Real-Time Streaming Platform

Software Design & Specifications Document

Avdhesh Kumar Singh
avdheshkumar.singh@gmail.com

Contents

Introduction

Purpose of the document

This document contains the requirements and design of a Real-Time Streaming Platform. This document also explains the tech-stack chosen to implement the components of the Platform along with the reason to choose.

This document covers functional and non-functional both kind of requirements in different sections.

The assumptions taken while designing the Platform and its scope has been covered. This gives clear picture what we can expect form this System and what all the areas which can be improved in further releases.

For designing and documenting the Architecture of the Platform, 4C model has been used. The 4C Design Methodology in software architecture is a systematic approach to designing, analyzing, and documenting software systems. It emphasizes breaking down the architecture into distinct levels of abstraction, each addressing different concerns and audiences. The "4C" stands for:

1. Context
2. Containers
3. Components
4. Code

Scope of the software

This design caters the functional and non-functional requirements which have been discussed in detail in Requirement Specifications section of this document.

There are certain important components which we can safely assume to be existing. The various alternatives of such components may have been discussed in this document.

Please see **Assumptions** section of this document for details.

System Overview

Key Objectives

Design a Real-Time Streaming Platform for an online shopping network (e.g., Amazon, Flipkart, Walmart Connect, or Target Roundel). This platform will process real-time data from retailer websites, capturing customer interactions (such as product views and add-to-cart actions), ad engagements (views and clicks), and other relevant events. The processed data will be utilized to generate insights via APIs, enabling marketers and retailers to optimize their advertising campaigns and enhance user engagement.

High-level description of the system

This Real-Time Streaming Platform will receive Customer's actions on retailer's portal as Events.
The Platform ingests this high-velocity Event stream and processes them in real-time.
The Platform stores Raw events and processed data into a persistence data storage.
This Data Storage supports real-time queries and batch analytics both.
The Platform have a set of RESTful APIs to provide insights processed in real-time.
The Platform is Scalable and Highly Available to cater traffic spikes during peak events traffic.
The RESTful APIs are secured, only register users can access them. There is RBAC implemented for APIs authorization.
The RESTful APIs have been deployed behind a API Gateway providing enhanced security, rate limiting etc.
The same Platform is a multi-tenant platform, caters multiple Retailer Portals to Event analytics.
The Platform is monitored for errors, it supports notifications in case of any issue. Metrics are getting collected to keep eye on the performance of the Platform.

Requirements Specification

Functional Requirements

(*From the original problem statement document shared*)

Event Ingestion:

- Identify technologies for handling high-velocity event streams from various sources while ensuring minimal latency.

Data Processing:

- Describe your approach for real-time event processing.
- Explain how you would manage stateful operations, such as tracking user sessions across multiple events.
- Specify which stream processing frameworks you would select and justify your choices.

Data Storage:

- Outline your design for the data storage layer to support both real-time querying and batch analytics.
- Recommend database technologies for storing raw events and aggregated insights (e.g., Cassandra, Snowflake, DynamoDB), and discuss data retention strategies.

Insights API Development:

- Design APIs for accessing insights:
  - GET /ad/{campaignID}/clicks → Returns the number of customers who clicked on the ad.
  - GET /ad/{campaignID}/impressions → Returns the number of customers who viewed the ads.
  - GET /ad/{campaignID}/clickToBasket → Returns the number of customers who added a product to their cart after clicking on the ad.
- Discuss how you would structure these APIs to support both real-time and historical queries while ensuring performance and reliability.

Multi-Tenancy:

- Describe how you would implement multi-tenancy in your design to support multiple retailers on a single platform while ensuring data isolation and security.
- Discuss strategies for managing tenant-specific configurations, data access controls, and resource allocation.

Non-functional Requirements

Metrics and Monitoring:

- Identify key metrics to monitor platform reliability and performance.
- Specify tools for logging, alerting, and diagnostics.

Following is the list of Software Quality Attributes this Platform must have-

- Availability
- Performance
- Security
- Testability
- Interoperability
- Modifiability
- Scalability

Assumptions

1. A campaign belongs to a Tenant only who is running it, a campaign is not valid across Tenants.
2. **Cloud**- AWS is the target cloud.
3. **Software Development Approach**- As much as possible the core tech-stack of the solution is cloud-native, but at infrastructure and deployment requirements it is allowed to use cloud managed services.

System Architecture

Architectural Patterns

Following Architectural patterns are suitable for Streaming Data Platform-

- Client & Server
- Microservices
- Publish & Subscribe

System Context Diagram

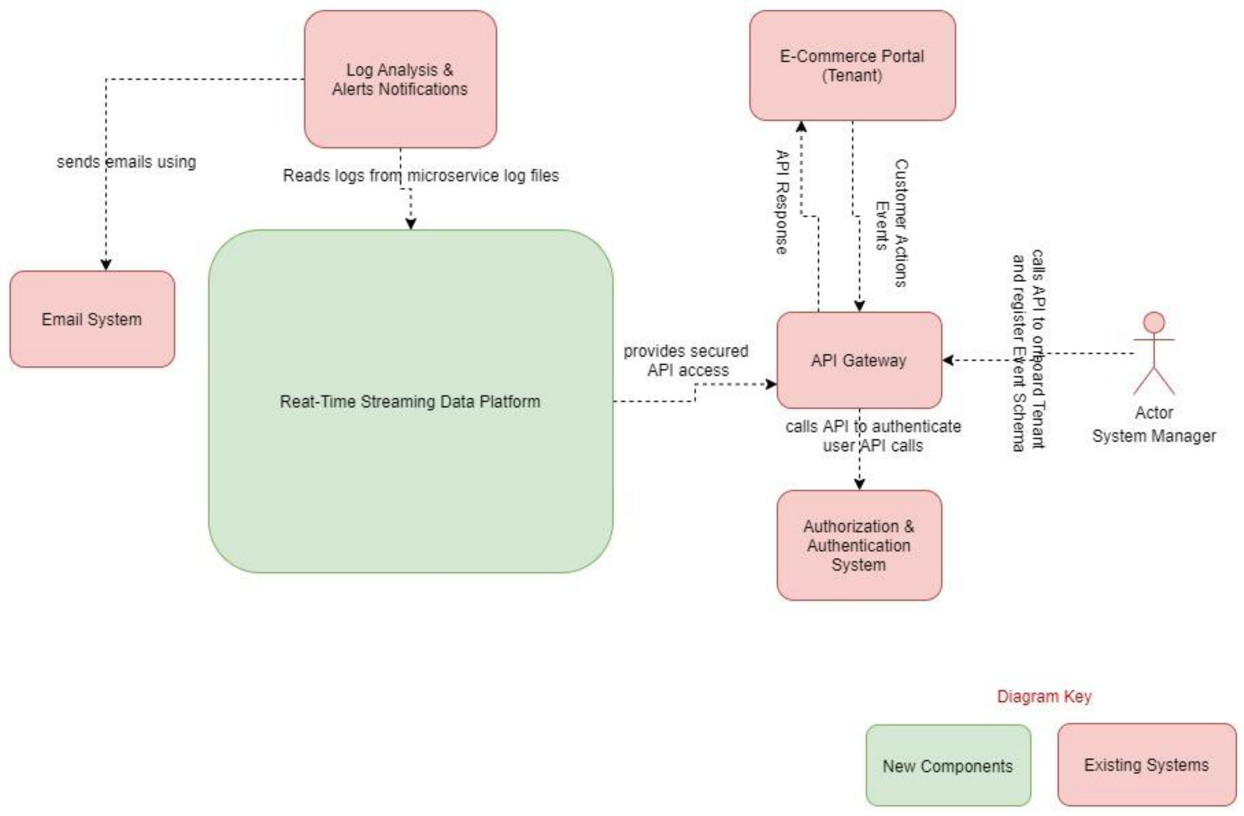

Log Analysis & Alerts Notifications
sends emails using
Reads logs from microservice log files
Email System
Reat-Time Streaming Data Platform
provides secured API access
E-Commerce Portal (Tenant)
API Response
Customer Actions Events
calls API to onboard Tenant and register Event Schema
API Gateway
Actor System Manager
calls API to authenticate user API calls
Authorization & Authentication System
Diagram Key
New Components
Existing Systems

Fig-1 System Context Diagram

Container Diagram

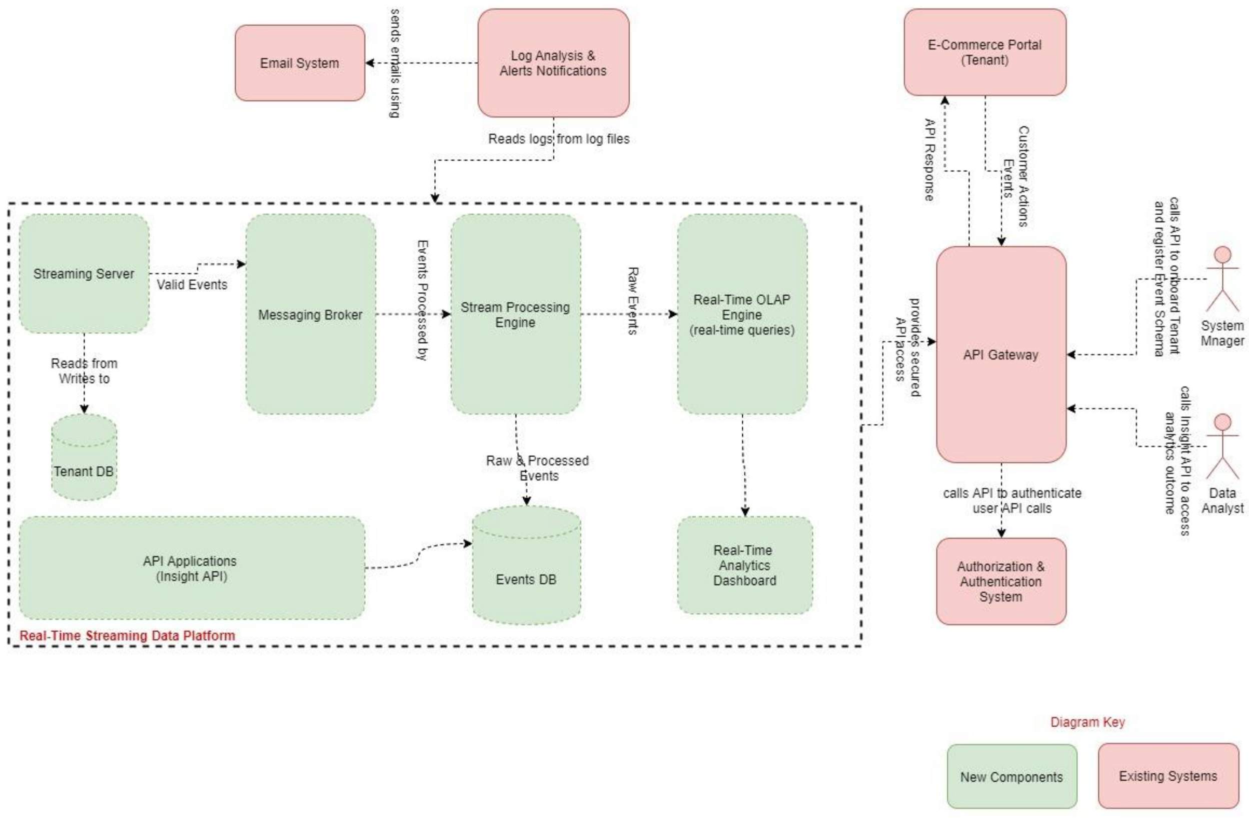

Email System
sends emails using
Log Analysis & Alerts Notifications
Reads logs from log files
E-Commerce Portal (Tenant)
API Response
Customer Actions Events
Streaming Server
Valid Events
Messaging Broker
Events Processed by
Stream Processing Engine
Raw Events
Real-Time OLAP Engine (real-time queries)
provides secured API access
API Gateway
calls API to onboard Tenant and register Event Schema
System Mnager
Reads from Writes to
Tenant DB
Raw & Processed Events
calls Insight API to access analytics outcome
Data Analyst
calls API to authenticate user API calls
API Applications (Insight API)
Events DB
Real-Time Analytics Dashboard
Authorization & Authentication System
Real-Time Streaming Data Platform
Diagram Key
New Components
Existing Systems

Fig-2 Container Diagram

Component Diagram - Streaming Server

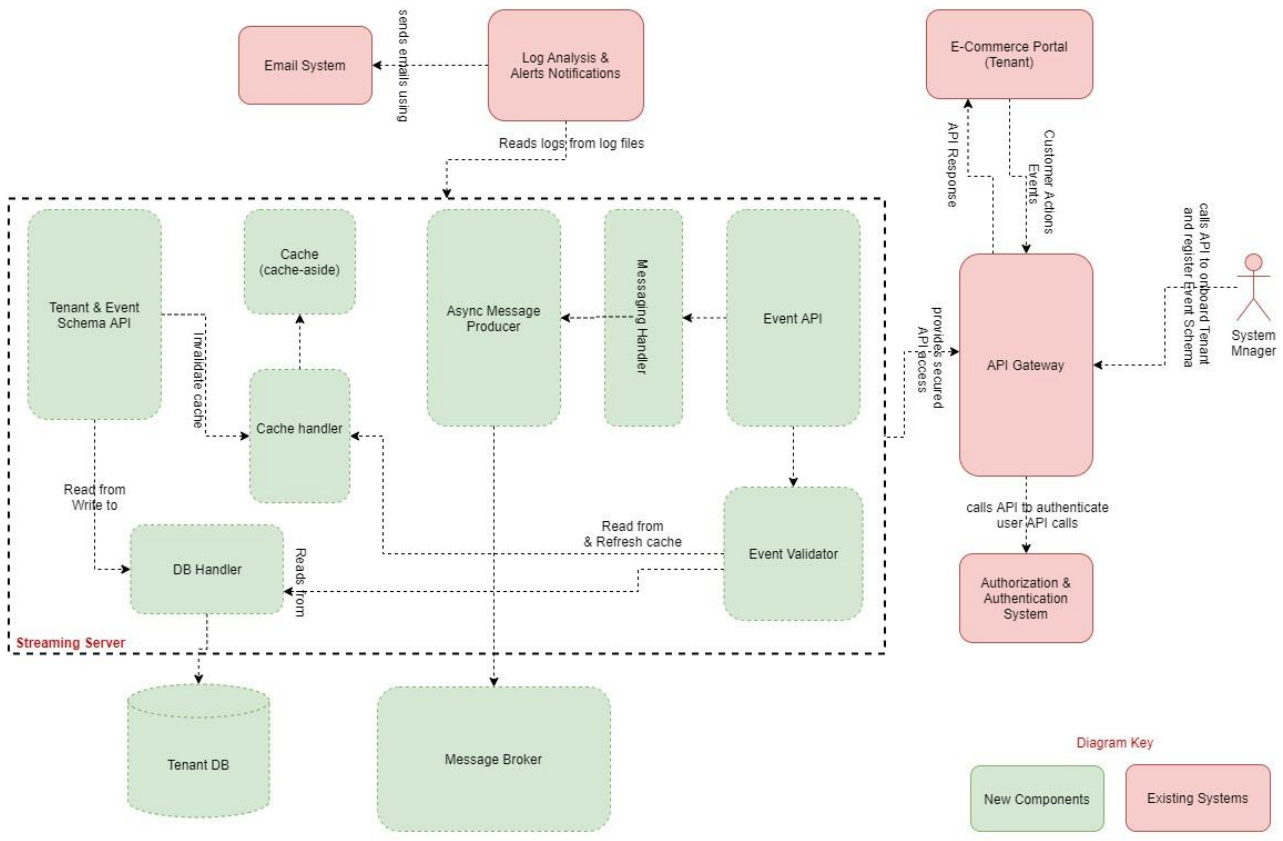


Fig-3 Component Diagram - Streaming Server

Detailed Design Description

Event Ingestion

In the Container Diagram (Fig-2) there is a component named **Streaming Server**. This service is supporting **multi-tenancy**, it exposes a RESTful API (POST, HTTPS/JSON) to receive the Event stream from registered Tenants only. This service is designed to handle very high velocity Event stream traffic.

It validates the incoming Event with the registered Schema persisted in Tenant DB. Every Event request will have Tenant ID and Event ID in header.

Once validated, Event will be sent to a Messaging cluster for processing in real-time. Every Tenant and Event type has a specific Topic to store the Events.

This is a Scalable and High Available service having multiple instances running behind a **Load Balancer**. This will be deployed in **Kubernetes** environment.

Tenant registration and schema of the Events which Tenant can send can be registered using **Tenant & Event Schema APIs**.

Streaming Server is using in-memory cache configured in cache-aside mode to keep Events Schema in the cache memory.

Please refer Fig-3 for Component Diagram of Streaming Server.

Data Processing

Stream Processing Engine

Once an Event lands to Messaging cluster in a Topic, **Stream Processing Engine** is responsible for process the Event in real-time.

This is a **Highly Available, Distributed** stream processing engine based on **Apache Flink**. Apache Flink cluster will be deployed in **Application mode** (using **Kubernetes Operator**) for better data and resource isolation guarantee as the resources are not shared across jobs (Tenants). This resource allocation model is the most preferred mode for production deployments.

Why Flink

Apache Flink is high performance, distributed, real-time Stream processing engine with its own cluster management UI with configuration and monitoring features.

Some of the main features it supports are-

- **Fault Tolerance**: Ensures that the application can **recover** from failures **without losing data** or computation progress.
- **Consistency**: Guarantees **exactly-once** processing semantics.
- **Scalability**: Efficiently manages large amounts of state using distributed storage and computation.
- Complex Applications: Enables building sophisticated streaming applications, such as machine learning pipelines, fraud detection systems, and **real-time analytics**.

Support of Stateful Operations

State management in Apache Flink is a critical concept that underpins its ability to process data in a fault-tolerant, consistent, and scalable manner. State refers to any information that a Flink application retains between events to make decisions or perform computations.

Apache Flink supports Stateful operations at **Keyed State** and **Operator State** level.

Data Storage

Events data is stored in **Apache Cassandra**, it stores data in a distributed and fault-tolerant manner designed to handle large amounts of structured and semi-structured data.

It supports tunable **Data Consistency** modes, **Eventual Consistency** by default but supports Strong Consistency as well.

It supports **Data Partitioning** and **Replication**. It uses **Bloom Filter** and **Index** files for performance. It is **Horizontally Scalable** and based on **Decentralized Architecture**.

It will store Raw Events for **Batch Analytics** and Processes Events for **Insight API** consumption.

Real-Time Query

Once Event is processed by **Stream Processing Engine**, it will store Raw Events in a **Real-Time OLAP Engine** for real-time queries.

Apache Pinot is an excellent candidate for this. Apache Pinot is an **open-source**, **distributed database** designed for **real-time analytics**, specifically focused on delivering **low-latency queries** at **very high throughput**, making it ideal for **user-facing applications** where fast response times are crucial. Apache Pinot is considered an **MPP** (Massively Parallel Processing) based system.

Real-Time Analytics Dashboard

Apache Superset is an excellent choice for Real-Time analytics dashboard. It is an open-source software application for data exploration and data visualization able to handle data at petabyte scale.

It supports integration with multiple OLAP Databases including Apache Pinot.

Batch Analytics

Apache Pinot is not meant for Batch Analytics, we have better tools in this space. We are saving Raw Events in Cassandra for Batch Analytics.

Apache Spark is an excellent choice here. With DataStax **Spark Cassandra Connector** it allows you to perform batch Analytics using Apache Spark accessing data stored in Cassandra database.

Data Retention Strategy

Retention of data is purely based on Analytics use cases implemented. But there are some generic guidelines, as per them-

1. **Cassandra** - Raw Events in Cassandra are used for Batch Analytics. Since its Click Stream data, the size of the data will increase quickly. On top of that application is a multi-tenant application. Processed Events data will be mostly aggregated data, very less in size compare to Raw Events.

So, mostly in Batch Analytics also, we don’t need Historic data since beginning, as per use cases we can keep certain months (e.g. 6 months) of data in the DB and rest we can export and move to cheap storage like S3.

Processed Data we can keep for longer duration.

2. **OLAP Database**- This is for real-time analytics and query only. Its rare to use historic data in such use cases. We can safely purge from OLAP DB. In case we need it, we can import from Cassandra or backed-up data on S3 storage.

Maximum 1 week data we can keep in OLAP database.

Insights API

Insight API should be a separate **Microservice** accessing the processed events in Cassandra DB. It should be secured with **Auth** implementation and deployed behind **API Gateway** and **WAF**.

The required endpoints in Insight API are as below-

- **GET /ad/{campaignID}/clicks**- Returns the number of customers who clicked on the ad.
- **GET /ad/{campaignID}/impressions**- Returns the number of customers who viewed the ads.
- **GET /ad/{campaignID}/clickToBasket**- Returns the number of customers who added a product to their cart after clicking on the ad.

API design for support of Real-Time and Historical queries

In the present form, above endpoints will fetch **real-time** processed data.

For Historic queries, **assuming historic window is by dates**, I would suggest to add 2 more query parameters in all the above endpoints as below-

- **GET /ad/{campaignID}/clicks/{fromDate}/{endDate}-** Returns the number of customers who clicked on the ad between given dates.
- **GET /ad/{campaignID}/impressions/{fromDate}/{endDate}**- Returns the number of customers who viewed the ads on the ad between given dates.
- **GET /ad/{campaignID}/clickToBasket/{fromDate}/{endDate}**- Returns the number of customers who added a product to their cart after clicking on the ad on the ad between given dates.

If only fromDate is present, endDate will be current Date.
If neither is present, it will get for current date only (Real-Time).

If historic window is by Time, we should replace **fromDate** and **endDate** by **fromTimestamp** and **endTimestamp** respectively.

In this case, we should consider **TimeZone** very cautiously. Better to store Data on Server on UTC timezone. In query parameter local timezone can be taken but before making query to Database, convert it to UTC timezone.

Data Partition Strategy

Data Partitioning depends upon how we need to access the Data.

Insight APIs are accessing the processed data and **assuming historic window is by dates** we should do Cassandra partitioning as below-

Partition Key- campaignID+eventType+date
Cluster Key- customerId

Here eventType representing customer actions e.g. clicks, impression, clickToBasket etc.

Multi-Tenancy

Whole Streaming Platform has been designed to support multi-tenancy as below-

Streaming Server- It receives Events from all the Tenant at one place. Incoming HTTP request has **tenantId** and **eventId** in the **header**.

Message Broker- Topics are created based on combination of **tenantId** and **eventId**.

Stream Processing Engine- Application mode deployment per Tenant, ensuring high security with complete data and resource isolation.

Databases- separate DB per Tenant. Since Stream Processing engine is deployed in Application mode, sink can be separate for Tenants.

Log Analysis and Alerts

ELK (Elastic, Logstash, Kibana) tech stack is suitable for Log Analysis and Alerts. Components of Streaming Platform should emit logs in file as a conventional way. Logstash needs to be configured to scrap the logs from these log files.

Kibana Alert system (Kibana Watch) can be configured to raise an Alert Notification(via email) in case some exception is found in the Logs.

Metrics and Monitoring

Prometheus and **Grafana** stack will be configured to collect metrices (CPU, memory, disk, API throughput, APIs requests per minute etc.) and display graphs/charts on Grafana dashboard.

Technology Stack

Streaming Server- **Java Spring Boot** microservice, running on Kubernetes cluster with HPA (Horizontal pod Autoscaler) support.

Messaging Cluster- Apache Kafka cluster with multiple brokers. Number of partitions of a Topic depends upon the traffic from Tenant, required capacity planning and calculations.

Stream Processing Engine- Apache Flink cluster.

Tenant & Event Databases- Cassandra cluster, highly available with backup and recovery options with Multi AZ deployments.

Tenant and Insight APIs- **Java Spring Boot** microservice, running on Kubernetes cluster with HPA (Horizontal pod Autoscaler) support.

API Gateway- Amazon API Gateway, configure all our RESTful APIs behind this with rate limiting and traffic management configured. API Gateway will be configured with Auth component to securing the APIs.

Auth System- **Amazon Cognito** is a good choice. Its managed service offering from Amazon based on OpenID Connect.
In case of **self-managed** open-source component, **KeyKloak** will be an excellent choice.

Email System- AWS SES managed service, easily configurable in Apache Airflow.

Container Orchestration- **Amazon EKS** is an excellent choice due to Managed Service. In case of cloud-native opne-source, **self managed Kubernetes Cluster** is another option. All microservices will be deployed in this environment for Scalability and High Availability.

Observability- **Prometheus** will be deployed along with microservices Pod to collect metrices (CPU, Memory, Disk, Requests per second. **Grafana** dashboard will be configured with prometheus to show the metrics as graphs/charts.

Log Analysis & Alerts- **ELK** (Elastic, Logstash, Kibana) stack to search logs and generate Notifications based on incidents/exceptions found in logs.

Web Application Firewall- Integrate **AWS WAF** to protect Report APIs from common web exploits like SQL injection, cross-site scripting (XSS), or request flooding. Define custom rules to block malicious traffic.

External Interfaces

Streaming Server, Insights API, Tenant API- RESTful services, HTTP/JSON, secured via HTTPS.

Email System- Native support for AWS SES for email routing, works on SMTP protocol.

Security Considerations

API Security

Amazon API Gateway configured with Amazon Cognito will provide security to the RESTful APIs.

API Gateway

Amazon API Gateway to be configured for Throttling, Rate Limiting and making endpoints working on HTTPS.

AWS WAF

To protect Report APIs from common web exploits like SQL injection, cross-site scripting (XSS), or request flooding. Define custom rules to block malicious traffic.

Deployment Strategy

Recommendations for **Deployment** strategy keeping **Disaster** and **Recovery** (DR) in view-

Streaming Server must be **Multi-site Active/Active** deployment.

Other RESTful APIs (microservices) may be **Multi-site Active/Passive** (**Warm Standby**) deployment.

For databases, **global replication** is required even for Active/Passive deployments.

For API (microservices) Blue-Green strategy for rolling updates.

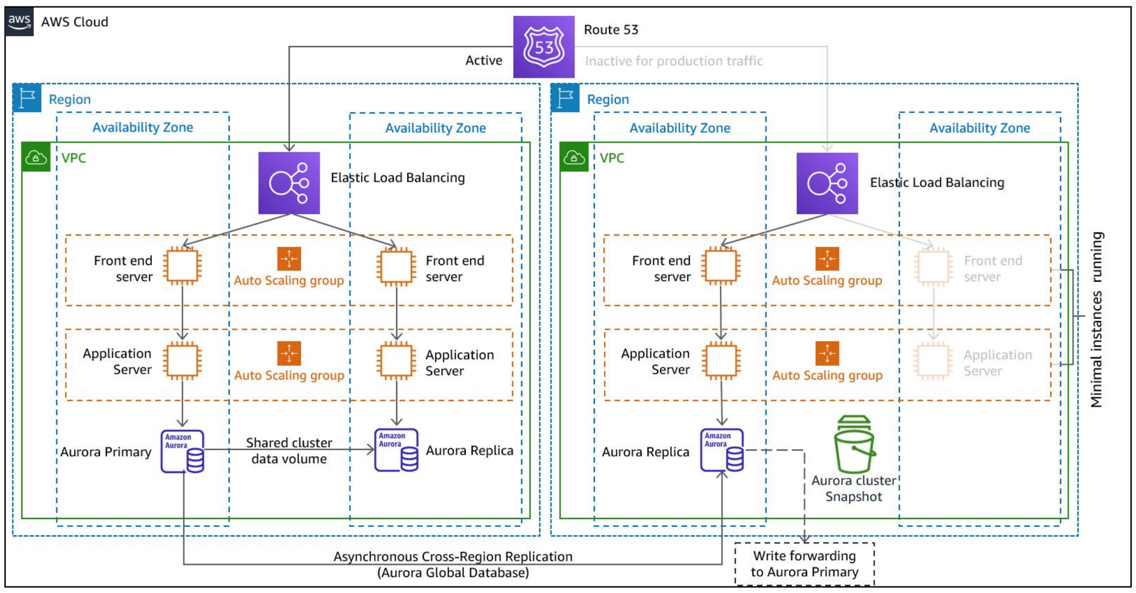
For messaging, Multi-site multi- Broker cluster is recommended

Deployment Diagram

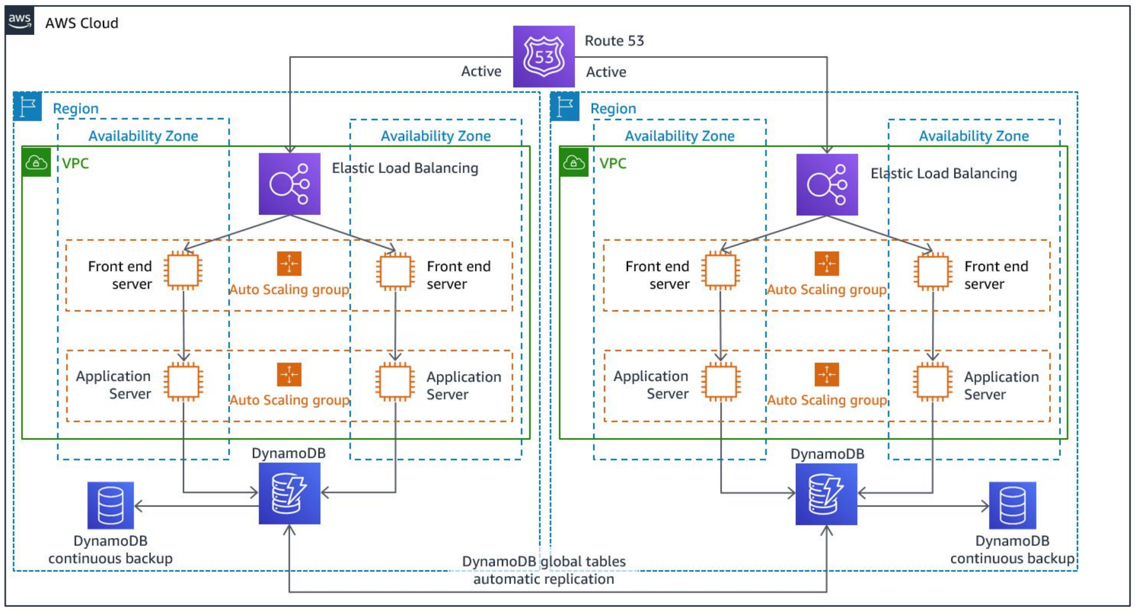
Following are the suggested Deployments strategy as per **AWS well Architected Framework** guidelines.

This is for reference only, please replace your services/database accordingly.

Multi-site Active/Passive (Warm Standby)


AWS Cloud
Route 53
Active
Inactive for production traffic
Region
Availability Zone
VPC
Elastic Load Balancing
Front end server
Auto Scaling group
Application Server
Aurora Primary
Shared cluster data volume
Aurora Replica
Aurora cluster Snapshot
Minimal instances running
Asynchronous Cross-Region Replication (Aurora Global Database)
Write forwarding to Aurora Primary

Multi-site Active/Active



Challenges & Trade-offs

Potential Challenges

Streaming Platform is a very complex, mission-critical Software System. It must be very **resilient** and robust, **Recovery** from failures and data consistency is must. In case of failures, data loss is unbearable.

These are complex aspects and hard to achieve. But with right set of Tools and Framework it can be done.

Since proposed Architecture is majorly cloud-native and mostly using self-managed services, it requires good amount of skill to implement.

Trade-off between real-time performance and data accuracy

Streaming Server is a Scalable and Highly Available application. Apache Flink is excellent Real-Time stream processing engine. It guarantees Exactly-Once delivery to ensure data accuracy and consistency.

Cost trade-off

The number of instances of a component should be decided based on anticipated load and it should be auto-scaling enabled. This way we can save money when load is less.

For most critical services only I have recommended Active/Active deployment whereas many components will follow Warm Standby deployment, hence saving money.

To lower the cost further, some non-critical APIs can be deployed in **Pilot Light** mode.

Code

Streaming Server is one of the most critical component of the Streaming Platform. Reference Implementation of this has been attempted as Coding part of the exercise.

Appendices