managing the entire platform, interacting with both Job Manager and Job Applicant subsystems. The Admin Service can:

- Create new applicants via the Applicant Service.
- Manage applicant information, including viewing and editing CVs and cover letters.
- Manage job posts within the Job Manager subsystem.
- Update premium subscription plans for applicants via the Premium Applicant Subscription Service.

This service allows administrative control over applicant data, job posts, and subscription statuses, ensuring that system management remains centralised while maintaining the integrity of the microservice architecture.

## 6. Backend Component Diagrams

### 6.1. Component List

#### Infrastructure Component:

- **API gateway:** Central entry point that routes all client requests to the appropriate microservice
- **Service Discovery:** Registers and locates active microservices dynamically.

#### External Infrastructure Services:

- **Kafka:** Acts as the message broker for event-driven communication between microservices.
- **Redis:** Serves as a high-performance cache and token store.

#### External Third-Party Service Providers:

- **Email API Provider:** External email service used to send account activation links, subscription reminders, and other notifications.
- **VNPAY & NAPAS Payment Gateway:** Trusted payment platforms that handle credit/debit card processing for company and applicant subscriptions.

- **Google OAuth Provider:** Provides Single Sign-On (SSO) authentication for companies registering or logging in using Google credentials.

Component	Description
Controller Layer	Handles incoming HTTP requests, routes them to the appropriate service methods, and returns responses to clients
Service Layer	Contains the business logic for processing requests and coordinating between controllers and repositories.
Repository Layer	Manages data persistence operations such as CRUD queries on the database.
Internal Interface	Defines contracts for communication within the same microservice
External Interface	Defines contracts exposed to other microservices or subsystems. These are used via the API Gateway.
Internal DTOs	Data Transfer Objects used only within the module, typically between layers
External DTOs	Data objects shared across services through external APIs
Kafka Config	Configuration for producing and consuming messages in Kafka topics
Redis Config	Defines Redis connection settings and caching policies.
Security Config	Configures authentication, authorization, and token validation policies for endpoints.
Redis Server	Embedded or external Redis instance for caching and token management

Filters	Request interceptors for logging, validation, authentication, and cross-service communication control.
---------	--

## 6.2. General Microservice Flow for Job Manager

### Entry Point – Client Requests

#### Purpose:

- All requests from frontend or external clients first hit the API Gateway.

#### Responsibilities / Actions:

- Originates from the frontend or external client.
- Forwarded to the API Gateway for initial processing.

### API Gateway

#### Purpose:

- Acts as the central entry point for all requests.
- Stops invalid or malicious requests early and reduces unnecessary load on microservices.

#### Responsibilities / Actions:

##### Performs superficial token validation:

- Checks that a token exists and is correctly formatted.
- Verifies basic role-based access for endpoints (e.g., public vs admin-only).
- Enforces global policies: rate limiting, IP restrictions, request logging.
- Uses Service Discovery to dynamically locate the appropriate microservice instance for routing requests.
- Routes requests to the correct microservice controller.

### Filters / Security Config:

#### Purpose:

- Acts as the second layer of defence within the microservice pipeline.
- Ensures all requests entering the microservice are fully authenticated and technically valid.

#### Responsibilities / Actions:

##### Performs detailed token verification:

- Checks token integrity, signature, and encryption (JWE/JWS).
- Applies service-specific security rules:
  - Endpoint-specific access control.
  - Request validation and logging.

### Controller Layer

#### Purpose:

- Acts as the entry point for a microservice to process incoming requests.
- Translates external requests into the internal format for the service layer.

#### Responsibilities / Actions:

- Receives requests after passing API Gateway and Filters/Security Config
- Converts External DTOs into internal objects for the Service Layer.
- Forwards requests to the Service Layer for business logic execution.
- Handles responses: maps internal results to External DTOs (for inter-service communication) or HTTP/JSON responses (for clients).
- Supports requests from both frontend clients and other microservices.

### Service Layer

**Purpose:**

- Encapsulates **business logic** and coordinates operations between the Controller Layer and the Repository Layer.
- Acts as the main layer where microservice-specific rules, validations, and workflows are applied.

**Responsibilities / Actions:**

- Receives requests from the Controller Layer and processes business logic
- Calls the Repository Layer to retrieve or persist data.
- Applies business rules such as:
  - Checking user permissions via the Authorization Service.
  - Validating complex input beyond basic controller-level checks.
  - Orchestrating multiple operations (e.g., updating multiple entities, triggering events via Kafka).
- Converts internal data to Internal DTOs before sending to Controller, or to External DTOs when responding to another microservice.

**Repository Layer****Purpose:**

- Provides data access abstraction, isolating the Service Layer from database details..
- Handles all interactions with the database (or other persistence systems).

**Responsibilities / Actions:**

- Executes queries or transactions against the microservice's database.
- Maps database entities to internal domain objects.
- Ensures data consistency and isolation within the microservice's bounded context:
- Supports internal and external DTOs for data transfer to Service Layer or other microservices (via External Interface).
- Implements reusable query operations that can be shared across the microservice.
- Manages cache or token-related data through Redis (if applicable):
  - Provides fast, in-memory access to frequently used data, such as authentication tokens, session information, or temporary verification results.
  - Reduces repeated database queries and improves performance for time-sensitive operations.
  - Ensures quick validation of data needed by the Service Layer without impacting the primary database

**Model Layer****Purpose:**

- Represents the core domain objects and their structure within the microservice.
- Encapsulates the state and behavior of entities used in business logic.

**Responsibilities / Actions:**

- Defines entities, value objects, and relationships relevant to the microservice.
- Provides methods or functions that enforce domain invariants (rules that must always hold for the entity).
- Ensures data consistency and isolation within the microservice's bounded context:
  - Serves as the foundation for Repository Layer mapping and Service Layer operations.

- Works only with internal DTOs, keeping internal representation separate from external communications.

## 7. Incoming/Outgoing Communications

This section based on the components diagram of each service to explain the incoming and outgoing communication method of the

### 7.3.1 [Authentication Service](#)

#### Incoming Communications:

- **From Frontend:**
  - **Communication type:** REST API
  - **Description:** The frontend sends registration and login requests (e.g., /register, /login) through the API Gateway and Filter/Security Config for validation and routing before reaching the controller.

#### Outgoing Communication:

- **To Authorization Service:**
  - **Communication Type:** Kafka Event
  - **Description:** Publishes authentication-related events to Kafka, allowing the Authorization Service to verify user roles and manage access control asynchronously.
- **To Google OAuth Provider:**
  - **Communication Type:** OAuth 2.0 Flow
  - **Description:** Handles SSO login by communicating with Google identity providers to exchange tokens and retrieve verified user identities.
- **To Company Profile Service:**
  - **Communication Type:** Kafka Event
  - **Description:** Publishes a CompanyRegistered event to Kafka, which triggers the Company Profile Service to create a new default company profile (regular type).
- **To Frontend (via API Gateway):**
  - **Communication Type:** REST API
  - **Description:** Returns registration, login, or activation responses and authentication tokens to the frontend through the API Gateway.

### 7.3.2 [Payment Microservice](#)

#### Incoming Communications:

- **From Frontend:**
  - **Communication type:** REST API
  - **Description:** The frontend sends payment initiation and subscription purchase requests (e.g., /payment/subscribe, /payment/process)

#### Outgoing Communication:

- **To Authorization Service:**
  - **Communication Type:** Kafka Event
  - **Description:** Publishes payment authorization or validation events to confirm that the company is permitted to make or renew a subscription payment.
- **To VNPAY & NAPAS Payment Gateways:**
  - **Communication Type:** API Integration

- **Description:** Sends encrypted transaction requests to third-party payment providers for credit card or online payment processing, and receives payment confirmation or failure callbacks.
- **To Frontend (via API Gateway):**
  - **Communication Type:** REST API
  - **Description:** Returns payment results, subscription confirmation, or error messages to the frontend after processing the transaction..

### 7.3.3 [Matching Service](#)

#### Incoming Communications:

- **From Frontend:**
  - **Communication type:** REST API
  - **Description:** Companies set or update their matching criteria (e.g., skills, education, experience, country)

#### Outgoing Communication:

- **To Applicant Subsystem:**
  - **Communication Type:** API Call
  - **Description:** Retrieves applicant profile data from the Applicant Subsystem via its provided API or Kafka topic to evaluate matches based on the company's defined criteria.
- **To Notification Service:**
  - **Communication Type:** Kafka Event
  - **Description:** Publishes MatchingApplicantFound events when an applicant meets the company's matching conditions, prompting the Notification Service to send real-time alerts to the company.
- **To Authorization Service:**
  - **Communication Type:** Kafka Event
  - **Description:** Sends verification requests to ensure that the company initiating the matching process has the proper permissions or active premium status to use automated matching features.

### 7.3.4 [Subscription Service](#)

#### Incoming Communications:

- **From Frontend:**
  - **Communication type:** REST API
  - **Description:** Receives requests from companies to check subscription status, view remaining subscription time, or renew subscriptions.
- **From Profile Service:**
  - **Communication type:** Kafka Event
  - **Description:** Receives queries from the Company Profile Service to verify if a company is still a premium subscriber, including checks for expiration dates and subscription type..

#### Outgoing Communication:

- **To Payment Service:**
  - **Communication Type:** Kafka Event

- **Description:** Sends requests to retrieve or validate payment transaction records to confirm successful payments and ensure accurate subscription renewal and expiration management.
- **To Authorization Service:**
  - **Communication Type:** Kafka Event
  - **Description:** Publishes subscription verification events so that other microservices can confirm if the company is authorized to access premium functionalities.

### 7.3.5 [Applicant Search Service](#)

#### Incoming Communications:

- **From Frontend:**
  - **Communication type:** REST API
  - **Description:** Receives search requests from companies, including filters like location, skills, education, and work experience.

#### Outgoing Communication:

- **To Applicant Subsystem:**
  - **Communication Type:** API Call
  - **Description:** Queries applicant data from the Applicant Subsystem using the provided API to retrieve relevant profiles matching the company's search criteria.
- **To Frontend (via API Gateway):**
  - **Communication Type:** REST API
  - **Description:** Returns the search results — including applicant names, skills, and locations — back to the frontend for display, allowing companies to view and interact with the matched applicant list.

### 7.3.6 [Authorization Microservice](#)

#### Incoming Communications:

##### From Authentication Service:

- **Communication Type:** Kafka Event
- **Description:** Receives authentication and token verification events to validate user identity and role information before authorizing access to protected endpoints.

##### From Payment Service:

- **Communication Type:** Kafka Event
- **Description:** Receives payment validation requests to confirm that the company has the required permissions or subscription level to complete a transaction.

##### From Company Profile Service:

- **Communication Type:** Kafka Event

- **Description:** Validates whether a company has permission to update or access specific profile information.

**From Job Post Service:**

- **Communication Type:** Kafka Event
- **Description:** Verifies that a company user has the required role to create, edit, publish, or delete job posts.

**From Applicant Search Service:**

- **Communication Type:** Kafka Event
- **Description:** Validates that a company has sufficient privileges or premium subscription to perform applicant search operations.

**From Matching Service:**

- **Communication Type:** Kafka Event
- **Description:** Checks authorization to use real-time applicant matching features, typically reserved for premium companies.

**From Subscription Service:**

- **Communication Type:** Kafka Event
- **Description:** Receives subscription validation requests to ensure that premium access rights are correctly linked to company roles.

**From Notification Service:**

- **Communication Type:** Kafka Event
- **Description:** Verifies that a company or user is authorized to receive or trigger specific notification events, such as match updates or subscription reminders.

**Outgoing Communications (none):**

The Authorization Microservice is a centralized validation component and does not initiate outbound communications. It responds to requests and events from other services with authorization decisions but does not send independent messages or trigger external actions.

### **7.3.7 Notification Microservice**

**Incoming Communications:**

- **From Service Discovery:**
  - **Communication Type:** REST API / Kafka
  - **Description:** The service interacts with Service Discovery to obtain the required service locations dynamically and ensures that notifications are routed properly for consistent communication.

**Outgoing Communications:**

- **To Email API Provider:**
  - **Communication Type:** REST API
  - **Description:** The Notification Service sends email notifications to applicants or companies via an external Email API Provider, notifying them about relevant events such as job post updates or successful application submissions.
- **To Frontend:**
  - **Communication Type:** REST API
  - **Description:** Sends updates about notifications to the frontend to display real-time alerts or information to users (applicants or companies) regarding job posts and application statuses.

### 7.3.8 [Job Post Service](#)

#### Incoming Communications:

- **From Frontend:**
  - **Communication Type:** REST API
  - **Description:** Receives job-related requests such as creating, updating, or deleting job posts, along with related metadata (e.g., title, description, salary, and skills).
- **From Profile Service:**
  - **Communication Type:** Kafka Event
  - **Description:** Receives queries from the Profile Service to check if the company has a valid profile and is authorized to post job ads.

#### Outgoing Communications:

- **To Authorization Service:**
  - **Communication Type:** Kafka Event
  - **Description:** Sends authorization events to the Authorization Service to verify if the company posting the job has sufficient permissions for the action being requested (e.g., creating or editing job posts).
- **To Profile Service:**
  - **Communication Type:** REST API
  - **Description:** Communicates with the Profile Service to verify the status of the company's profile and ensure they meet the necessary requirements to post jobs.

### 7.3.9 [Profile Service:](#)

#### Incoming Communication:

- The API Gateway routes HTTP requests to the Job Post microservice after validating user requests through token verification (via the Filter component).
- Services Discovery dynamically provides the microservice location as part of the routing process.

#### Outgoing Communication:

- The Authorization Service is queried to check for proper permissions before allowing access to job post creation and management functions.
- The microservice also interacts with the Kafka Service to publish job post-related events for consumption by other services like Applicant Search or Matching Service.

## 8. Frontend Component Diagrams

The DEVision Frontend adopts a strictly modular Componentized Architecture designed to ensure scalability and maintainability. To enforce the Separation of Concern principle, every functional feature such as Search, Payment, Job Management are implemented using a standardized “UI-Hook-Service” layered pattern: Presentation Layer (UI), Logic Layer (Hooks), Data Layer (Services). This structure is supported by centralized infrastructure components, including HTTP Util for secure networking and State Management for global data synchronization, ensuring a cohesive and robust client-side application.

### 8.1. Auth components – [Login Component](#) & [Registration Component](#)

- **Purpose of Login Component Subsystem:**

The Login module is responsible for verifying user identities and establishing secure sessions via JWE (JSON Web Encryption) tokens.

- **Login UI & Social Login UI:** These components render the entry interfaces. The Login Form handles standard email/password inputs using elements from the Reusable library, while the Social Login Form integrates external identity providers (Like Google). Both utilize the Assets module for branding and Icons.
- **Login Hook:** Acting as the logic controller, this hook manages the complex state of the authentication process. It handles form validation, manages “Loading” indicators during network requests, and processes error states (Invalid Credentials).
- **Login Service:** This layer encapsulates the API interaction. It prepares the credentials and executes the POST request to the backend authentication microservice.

- **Login Component Interaction & Data Flow:**

- **Credential Submission:** When a user submits the form, the Login Hook captures the input.
- **Authentication Request:** The hook delegates the request to the Login Service, which retrieves the specific endpoint from URL Config and uses HTTP Util to send the data.
- **Session Establishment:** Upon a successful response (200 OK), the backend returns a JWE token.
- **State Synchronization:** The Login Service immediately dispatches this token and the user’s profile data to the State Management module. This global state update instantly authenticates the user across the entire application, redirecting them from the Login page to their Dashboard.

- **Purpose of Registration Component Subsystem:**

The registration module manages the creation of new accounts, ensuring that all necessary user data is collected and validated before interacting with the backend.

- **Social Register & Register UI:** These presentational components handle the data entry. They are responsible for collecting user details (Name, Role, Email) and offer OAuth alternatives via the Social Register component.
- **Register Hook:** This hook implements critical business rules on the client side. It performs real-time validation to provide immediate feedback to the user before the form is submitted.
- **Register Service:** This layer handles the transmission of new user data. It formats the payload to match the backend’s expected schema (DTOs) for user creation.

- **Registration Component Interaction & Data Flow:**

- **User Onboarding:** The process begins via the Page entry point. The user interacts with the registration forms, backed by Reusable UI components.
- **Validation & Submission:** The Register Hook ensures data integrity. Once validated, it triggers the Register Service.

- **Account Creation:** The Service uses HTTP Util to send a POST request to the backend creation endpoint defined in URL Config.
- **Post-Registration Logic:** Upon success, the system typically triggers an automatic login flow or redirects the user to the Login screen, depending on the configured UX flow, while updating State Management to reflect the new user status.
- **Justification for Authentication Components Design (Login & Register)**
  - **Token-Based Authentication (JWE):** The use of JWE allows for Stateless Authentication. The server does not need to store session data, making the backend easier to scale horizontally. The Frontend Security container manages these tokens to prevent XSS attacks.
  - **Interceptors for Authorization:** The HTTP Util is designed to automatically attach the JWE token to outgoing requests. This Aspect-Oriented Programming approach removes the burden of manual authentication from individual developers, ensuring that no API call is accidentally left unsecured.

## 8.2. [Profile components](#)

- **Purpose of Component Sub-Systems:**  
The Profile Management module allows users (both Admins and Companies) to view and modify their account details. This module adheres to the same “UI-Hook-Service” layered architecture used throughout the DEVision application, ensuring that sensitive user data handling is secure, modular, and decoupled from the visual interface.
  - **Profile Detail Component (Read Operations):**  
This subsystem handles the retrieval and displays of the user’s current account information
    - UI Layer: Renders a read-only view of the user’s profile, displaying fields such as Name, Email, Role, and Company affiliation using standard styles from the Reusable library.
    - Hook Layer: Manages the lifecycle of data fetching. It triggers the retrieval process when the component mounts and handles “Loading” states to provide visual feedback to the user.
    - Service Layer: Encapsulates the transactional logic, sending PUT or PATCH requests to the backend to persist changes or DELETE requests for account deactivation.
  - **Profile Update Component (Write Operations):**  
This subsystem manages the editing of text-based profile information.
    - UI Layer: Provides an interactive form interface where users can modify their personal details. It utilizes input components from the Reusable system.
    - Hook Layer: Acts as the form controller. It is responsible for validating user input (ensuring email format is correct), capturing form data, and managing the submission process.
    - Service Layer: Encapsulates the transactional logic, sending PUT or PATCH requests to the backend to persist changes or DELETE requests for account deactivation.
  - **Profile Avatar Component (Media Management):**  
This specialized subsystem focuses specifically on the user’s profile picture.
    - UI Layer: Displays the current avatar image and provides controls for uploading a new one (file picker or drag-and-drop zone), utilizing icons from the Assets module.
    - Hook Layer: Handles file-specific logic, such as generating local image previews before uploading and managing binary file data.

- Service Layer: Manages the multipart/form-data network requests required to upload image files to the server.

- **Component Interaction and Data Flow**

The interaction flow ensures that profile data remains consistent across the application.

- **Initialization:** When the Page (Blue Box) loads, the Profile Detail Hook immediately triggers the Profile Detail Service. The service retrieves the endpoint from URL Config, calls the backend via HTTP Util, and populates the view with the user's data.
- **User Modification:** When a user edits their information in the Profile Update UI, the Update Hook validates the data. Upon submission, it calls the Update Service to save changes to the database.
- **Global State Synchronization:**  
A critical interaction shown in the diagram is the connection to State Management (Blue Box, left). When the Profile Update Component successfully saves new data (changing user's name), it dispatched an update to the global state. Because the Profile Detail Component and other parts of the app subscribe to this state, they automatically re-render to show the new name immediately. This eliminates the need for a page refresh and ensures the user sees their changes reflected instantly across the entire system.

- **Justification for Profile Management Design**

- **Specialized Handling for Media:** The architecture separates Profile Avatar logic from standard text updates. This is justified by the technical differences in data handling such as images require multipart/form-data and binary processing, whereas text uses standard JSON. Separating them prevents the lightweight text update logic from being bogged down by heavy file processing logic.
- **Reactive User Experience:** The tight integration with Global State Management is critical here. When a user updates their profile, the change must be reflected immediately in the application header/navbar. This UI approach improves performance and user satisfaction.

### 8.3. [Job Post components](#)

- **Purpose of Component Sub-Systems:**

The module is segmented into three logical functional groups:

- **Job Post Table Component:**  
This subsystem is responsible for displaying the inventory of job postings created by the company
  - UI Layer: Renders the job list in a tabular format, displaying key metrics such as Job Title, Status (Active/Closed), Posted Date, and Applicant Counts.
  - Hook Layer: Manages client-side logic for data organization, including pagination state (current page index) and row selection for bulk actions (deleting multiple jobs).
  - Service Layer: Encapsulates the logic for fetching paginated job lists
- **Job Post Form Component:**
  - UI Layer: Provides the input interface, including text fields for Job Titles, rich text editors for Descriptions, and dropdowns for Salary Ranges and Skill Tags.
  - Hook Layer: Acts as the form controller. It handles form validation (ensuring required fields are filled), manages the "Loading" state during submission, and captures user input.
  - Service Layer: Responsible for executing transactional API calls. It handles POST requests to create new jobs and PUT/PATCH requests to update existing ones.

- **Job Post Search and Filter Component:**  
This subsystem allows users to locate specific job posts within large datasets.
  - UI Layer: Displays filter controls such as “Filter by status” (Open/Closed), “Date Range”, or keyword search bars.
  - Hook Layer: Implements optimization logic like debouncing to prevent excessive API calls while the user types and manages the active filter state.
  - Service Layer: Serializes the filter state into URL query parameters to request filtered datasets from the backend.
- **Component Interaction and Data Flow**  
The components operate through a synchronized flow that ensures data integrity between the user’s view and the server state:
  - **User Interaction:** The workflow begins when a user interacts with the Page (Blue box), such as clicking “Create Job” or typing in the search bar. The UI Components (Pink boxes, left) render these elements using consistent styles from the Reusable library.
  - **State & Logic Processing:** The Hooks (Pink boxes, center) intercept these interactions. For a “Create Job” action, the Job Post Form Hook validates the input data. For A “Search” action, the Search Hook updates the local filter state.
  - **API Execution:** The Hooks delegate network operations to the Services (Pink boxes, right). The Services retrieve the correct API endpoints the payload to the HTTP Util.
  - **Global Synchronization:** The State Management module (Blue box, bottom) plays a critical role here. When a Job is successfully created via the Form Component, the application state is updated. Because the Table Component subscribes to this state, it automatically refreshes the list to include the newly created job without requiring a manual page reload. This ensures immediate feedback and reactive user experience.
- **Justification for Job Post Management Design**
  - **Reusability of UI Elements:** The Job Post module heavily relies on the Reusable Components library for form inputs and table structures. This ensures Interface Consistency (UI/UX) across the application and accelerates development speed.
  - **URL Configuration Extraction:** By extracting API endpoints into a dedicated URL Config module, the system achieves Environment Independence. The application can easily switch between Development, Staging, and Production API environments without modifying component code, improving the DevOps pipeline.

#### 8.4. [Applicant Search components](#)

- **Purpose of Component Sub-Systems**  
The module is divided into three functional groups, indicated by the dashed boundaries in the diagram:
  - **Applicant Table Component:**  
The primary purpose of this group is to handle the macroscopic view of the candidate database.
    - UI Layer: Renders the data in a structured, tabular format, providing visual controls for sorting columns.
    - Hook Layer: Manages the complexity of pagination logic and row selection state.
    - Service Layer: Responsible for constructing the specific API requests required to fetch “slice” of data.
  - **Applicant Card Component:**  
This group manages detailed interaction with individual applicant records.

- UI Layer: Displays rich media content, such as profile pictures, skill tags, and status indicators.
- Hook Layer: Handles user-initiated transactions, such as “Shortlisting” a candidate or “Viewing Profile details”.
- Service Layer: Executes the transactional API calls to persist these status changes to the backend database.
- **Applicant Search and Filter Component:**  
This group acts as the control mechanism for the data view.
  - UI Layer: Provides the interface for inputting keywords, location data, and educational criteria.
  - Hook Layer: Implements performance optimization logic, such as debouncing, which delays the API call until the user stops typing to prevent network congestion.
  - Service Layer: Translates user inputs into standardized query parameters to be sent to the search endpoint.
- **Component Interaction and Data Flow**  
The components interact through a defined unidirectional data flow that connects the user interface to the backend infrastructure:
  - **Event Triggering:** the flow initiates in the UI Components (Pink boxes, left) when a user interacts with the system, for example typing a skill in the Search UI or Clicking “Next Page” in the Table UI. These components rely on the Assets and Reusable external modules to ensure the interface matches the DEVision design system.
  - **Logic Processing:** the interaction is captured by the corresponding Custom Hook (Pink boxes, center). The hook updates the local React state and processes the input data.
  - **Data Request:** the hook invokes a function in the Service Layer (Pink boxes, right). The Service layer retrieves the precise API endpoint string from the centralized URL config module. This ensures that endpoint management is decoupled from the component logic.
  - **Network Execution:** The Service delegates the actual network execution to the HTTP Util. This utility intercepts the request to attach necessary security headers (such as JWE Tokens) and manages communication with the backend Microservices.
  - **State Synchronization:** crucially, all three sub-systems connect to the global State management module (Blue box, bottom). When the Search Component updates the filter criteria, it modifies the global state. The Table Component, which is subscribed to this state, automatically detects the change and triggers a re-fetch of the data. This ensures that the search bar and the results table remain perfectly synchronized without requiring direct coupling between the two components.
- **Justification for Applicant Search Component Design:**
  - **UI-Hook-Service Pattern:** We implemented a layered architecture within components to enforce the Single Responsibility Principle:
    - UI: Purely presentational to ensure easy redesigns.
    - Hooks: Contain the complex business logic and state, allowing for easier unit testing of logic without DOM rendering.
    - Services: Isolate external API dependencies. If the backend API changes, only the Service layer needs updates, leaving the UI untouched.
  - **Debouncing Strategy:** The Search Hook implements debouncing logic to optimize Network Performance. By delaying API calls until user input stabilizes, we significantly reduce the load on the backend server and prevent race conditions where old search results overwrite new ones.

## 8.5. Payment components

- **Purpose of Component Sub-Systems:**

The Payment Module handles the monetization and subscription lifecycle within the DEVision platform. It is designed to ensure secure processing of financial transactions by strictly separating the user interface from the transaction logic. This modules interfaces with external payment gateways (via the backend) and manages the user's subscription status.

- **Payment Plan Component (Selection & Display):**

- UI Layer: Renders pricing cards that display plan detail, pricing, and feature lists.
    - Hook Layer: Manages the state of the currently selected plan and handles the initial data loading to ensure up-to-date pricing is displayed.
    - Service Layer: Executes GET requests to fetch the active pricing configuration from the backend, ensuring the frontend never displays hard-coded or outdated prices.

- **Payment Form Component (Transaction Processing):**

- UI Layer: Provides the input fields for payment information. Crucially, this often integrates secure elements from payment providers to ensure PCI compliance.
    - Hook Layer: Acts as the secure controller. It validates user input and manages the tokenization process which exchanging raw card data for a secure single-use token before any data touches the DEVision backend.
    - Service Layer: Transmits the secure payment token and the selected plan ID to the backend API to finalize the subscription.

- **Payment History Component (Record Keeping):**

- UI Layer: Displays a tabular view of past payments, showing dates, amounts, and status (Success/Failed).
    - Hook Layer: Manages the loading state and pagination of the transaction log.
    - Service Layer: Fetches the user's billing history from the backend database.

- **Component Interaction and Data Flow**

The interactions flow focuses on security and immediate state synchronization

- **Plan Selection:** When the component mounts, the Payment Plan Hook triggers the Service to load available tiers via HTTP Util. The user selects a plan, which passes the PlanID to the Payment Form Component.
  - **Secure Transaction Execution:** When the user submits the form, the Payment Form Hook validates the inputs. It then uses the Payment Form Service to send the secure token to the backend. The HTTP Util attaches the user's JWE (Authentication Token) to ensure the request is authorized.
  - **State Synchronization:** A critical interaction involves State Management (Blue box, bottom center). Upon receiving a "Success" response from the Payment Form Service, the hook dispatches an action to the global state. This instantly updates the auth state stored in the application. Consequently, other parts of the application immediately unlock premium features without requiring the user to log out or refresh the page.
  - **History Refresh:** Simultaneously, the successful transaction triggers the Payment History Hook to re-fetch the transaction list, ensuring the new payment appears in the history log immediately.

- **Justification for Payment Component Design**

- **Security and PCI Compliance:** The Payment Form Hook handles the tokenization of credit card data directly with the provider such as Stripe before it reaches our services. This design decision is critical for Security Compliance, ensuring that

sensitive raw edit card data is never stored or processed by the DEVision backend, reducing the attack surface.

- **Decoupled Subscription Logic:** The separation of the Payment Component from the Subscription Manager allows for flexible business logic. We can change the definition of “Premium Features” in the Subscription Manager without risking breaking the critical payment processing code.

## 9. Architecture Rationale

### 9.1. Data model

This section explains how the design of our ERD affects maintainability, extensibility, resilience, scalability, security, and performance of our system, including both advantages and disadvantages in the DEVision application context.

#### 9.1.1. Maintainability

##### Advantages.

- The data model is split into small, focused aggregates (for example, Company / PublicProfile / CompanyMedia for profile, JobPost / JobPostSkill / SkillTag for jobs, Subscription / PaymentTransaction for billing). Each backend service owns its own tables, APIs, and business rules. Therefore, this makes it easier for future developers to understand and test one feature at a time. A practical scenario could be creating a new “JobPost salary rule”; this only touches the Job Post service and its entities, not the whole database. This will ensure the integrity of our data and prevent data leakage or interruption.
- On the Frontend, the UI is structured by feature (Company Profile pages, Job Management pages, Subscription pages) and consumes typed DTOs. This clear mapping between the system components improves readability of the code while keeping changes local.

##### Disadvantages.

- Some end-to-end flows are processed by multiple services, which makes them harder to debug. For example, “company with a premium subscription creates a headhunting profile and then receives notifications” goes through different services, including Auth, Subscription, SearchProfile, Notification, and external Kafka events. Understanding or testing this flow requires in-depth technical skills and thorough integration between services instead of one codebase.
- Our team must keep DTOs, ERD, and API docs in sync across repos; if someone forgets, it can cause subtle bugs (a field renamed in the JobPost entity but not in the frontend DTO).

#### 9.1.2. Extensibility

##### Advantages.

- Because the models are separated and cohesive, we can extend the behaviour of our application by adding new services or new entities without breaking existing ones. For instance, adding a “CompanyReview” or “RecruiterUser” feature would mainly require a new aggregate and a few new endpoints, leaving Company, JobPost, and Subscription unchanged. Thus, it requires minimal changes and configurations.
- Shared concepts like SkillTag and soft IDs (companyId, jobPostId, applicantId) make it easy to plug new features into the existing matching logic, such as adding recommended jobs for a company based on recruitment history, by reusing the same IDs and enums.

**Disadvantages.**

- Extending features that require cross-system communication (like adding another premium tier with different matching behaviour) might require coordinated changes in multiple services (Subscription, SearchProfile, Notification, UI), which adds additional effort and raises the cost of bigger features.
- Because Job Manager and Job Applicant share the SkillTag catalog logically, evolving skills such as merging or renaming skills must be coordinated between squads and may require versioning or migration scripts.

**9.1.3. Resilience****Advantages.**

- The microservice + Kafka event-driven backend design naturally isolates failures if one service is shut down. If the Payment service or Notification service is temporarily down, companies can still log in, edit profiles, and manage job posts; only payments or alerts are degraded.
- Using soft references instead of hard foreign keys for cross-system data (for example, JA stores jobPostId and companyId, and JM stores applicantId in ApplicantFlag) means that a failure on the other subsystem’s database does not directly corrupt and interfere with our own. We only need to handle missing or stale IDs defensively at the API layer.

**Disadvantages.**

- More network hops and services mean more potential chances where things can fail or not work properly. A simple action like “show all applications for this job” can depend on Job Manager, Applicant, and Notification services all being reachable.
- We must implement retry logic, timeouts, and circuit breakers in the backend and show fallback states in the frontend, which increases implementation complexity compared to a single in-process system.

**9.1.4. Scalability****Advantages.**

- The data model design supports the growth of our system: write-heavy entities like Company, JobPost, and SearchProfile are designed to work with sharding or partitioning by country / companyId and are indexed on common filters (status, employmentTypes, salary range, location). This supports SRS requirements around searching for jobs and applicants at scale.
- On the backend, each service can be scaled independently. For example, when there is heavy load on job search, we can scale the Job Post and Applicant Search services (and their read replicas) without scaling Payment or Notification.
- The frontend makes only the necessary API calls, which helps control load on backend services.

**Disadvantages.**

- Sharding and partitioning add operational overhead to our system. If a company changes its location, we may need to migrate its data to a different shard, and global analytics (for example, all jobs by salary range across all countries) becomes more complex to implement.
- Scaling many small services requires more DevOps work (deployment, monitoring, load balancing for each service) compared with a single monolithic server.

**9.1.5. Security****Advantages.**

- Sensitive data is isolated in the Authentication & Authorization services: AuthAccount and AuthToken store passwords, SSO identifiers and token metadata. Business entities like Company, PublicProfile, and JobPost never contain password hashes or raw tokens. The frontend and other services only receive safe DTOs and opaque token strings.
- Service boundaries make it easier to apply stricter access control around high-risk components. For example, only the Auth service can access credentials; only the Payment service can access gateway transaction references; all other services can only see the filtered views.

**Disadvantages.**

- Because security checks are enforced at multiple layers (API Gateway, Auth, individual services), misconfiguration can happen during the development process; we might forget to apply an authorization check to a new endpoint, for example. This requires careful testing and documentation.
- Token-based flows and system-to-system authorization, such as internal APIs between JM and JA, will add cognitive load for developers compared to a simple cookie-based monolith. This requires more time spent on the development journey and requires careful investigation.

### 9.1.6. Performance

#### Advantages.

- Normalised entities and clear boundaries avoid large, bloated tables. For example, separating CompanyMedia and PublicProfile from Company means we don't load media metadata when we only need basic company info for job lists, which keeps queries lighter and faster to process.
- Indexing on job search filters (location, status, salary range, skill requirements) and using a shared SkillTag catalog allow the Job Post and Applicant Search services to implement efficient queries and caching for common scenarios like “find all jobs in Viet Nam requiring React”.
- On the frontend, feature-based pages call only the APIs they need and can cache static data like SkillTag lists, reducing repeated network calls.

#### Disadvantages.

- Microservices introduce network latency and serialization overhead. A complex screen like “Company dashboard with jobs, premium status, flags and notifications” may require multiple API calls and therefore be slower than a monolithic view if not carefully optimised by our team.

Normalisation sometimes requires extra joins or service calls; for instance, building a rich job detail page may require data from JobPost, JobPostSkill, SkillTag, and PublicProfile, which must be composed efficiently to avoid slow responses

## 9.2 System Architecture

- **Maintainability**

#### Advantages:

- Each microservice has a clear responsibility, making it easier to understand, update, and test independently.
- Issues in one service can be fixed without affecting others, reducing downtime and deployment complexity.

#### Disadvantages:

- Increased number of components can make the system harder to trace and debug if services interact incorrectly.
- Requires disciplined documentation and versioning to maintain clarity.

- **Extensibility**

#### Advantages:

- Modular services allow new features or enhancements to be added with minimal impact on existing services.
- Easy integration of additional third-party providers or new functionality without touching core logic.

#### Disadvantages:

- Over-extending services without proper design can lead to fragmented responsibilities and coupling.

- Each extension may require additional communication mechanisms, increasing system complexity.
- **Resilience**
  - Advantages:**
    - Asynchronous Kafka communication allows services to continue functioning even if one service temporarily fails.
    - Prevents cascading failures and ensures continuous operation during unexpected conditions.
  - Disadvantages:**
    - Failure recovery can be complex if events are not processed correctly or if message queues backlog.
    - Debugging failures in an asynchronous system is more difficult than in a monolithic synchronous flow.
- **Scalability**
  - Advantages:**
    - Services can be scaled independently based on load (e.g., Authentication or Applicant Search under heavy demand).
    - Containerization (Docker) allows horizontal scaling across hosts and environments.
  - Disadvantages:**
    - Requires proper orchestration and monitoring to avoid over/under-provisioning.
    - Scaling multiple services may increase infrastructure costs.
- **Performance**
  - Advantages:**
    - REST APIs provide immediate response to frontend requests.
    - Kafka enables asynchronous background processing for notifications, updates, and validation, reducing blocking.
  - Disadvantages:**
    - Inter-service communication adds network latency compared to in-process calls.
    - Performance depends on proper tuning of Kafka, databases, and caching.
- **Security**
  - Advantages:**
    - Multi-layered checks (API Gateway, Filters/Security Config, Authorization Service) reduce risk of unauthorized access.
    - Tokens and sensitive data are validated efficiently, providing robust protection.
  - Disadvantages:**
    - Multiple layers can introduce complexity and potential misconfigurations.
    - Ensuring consistency in role/permission rules across services requires careful management.
- **Performance**
  - Advantages:**
    - REST APIs provide immediate feedback to users for frontend interactions, ensuring responsive UI.
    - Kafka enables asynchronous processing of background tasks (e.g., notifications, profile updates, subscription validation), preventing blocking of critical operations.

- Use of caching (e.g., Redis) in some services reduces database load and speeds up repeated data retrieval.

**Disadvantages:**

- Inter-service communication (Kafka or REST) introduces network latency compared to in-process calls in monolithic systems.
- Performance depends heavily on proper tuning of Kafka topics, consumer groups, databases, and caching layers.
- High volume of events or requests may require careful scaling and monitoring to avoid bottlenecks.

## 10. Conclusion

This architecture documentation has outlined how the DEVision Job Manager subsystem is structured to satisfy the functional and quality requirements defined in the SRS. The proposed data model organises entities around clear business capabilities (company onboarding, profile management, job posting, applicant search, subscription, payment, and notifications). This alignment with microservice boundaries enables independent development, deployment, and testing of each service while still supporting cross-team data exchange with the Job Applicant subsystem.

On the backend, a microservice-based architecture combined with API Gateway, Service Discovery, Kafka, and Redis provides a scalable and resilient foundation. Kafka decouples services through event-driven communication, Redis supports secure token revocation and caching, and country-based sharding prepares the platform for regional growth and reduced latency. Security is enforced through layered controls (Gateway, Filters/Security Config, Authentication and Authorization services), ensuring that sensitive data is isolated while still allowing other services to use safe DTOs and opaque tokens.

On the frontend, the UI–Hook–Service pattern and centralized state management deliver a modular and maintainable client application. Each feature (Auth, Profile, Job Post, Applicant Search, Payment) has clear presentation, logic, and data layers, which simplifies future enhancements and supports a consistent user experience. Tight integration with global state ensures that critical flows such as login, profile updates, and subscription upgrades are reflected instantly across the interface.

Overall, the chosen design involves trade-offs—such as increased operational complexity and more demanding debugging compared to a monolith—but these are justified by gains in scalability, resilience, extensibility, and security. This document therefore serves as a single reference point for future engineers and stakeholders, guiding implementation and evolution of the DEVision Job Manager subsystem as the platform and its requirements continue to grow.