Table Of Contents

XENETA DATA INGESTION PIPELINE

System Overview

Above is shown the Rates Application comprising of -

1. Incoming Price Batch Files
2. Rates Application comprising Frontend and Postgres DB
3. Consumers who are accessing this rates information.

System Design

The system will be designed as below –

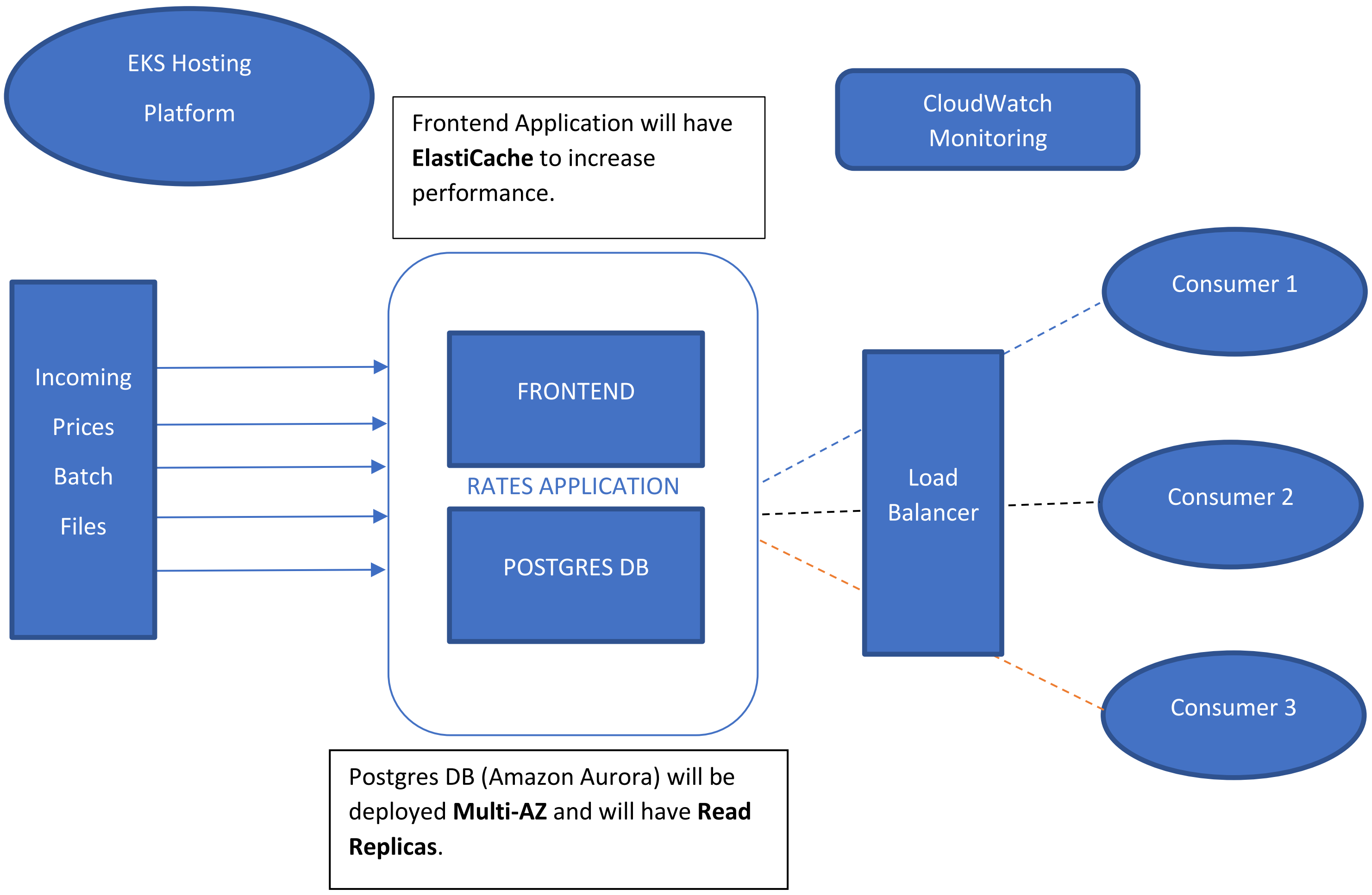
1. The application will be hosted on EKS using Load Balancer which will take care of **Auto-scaling** and distribution of load amongst the cluster nodes based upon sudden traffic changes.

2. The application will be deployed on a multi-master cluster so that this is **Highly Available** and survives if a master node goes down.

3. The Postgres DB will be deployed across Multi-AZ to make the application **Fault-tolerant** and will also have Read Replicas to enhance query performance for consumers.

4. The Frontend application will be using AWS **ElasticCache** and instead of hitting the application, response will be provided by cache for similar type of requests, thereby decreasing application load.

High Level Architecture Diagram



Frontend Application will have **ElastiCache** to increase performance.

Postgres DB (Amazon Aurora) will be deployed **Multi-AZ** and will have **Read Replicas**.

Application Monitoring

1. **CloudWatch** logs can be utilized to monitor our application (pods, nodes, services etc..) over EKS. We can have a Python script search for Pod keywords like - CrashLoopBackOff, ImagePullBackOff etc.

2. Alarms can be generated in the form of **notifications** like email or tickets to check the application health.

3. Kubernetes also provides **metrics** and **dashboards** showing the CPU, Memory Utilization.

Bottlenecks and Proposed Solution

1. With increasing load, the **Node size** may not be sufficient to start new pods and thus will require increasing the compute from t3.medium to m6a.4xlarge.

2. When consumers are trying to access application and nature of requests changes, we should be using **dynamic ElasticCache** so that refreshed data can be provided without delays.

3. When an Availability-Zone or Region in AWS goes down, the Postgres DB will not be accessible. Choosing **Multi-AZ** deployment and enabling **Global Replication** will help application to run on different regions in case of failure.

Additional Requirements Design Change

Which parts of the system are the bottlenecks or problems that might make it incompatible with the new requirements?

1. Processing the batch files would be relatively slower when there are huge incoming volumes.

2. Application might experience downtime when pushing in code updates when deployed on a single master cluster.

How would you restructure and scale the system to address those?

1. Modify **cluster node instance** type based on the deployment environment – lower size in Dev/Staging and higher compute in Production. For the application in Dev and Staging since the load would not be as high in Production, would choose a lower Node size – t3.medium.

2. Also, would recommend individual cluster running a different app-version so that the latest version can be scaled up and pushed to Production once tested and ready.

3. Spinning up a Spark cluster so that large batch files can be broken down into chunks and can be processed in parallel using **MPP (Massive Parallel Processing)** technique.

4. In order to make code updates, the **multi-master cluster** approach should work. Here would make code changes in 1 master cluster and during code updates, data would be loaded and refreshed in the other node. This will result in near-zero application downtime.