

# **Design and Implementation of a Real-Time Food Delivery System Using AWS Cloud Services**

DOCUMENTED PROJECT

BY

**Aayush Pandey**

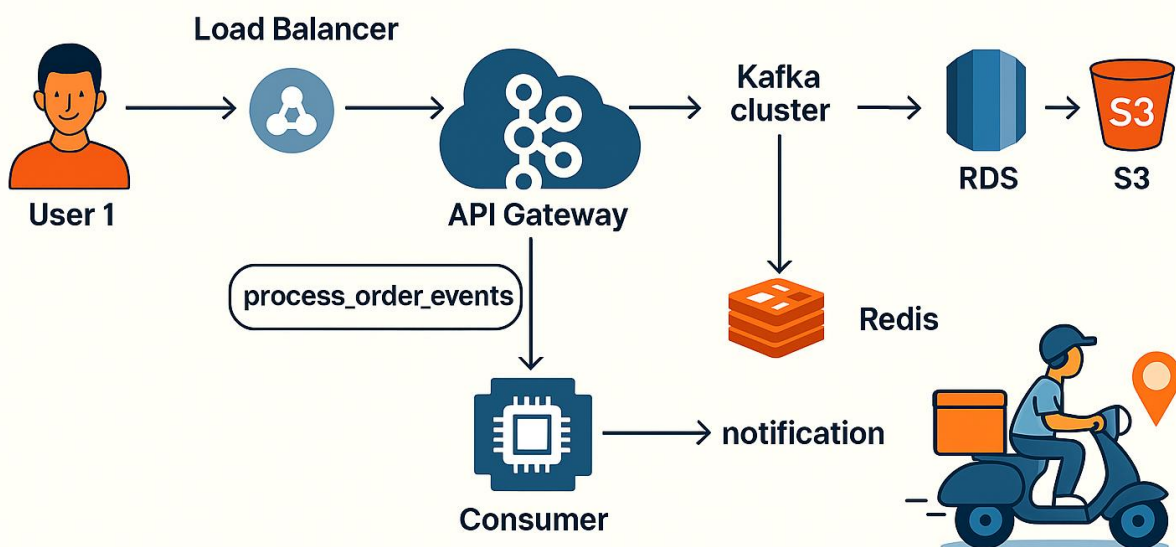
# Design and Implementation of a Real-Time Food Delivery System Using AWS Cloud Services

## # Description:

This project aims to provide an effective and scalable AWS-based architecture for a contemporary meal delivery service. To handle traffic spikes during peak hours, this platform needs dynamic resource scaling, smooth real-time order monitoring, and strong analytics for delivery optimization.

The design will use core AWS services (compute, storage, database, networking, and security) as well as advanced services like container orchestration, serverless functions, content delivery, and monitoring, with an emphasis on high availability, low latency, security, and cost-effectiveness.

This project will give participants practical experience in developing cloud-native solutions and implementing AWS best practices in high-demand, real-world applications. The foundation for more sophisticated, production-grade architectures in the field of logistics and food delivery will also be laid by it.



## # Basic AWS Services to be used

- **Amazon S3:** For storing static assets such as restaurant images, menu, user-uploaded content, and backup files.
- **Amazon EC2:** To host backend microservices and APIs (containerized using Docker) for core functionalities like order processing, user management, and payment handling.
- **AWS Identity and Access Management (IAM):** To securely manage access and permissions for users, services, and resources with restricted policies following the principle of least privilege
- **AWS VPC:** To create a secure and isolated network environment, managing subnets, route tables, NAT Gateways, and ensuring private/public segmentation for services.
- **Amazon RDS (Aurora):** As the relational database for transactional data, including customer information, order records, and payment history with high availability and read replicas for scaling.

## # Advanced AWS Services

- **Amazon MSK (Kafka Cluster):** For real-time event streaming like order placement, rider location updates, and streaming data pipelines for analytics or machine learning.
- **Amazon ElastiCache (Redis):** For caching and accelerating access to frequently used data, such as restaurant menus, delivery zones, or active user sessions.
- **AWS Lambda:** For executing serverless backend tasks such as real-time notifications (e.g., SMS, push notifications to users and delivery agents), processing order events, and triggering data pipelines for analytics or ML models.
- **AWS Auto Scaling:** To automatically adjust EC2 instances or containerized workloads within Amazon EKS, based on dynamic traffic and demand fluctuations (e.g., peak lunch and dinner hours or during promotional events).
- **Amazon Elastic Load Balancer:** To distribute incoming HTTP/HTTPS traffic efficiently across multiple EC2 instances or containers running microservices within Amazon EKS.
- **Amazon CloudWatch:** For comprehensive monitoring of microservices, Lambda functions, EC2 instances, and databases. Used for setting alarms, viewing logs, and ensuring system health during high-traffic periods.

## # **Scenario:**

A food delivery platform is expanding its services across multiple cities and is experiencing challenges due to high data processing demands, real-time tracking, unpredictable order surges during peak hours, and the need for secure, scalable infrastructure. The company requires AWS-based architecture that ensures real-time analytics, seamless user experience, and highly scalable backend services, all while optimizing for cost and security.

## # **Problem Statement:**

Design an AWS architecture to support a modern food delivery platform with real-time order tracking, dynamic scaling to handle high traffic surges, and advanced analytics for optimizing delivery operations and enhancing customer experience.

## # **Objectives:**

- To build a scalable, highly available, and secure cloud environment for a food delivery app.
- To enable real-time order tracking and delivery status updates.
- To handle dynamic workloads, including traffic spikes during lunch/dinner and promotional campaigns.
- To integrate machine learning models for route optimization and demand forecasting.
- To process large amounts of operational and customer data for analytics and insights.

## # **Outcomes:**

- The system automatically scales backend services based on real-time demand, reducing latency during peak hours.
- Real-time tracking and updates for users and delivery personnel using AWS services.
- A secure platform with robust user and payment data protection.
- Effective use of analytics and machine learning for delivery optimization and customer engagement.
- Optimized cost management by leveraging serverless components and auto-scaling.

## # Proposed AWS Components:

- **Amazon EC2:** For deploying core backend microservices as Docker containers or running workloads requiring dedicated compute capacity.
- **Amazon ElastiCache (Redis):** To cache frequently accessed data such as active user sessions, order statuses, and restaurant/restaurant menu details for faster response times.
- **Amazon Kafka Cluster:** For real-time event streaming including order placement, delivery agent location tracking, and data pipelines for real-time analytics or machine learning.
- **Amazon RDS (Aurora):** For managing transactional and relational data like customer information, orders, payments, and delivery records with high availability and scalability.
- **Amazon S3:** For storing static content like restaurant images, menus, user-uploaded content, and system backups.
- **Amazon CloudFront:** As a CDN to ensure fast, low-latency delivery of static assets globally to enhance user experience.
- **AWS Lambda:** To process asynchronous backend tasks such as sending push notifications, triggering workflows on S3 uploads, and consuming Kafka event streams.
- **AWS Auto Scaling:** To automatically scale EC2 instances based on varying workloads, ensuring high availability during traffic surges.
- **Elastic Load Balancer (ALB):** To distribute incoming traffic (API calls and web traffic) efficiently across EC2 instances or containerized services.
- **Amazon CloudWatch:** For centralized monitoring of service performance, resource usage, and real-time log management with automated alerts.
- **AWS IAM:** For secure identity and access management, enforcing strict permissions and roles across AWS services.

# # Solution

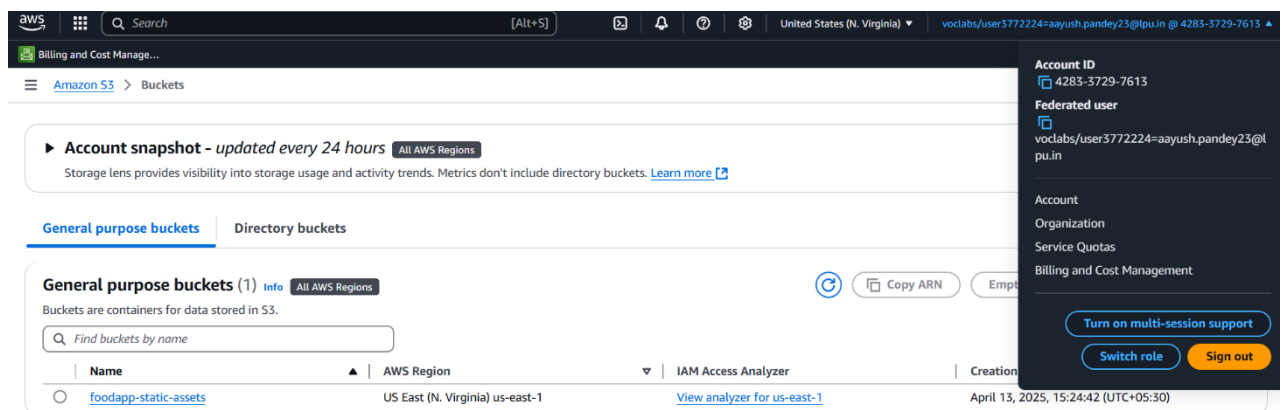
## ❖ Implementation: Hybrid Microservices + Serverless Approach

- This architecture leverages containerized microservices deployed on EC2, real-time event streaming with Kafka (Amazon MSK), and serverless functions (AWS Lambda) to enable high scalability, real-time order tracking, and efficient operations for a food delivery platform.

## ❖ Steps:

### 1. Frontend and Content Delivery:

- Use Amazon CloudFront integrated with Amazon S3 to cache and accelerate the delivery of static assets such as restaurant images, menus, and user-uploaded content globally.

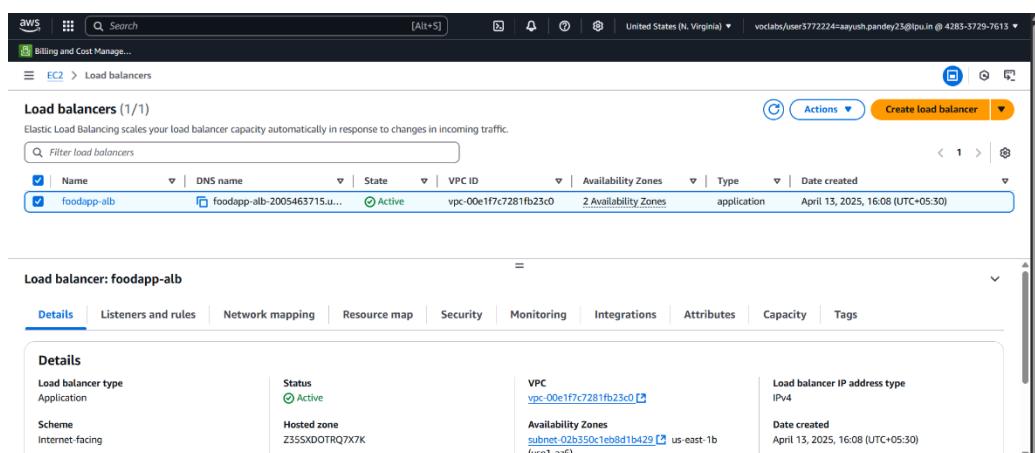
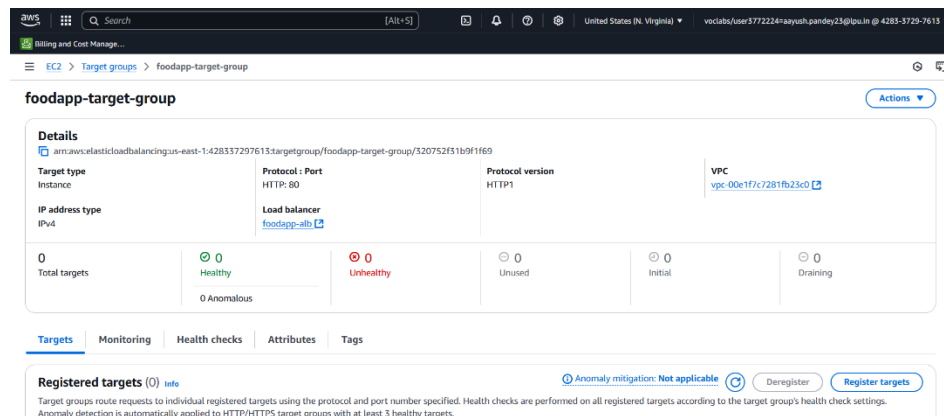


## 403 Forbidden

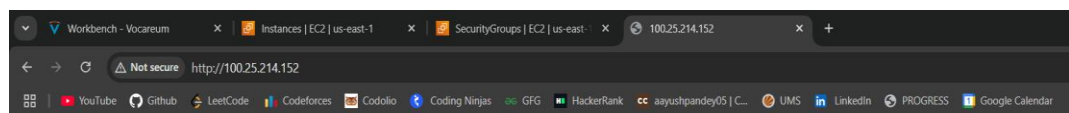
- Code: AccessDenied
- Message: Access Denied
- RequestId: SYJR3C01MGDNAP7T
- HostId: Re3hibmNGfLEzGSoeRz1iI4GkPOqkRt254caRkcqFSivcJfPHV2JBZYNPVarPiE8VUM4PSVV0A=

```
1 {
2   "Version": "2012-10-17",
3   "Statement": [
4     {
5       "Sid": "PublicReadGetObject",
6       "Effect": "Allow",
7       "Principal": "*",
8       "Action": "s3:GetObject",
9       "Resource": "arn:aws:s3:::foodapp-static-assets/*"
10    }
11  ]
12 }
13
```

- Deploy APIs behind an Elastic Load Balancer (ALB) to distribute incoming traffic efficiently to EC2-hosted microservices.



- EC2 Properly Hosting the API in the Backend



**FoodApp Backend API is Live**

## 2. Compute Resources:

- Deploy backend microservices (e.g., order management, user service, payment service) as Docker containers running on Amazon EC2.
- Docker Installation

```
Installing : libnetfilter_conntrack-1.0.8-2.amzn2023.0.2.x86_64 6/10
Installing : iptables-libs-1.8.8-3.amzn2023.0.2.x86_64 7/10
Installing : iptables-nft-1.8.8-3.amzn2023.0.2.x86_64 8/10
Running scriptlet: iptables-nft-1.8.8-3.amzn2023.0.2.x86_64 8/10
Installing : libxcproup-3.0-1.amzn2023.0.1.x86_64 9/10
Running scriptlet: docker-25.0.8-1.amzn2023.0.1.x86_64 10/10
Installing : docker-25.0.8-1.amzn2023.0.1.x86_64 10/10
Running scriptlet: docker-25.0.8-1.amzn2023.0.1.x86_64 10/10
Created symlink /etc/systemd/system/sockets.target.wants/docker.socket -> /usr/lib/systemd/system/docker.socket.

Verifying : containerd-1.7.27-1.amzn2023.0.1.x86_64 1/10
Verifying : docker-25.0.8-1.amzn2023.0.1.x86_64 2/10
Verifying : iptables-libs-1.8.8-3.amzn2023.0.2.x86_64 3/10
Verifying : iptables-nft-1.8.8-3.amzn2023.0.2.x86_64 4/10
Verifying : libxcproup-3.0-1.amzn2023.0.1.x86_64 5/10
Verifying : libnetfilter_conntrack-1.0.8-2.amzn2023.0.2.x86_64 6/10
Verifying : libnftnl-1.0.1-19.amzn2023.0.2.x86_64 7/10
Verifying : libnftnl-1.2.2-2.amzn2023.0.2.x86_64 8/10
Verifying : pigz-2.5-1.amzn2023.0.3.x86_64 9/10
Verifying : runc-1.2.4-1.amzn2023.0.1.x86_64 10/10

Installed:
containerd-1.7.27-1.amzn2023.0.1.x86_64 docker-25.0.8-1.amzn2023.0.1.x86_64 iptables-libs-1.8.8-3.amzn2023.0.2.x86_64
iptables-nft-1.8.8-3.amzn2023.0.2.x86_64 libnftnl-1.0.1-19.amzn2023.0.2.x86_64 libnftnl-1.2.2-2.amzn2023.0.2.x86_64
libnetfilter_conntrack-1.0.8-2.amzn2023.0.2.x86_64 libxcproup-3.0-1.amzn2023.0.1.x86_64 pigz-2.5-1.amzn2023.0.3.x86_64
runc-1.2.4-1.amzn2023.0.1.x86_64

Complete!
Created symlink /etc/systemd/system/multi-user.target.wants/docker.service -> /usr/lib/systemd/system/docker.service.
```

- Order-Service

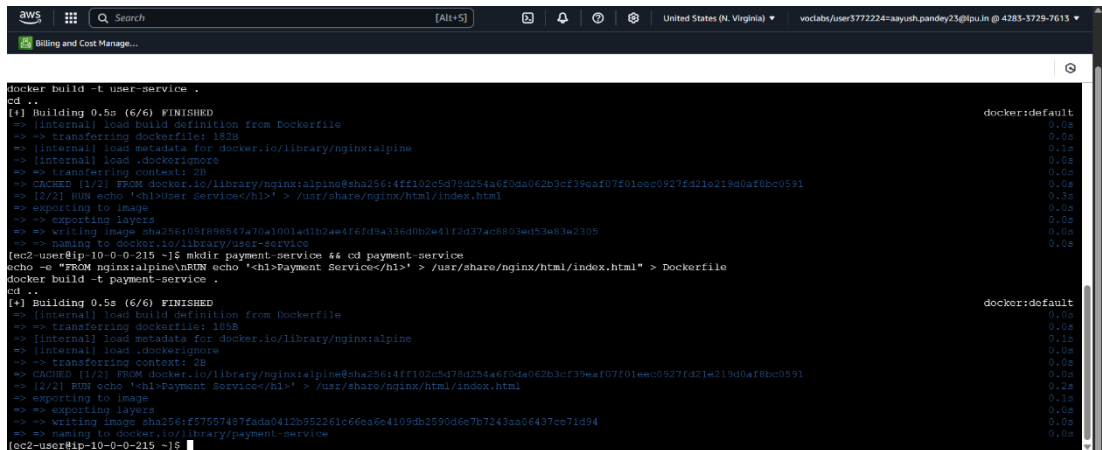
```
>>> Transferring dockerfile: 183B 0.0s
>>> [internal] load metadata for docker.io/library/nginx:alpine 0.3s
>>> [internal] load .dockerignore 0.0s
>>> >>> Transferring context: 2B 0.0s
>>> [1/2] FROM docker.io/library/nginx:alpine:sha256:4ff102c5d78d254a6f0da062b3cf39eaf07f01e0c0927fd21e219d0af8bc0591 1.5s
>>> resolve docker.io/library/nginx:alpine:sha256:4ff102c5d78d254a6f0da062b3cf39eaf07f01e0c0927fd21e219d0af8bc0591 0.0s
>>> sha256:1ff4b44caebcfd17e0114fa9904a570ab5ab8e64457855f6c6c0bafca07 11.2kB / 11.2kB 0.0s
>>> sha256:ccc39e3d420d6d115290f10741f6dad1c474b6e94897434ebcdd3be731d2 1.79MB / 1.79MB 0.1s
>>> sha256:4ff102c5d78d254a6f0da062b3cf39eaf07f01e0c0927fd21e219d0af8bc0591 10.36kB / 10.36kB 0.0s
>>> sha256:1a71e08847f115ecf5d6c062b7d46b54ad63f0cc1b8aa7e705f739a97c2fc 2.50kB / 2.50kB 0.0s
>>> sha256:1f8232174bc91741fdd3da96d85011032101a032a93a388b79e99e69c2d5c870 3.64MB / 3.64MB 0.1s
>>> sha256:432ec460bdf5babe56d7e13b960b308f42c0f2103f6665c349f0b9dac8962 627B / 627B 0.1s
>>> extracting sha256:1f8232174bc91741fdd3da96d85011032101a032a93a388b79e99e69c2d5c870 0.3s
>>> sha256:1c74e4c092ab716c99e1a8049434984ac5e65d1alc74486528b103b8e67a 15.38MB / 15.38MB 0.5s
>>> sha256:984583bcf083fa6900b5e7834795a9a57a9b4dfe7448d5350474f5d309625ece 955B / 955B 0.1s
>>> sha256:8d27c072a58f1ecf2425172ac0e5b25010f2d014f89de35b90104e46256eb 402B / 402B 0.0s
>>> sha256:ab3286a7346303a31b69a5189f63f1414cc1de44e397088dcd07edb322df1fe9 1.21kB / 1.21kB 0.0s
>>> sha256:6d79cc084d434876ce0f038c675d20532f28e47623a29e7e63bc0bf13a4ed6 1.40kB / 1.40kB 0.0s
>>> extracting sha256:ccc39e3d420d6d115290f10741f6dad1c474b6e94897434ebcdd3be731d2 0.2s
>>> extracting sha256:432ec460bdf5babe56d7e13b960b308f42c0f2103f6665c349f0b9dac8962 0.0s
>>> extracting sha256:984583bcf083fa6900b5e7834795a9a57a9b4dfe7448d5350474f5d309625ece 0.0s
>>> extracting sha256:8d27c072a58f1ecf2425172ac0e5b25010f2d014f89de35b90104e46256eb 0.0s
>>> extracting sha256:ab3286a7346303a31b69a5189f63f1414cc1de44e397088dcd07edb322df1fe9 0.0s
>>> extracting sha256:6d79cc084d434876ce0f038c675d20532f28e47623a29e7e63bc0bf13a4ed6 0.0s
>>> extracting sha256:1c74e4c092ab716c99e1a8049434984ac5e65d1alc74486528b103b8e67a 0.0s
[2/2] RUN echo <html>Order Service</html> > /usr/share/nginx/html/index.html 0.4s
>>> exporting to image 0.0s
>>> exporting layers 0.0s
>>> writing image sha256:63256c5ae08a832f25ed2c2e608b1e638d376a0849b5cc1c232444fe878225 0.0s
>>> naming to docker.io/library/order-service 0.0s
ec2-user@ip-10-0-0-215 ~$
```

- User-Service

```
>>> sha256:ab3286a7346303a31b69a5189f63f1414cc1de44e397088dcd07edb322df1fe9 1.21kB / 1.21kB 0.2s
>>> sha256:6d79cc084d434876ce0f038c675d20532f28e47623a29e7e63bc0bf13a4ed6 1.40kB / 1.40kB 0.2s
>>> extracting sha256:ccc39e3d420d6d115290f10741f6dad1c474b6e94897434ebcdd3be731d2 0.2s
>>> extracting sha256:432ec460bdf5babe56d7e13b960b308f42c0f2103f6665c349f0b9dac8962 0.0s
>>> extracting sha256:984583bcf083fa6900b5e7834795a9a57a9b4dfe7448d5350474f5d309625ece 0.0s
>>> extracting sha256:8d27c072a58f1ecf2425172ac0e5b25010f2d014f89de35b90104e46256eb 0.0s
>>> extracting sha256:ab3286a7346303a31b69a5189f63f1414cc1de44e397088dcd07edb322df1fe9 0.0s
>>> extracting sha256:1c74e4c092ab716c99e1a8049434984ac5e65d1alc74486528b103b8e67a 0.0s
>>> extracting sha256:07e4c092ab716c99e1a8049434984ac5e65d1alc74486528b103b8e67a 0.0s
[2/2] RUN echo <html>Order Service</html> > /usr/share/nginx/html/index.html 0.4s
>>> exporting to image 0.0s
>>> exporting layers 0.0s
>>> writing image sha256:63256c5ae08a832f25ed2c2e608b1e638d376a0849b5cc1c232444fe878225 0.0s
>>> naming to docker.io/library/user-service 0.0s
ec2-user@ip-10-0-0-215 ~$ mkdir user-service && cd user-service
ec2-user@ip-10-0-0-215 ~$ echo -e "FROM nginx:alpine\nRUN echo <html>User Service</html> > /usr/share/nginx/html/index.html" > Dockerfile
ec2-user@ip-10-0-0-215 ~$ docker build -t user-service .
[+] Building 0.5s (6/6) FINISHED docker:default
>>> [internal] load build definition from Dockerfile 0.0s
>>> >>> Transferring dockerfile: 182B 0.0s
>>> [internal] load metadata for docker.io/library/nginx:alpine 0.1s
>>> [internal] load .dockerignore 0.0s
>>> >>> Transferring context: 2B 0.0s
>>> CACHED [1/2] FROM docker.io/library/nginx:alpine:sha256:4ff102c5d78d254a6f0da062b3cf39eaf07f01e0c0927fd21e219d0af8bc0591 0.0s
>>> [2/2] RUN echo <html>User Service</html> > /usr/share/nginx/html/index.html 0.3s
>>> exporting to image 0.0s
>>> exporting layers 0.0s
>>> writing image sha256:09f998547a70a1b01ad2bae46fd9a336d0b2e41f2d37ac803ed5e83e2305 0.0s
>>> naming to docker.io/library/user-service 0.0s
ec2-user@ip-10-0-0-215 ~$
```



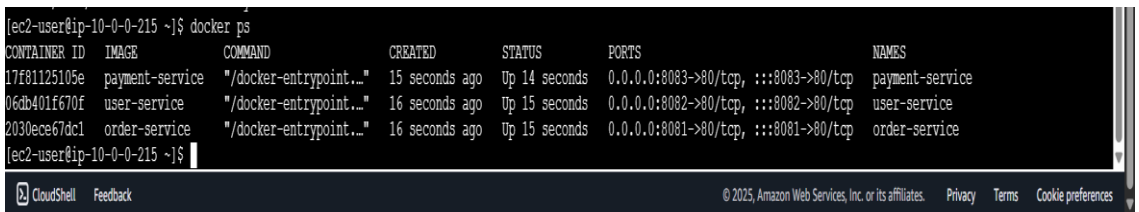
- Payment-Service



```
aws
[+] Search [Alt+S] United States (N. Virginia) voclabs/user3772224=aayush.pandey23@lpu.in @ 4283-3729-7613
Billing and Cost Manage...

docker build -t user-service .
cd ..
[+] Building 0.5s (6/6) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 152B
=> [internal] load metadata for docker.io/library/nginx:alpine
=> [internal] load .dockerignore
=> => transferring context: 2B
=> CACHED [1/2] FROM docker.io/library/nginx:alpine@sha256:4ff102c5d78d254a6f0da062b3cf39eaf07f01eac0927fd21e219d0af8bc0591
=> [2/2] RUN echo '<html>Payment Service</html>' > /usr/share/nginx/html/index.html
=> exporting to image
=> => exporting layers
=> => writing image sha256:f898547a70a1001ad1b2ae4f6f49a336d0b2e41f2d37ac8803ed53e83e2305
=> => naming to docker.io/library/user-service
[ec2-user@ip-10-0-0-215 ~]$ mkdir payment-service
echo -e "FROM nginx:alpine\nRUN echo '<html>Payment Service</html>' > /usr/share/nginx/html/index.html" > Dockerfile
docker build -t payment-service .
cd ..
[+] Building 0.5s (6/6) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 152B
=> [internal] load metadata for docker.io/library/nginx:alpine
=> [internal] load .dockerignore
=> => transferring context: 2B
=> CACHED [1/2] FROM docker.io/library/nginx:alpine@sha256:4ff102c5d78d254a6f0da062b3cf39eaf07f01eac0927fd21e219d0af8bc0591
=> [2/2] RUN echo '<html>Payment Service</html>' > /usr/share/nginx/html/index.html
=> exporting to image
=> => exporting layers
=> => writing image sha256:f87567487fada0412b952261c66ea6e4109db2590d6e7b7243aa06437ce71d94
=> => naming to docker.io/library/payment-service
[ec2-user@ip-10-0-0-215 ~]$
```

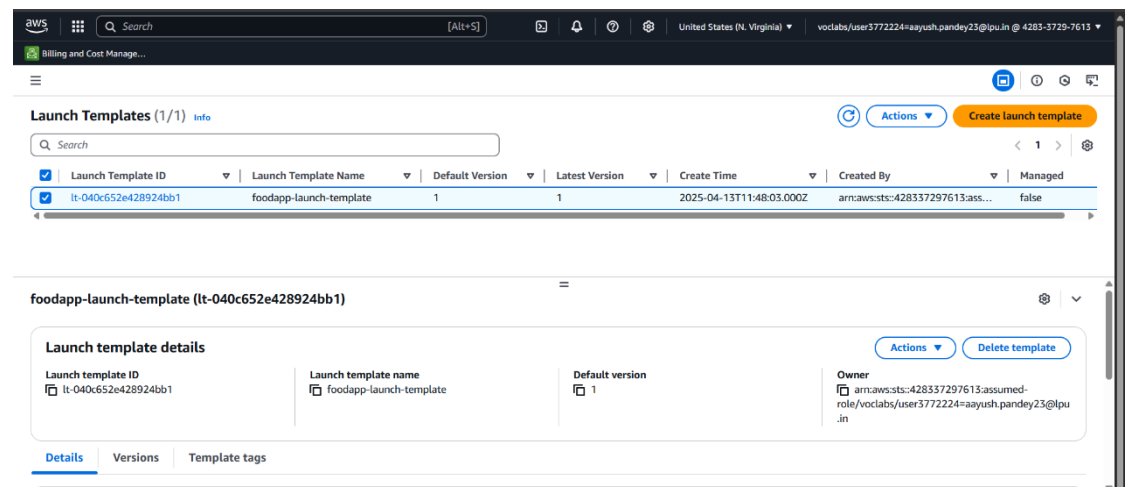
- All Microservices Running in Docker Containers



```
[ec2-user@ip-10-0-0-215 ~]$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                               NAMES
17f81125105e   payment-service "/docker-entrypoint..." 15 seconds ago Up 14 seconds 0.0.0.0:8083->80/tcp, :::8083->80/tcp payment-service
06db401f670f   user-service   "/docker-entrypoint..." 16 seconds ago Up 15 seconds 0.0.0.0:8082->80/tcp, :::8082->80/tcp user-service
2030ece67dc1   order-service  "/docker-entrypoint..." 16 seconds ago Up 15 seconds 0.0.0.0:8081->80/tcp, :::8081->80/tcp order-service
[ec2-user@ip-10-0-0-215 ~]$
```

- Configure AWS Auto Scaling Groups to dynamically adjust the number of EC2 instances based on real-time traffic patterns (e.g, lunch/dinner surges or promotional events).

- Launch Template



**Launch Templates (1/1)**

Launch Template ID	Launch Template Name	Default Version	Latest Version	Create Time	Created By	Managed
lt-040c652e428924bb1	foodapp-launch-template	1	1	2025-04-13T11:48:03.000Z	arn:aws:sts:428337297613:ass...	false

**foodapp-launch-template (lt-040c652e428924bb1)**

**Launch template details**

Launch template ID lt-040c652e428924bb1	Launch template name foodapp-launch-template	Default version 1	Owner arn:aws:sts:428337297613:assumed-role/voclabs/user3772224=aayush.pandey23@lpu.in
--	---	----------------------	---

[Details](#) [Versions](#) [Template tags](#)

aws | [Alt+S] | United States (N. Virginia) | voclabs/user3772224=aayush.pandey23@lpu.in @ 4283-3729-7613

Billing and Cost Manage...

Launch Templates (1/1) Info

Search

Actions Create launch template

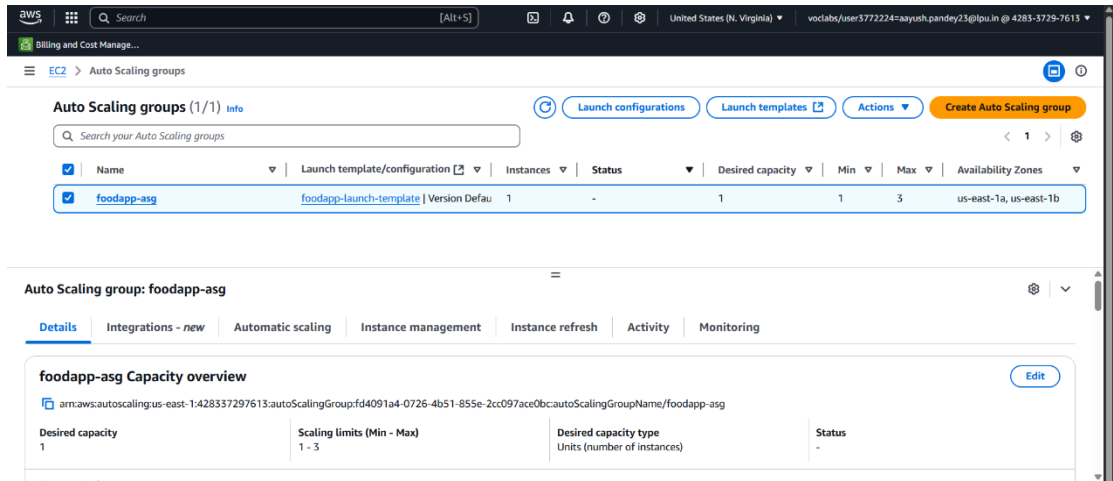
Launch Template ID	Launch Template Name	Default Version	Latest Version	Create Time	Created By	Managed
lt-040c652e428924bb1	foodapp-launch-template	1	1	2025-04-13T11:48:03.000Z	arn:aws:sts:428337297613:ass...	false

foodapp-launch-template (lt-040c652e428924bb1)

Launch template details

Launch template ID lt-040c652e428924bb1	Launch template name foodapp-launch-template	Default version 1	Owner arn:aws:sts:428337297613:assumed-role/voclabs/user3772224=aayush.pandey23@lpu.in
--	---	----------------------	---

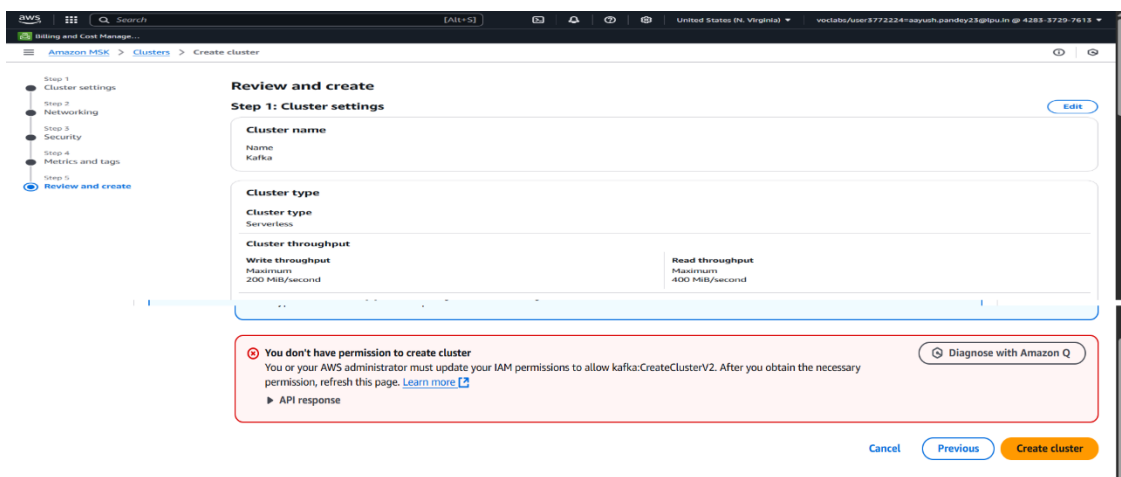
Details Versions Template tags



- Auto Scaling Group Configured with Load-Based Scaling

### 3. Real-time Event Streaming & Processing:

- Deploy a Kafka Cluster (Amazon MSK) to manage real-time event streaming for use cases like order lifecycle events, rider location tracking, and user notifications.
- Kafka Cluster Configuration (Amazon MSK)



- Use AWS Lambda to consume events from the Kafka cluster, perform lightweight backend operations (e.g., triggering notifications, updating order statuses), and integrate with other services.
- Lambda Function with Existing IAM Role

**Create function** [Info](#)

Choose one of the following options to create your function.

☒ **Author from scratch**  
Start with a simple Hello World example.

☐ **Use a blueprint**  
Build a Lambda application from sample code and configuration presets for common use cases.

☐ **Container image**  
Select a container image to deploy for your function.

---

**Basic information**

**Function name**  
Enter a name that describes the purpose of your function.  
KafkaConsumerFunction  
Function name must be 1 to 64 characters, must be unique to the Region, and can't include spaces. Valid characters are a-z, A-Z, 0-9, hyphens (-), and underscores (\_).

**Runtime** [Info](#)  
Choose the language to use to write your function. Note that the console code editor supports only Node.js, Python, and Ruby.  
Python 3.12

**Architecture** [Info](#)  
Choose the instruction set architecture you want for your function code.  
☒ x86\_64  
☐ arm64

**Permissions** [Info](#)

- Used the test event to simulate a Kafka message and verify that our Lambda function works correctly—even without a live Kafka stream

**Executing function: succeeded** [\(logs\)](#)

**Test event** [Info](#) [Delete](#) [CloudWatch Logs Live Tail](#) [Save](#) [Test](#)

To invoke your function without saving an event, modify the event, then choose Test. Lambda uses the modified event to invoke your function, but does not overwrite the original event until you choose Save.

**Test event action**  
☐ Create new event ☒ Edit saved event

**Event name**  
KafkaSimTest

**Event JSON** [Format JSON](#)

```

1 {
2   "topic": "order-events",
3   "eventType": "OrderPlaced",
4   "orderId": "12345",
5   "status": "pending"
6 }
7
  
```

- Cloud Watch Logs

**/aws/lambda/KafkaConsumerFunction** [Actions](#) [View in Logs Insights](#) [Start tailing](#) [Search log group](#)

**Log group details**

<b>Log class</b> <a href="#">Info</a> Standard  <b>ARN</b> arn:aws:logs:us-east-1:428337297613:log-group:/aws/lambda/KafkaConsumerFunction:  <b>Creation time</b> 6 minutes ago  <b>Retention</b> Never expire  <b>Stored bytes</b> -	<b>Metric filters</b> 0  <b>Subscription filters</b> 0  <b>Contributor Insights rules</b> -  <b>KMS key ID</b> -  <b>Anomaly detection</b> <a href="#">Configure</a>	<b>Data protection</b> -  <b>Sensitive data count</b> -  <b>Field indexes</b> <a href="#">Configure</a>  <b>Transformer</b> <a href="#">Configure</a>
--	---	---

[Log streams](#) [Tags](#) [Anomaly detection](#) [Metric filters](#) [Subscription filters](#) [Contributor Insights](#) [Data protection](#) [Field indexes - new](#) [Transformer - new](#)

- Cloud Watch Events log

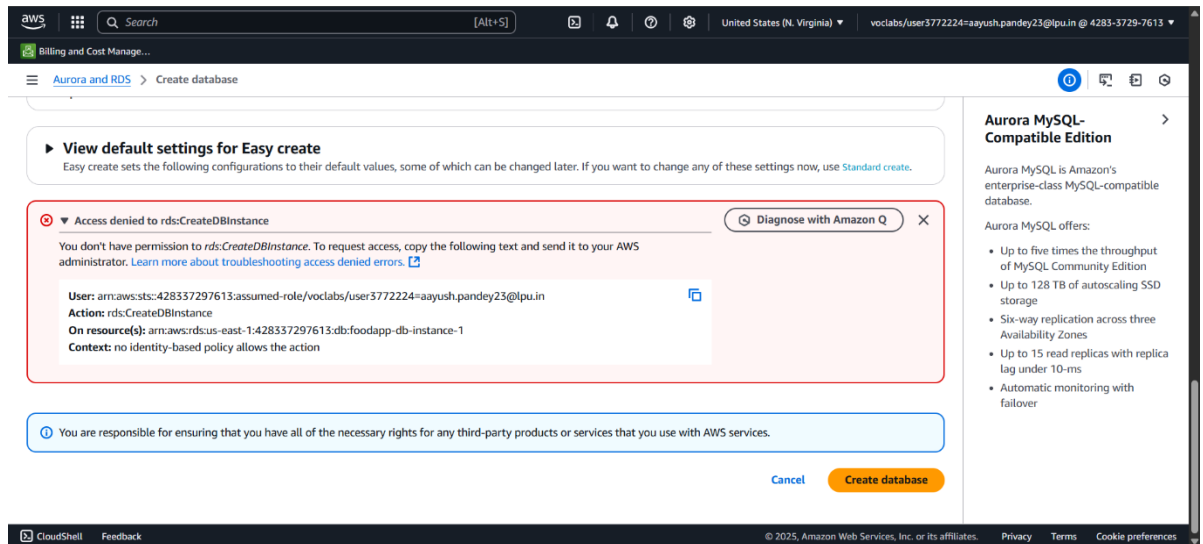
The screenshot shows the AWS CloudWatch Logs console. The breadcrumb navigation is: CloudWatch > Log groups > /aws/lambda/KafkaConsumerFunction > 2025/04/13/[LATEST]72f43182574d41b9ab9153409400cb74. The 'Log events' section is active, displaying a message: 'No more records within selected time range Retry'. Below this, a log event is shown with the following details: INIT\_START Runtime Version: python:3.12.v56, Runtime Version ABI: arn:aws:lambda:us-east-1::runtime:40d617d2be93996819b6c44f30965daf9adf07d58603c070365dbdc7e1d8db16, START RequestId: f558d504-a22b-401d-b862-4593380a91bb, Version: \$LATEST, END RequestId: f558d504-a22b-401d-b862-4593380a91bb, REPORT RequestId: f558d504-a22b-401d-b862-4593380a91bb, Duration: 1.81 ms, Billed Duration: 2 ms, Memory Size: 128 MB, Max Memory Used: 32 MB, Init Duration: 87.78 ms. The message 'No more records within selected time range Auto retry paused. Resume' is displayed at the bottom.

- Lambda Function Successfully Processed Kafka Order Event

The screenshot shows the AWS CloudWatch Logs console. The breadcrumb navigation is: CloudWatch > Log groups > /aws/lambda/KafkaConsumerFunction > 2025/04/13/[LATEST]b4fda013731342819ef0ea537dc83564. The 'Log events' section is active, displaying a list of log events. The first event is a START event with RequestId: 63cc4490-9c3c-4ed9-9b0f-c1105b17579d and Version: \$LATEST. The second event is a RECEIVED event with RequestId: 63cc4490-9c3c-4ed9-9b0f-c1105b17579d, timestamp 2025-04-13T18:16:01.930+05:30, and message: 'Received event: {"topic": "order-events", "eventType": "OrderPlaced", "orderId": "12345", "status": "pending"}'. The third event is an END event with RequestId: 63cc4490-9c3c-4ed9-9b0f-c1105b17579d. The fourth event is a REPORT event with RequestId: 63cc4490-9c3c-4ed9-9b0f-c1105b17579d, timestamp 2025-04-13T18:16:01.943+05:30, and message: 'REPORT RequestId: 63cc4490-9c3c-4ed9-9b0f-c1105b17579d Duration: 2.08 ms Billed Duration: 3 ms Memory Size: 128 MB Max Memory Used: 32 MB Init Duration: 93.83 ms'. The message 'No newer events at this moment. Auto retry paused. Resume' is displayed at the bottom. The 'Back to top' button is visible in the bottom right corner.

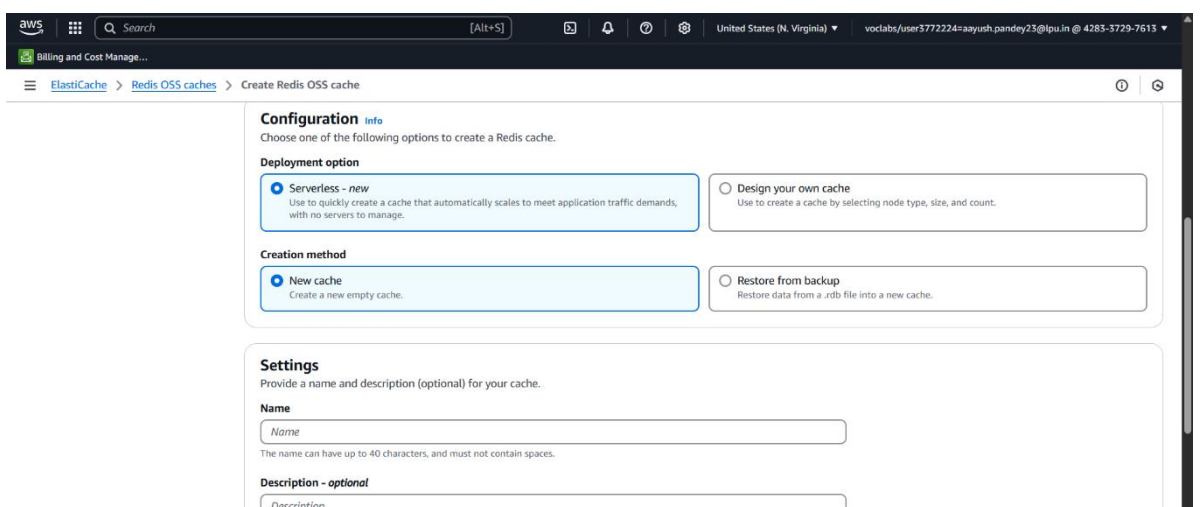
## 4. Database Configuration:

- Deploy Amazon RDS (Aurora) to handle relational transactional data such as user profiles, order history, and payment transactions.
- RDS creation blocked due to sandbox IAM policy restrictions



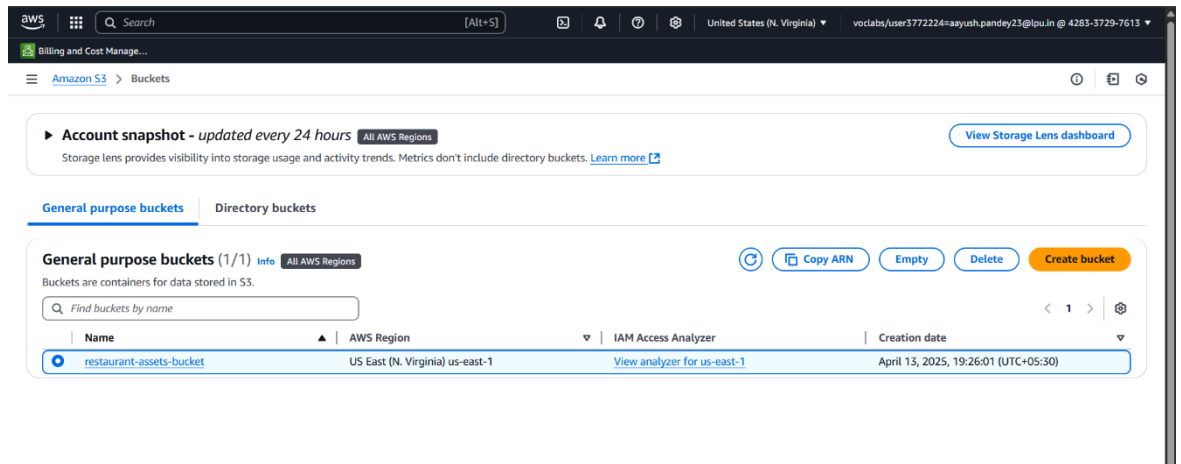
- Use Amazon ElastiCache (Redis) to cache frequently accessed data like user sessions, restaurant information, and active order statuses to reduce database load and improve response times.

- Permission Denied While Creating Redis OSS Cache (ElastiCache)

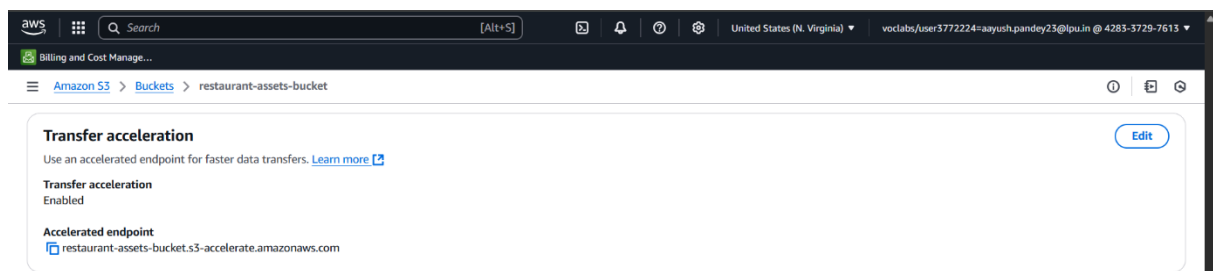


## 5. Storage and Backup:

- Store static assets such as restaurant images, menus, user-uploaded files, and system backups in Amazon S3.
- S3 Bucket Created for Static Asset Storage and Backups

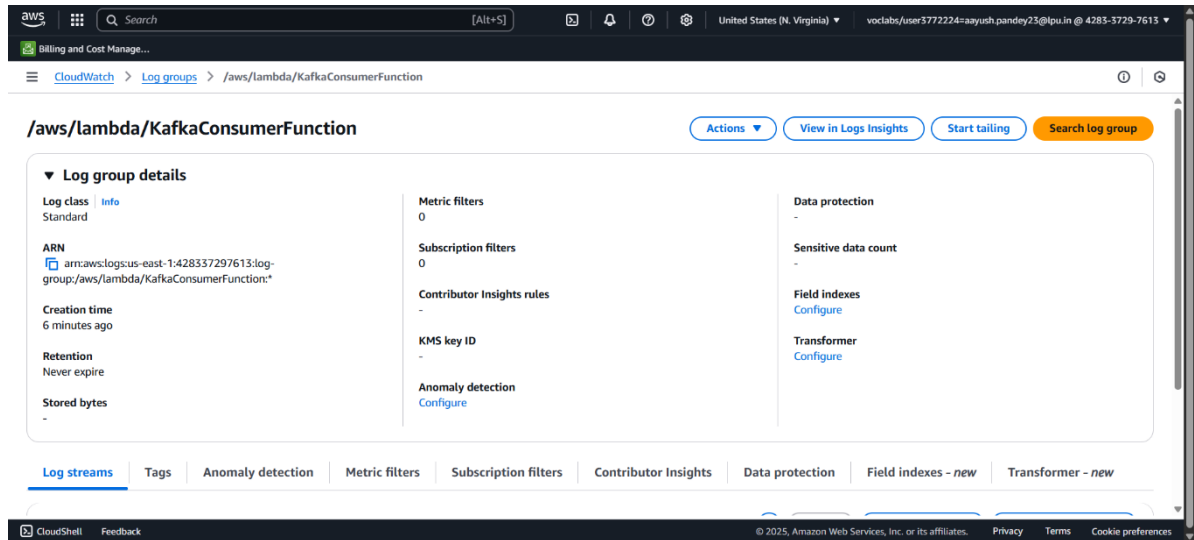


- Enable S3 Transfer Acceleration to speed up content upload/download across geographically dispersed users.
- S3 Transfer Acceleration Enabled for Faster Global Access

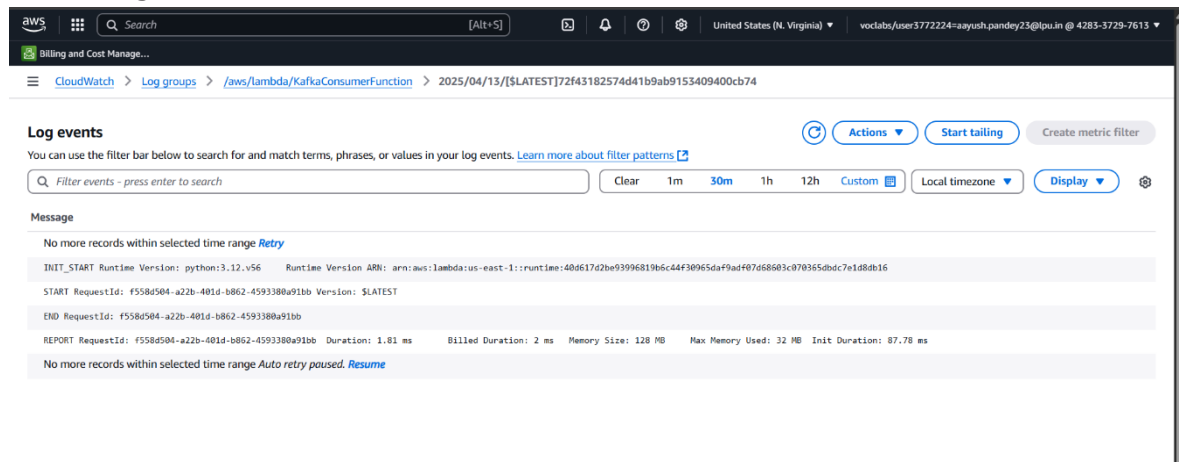


## 6. Monitoring and Cost Optimization:

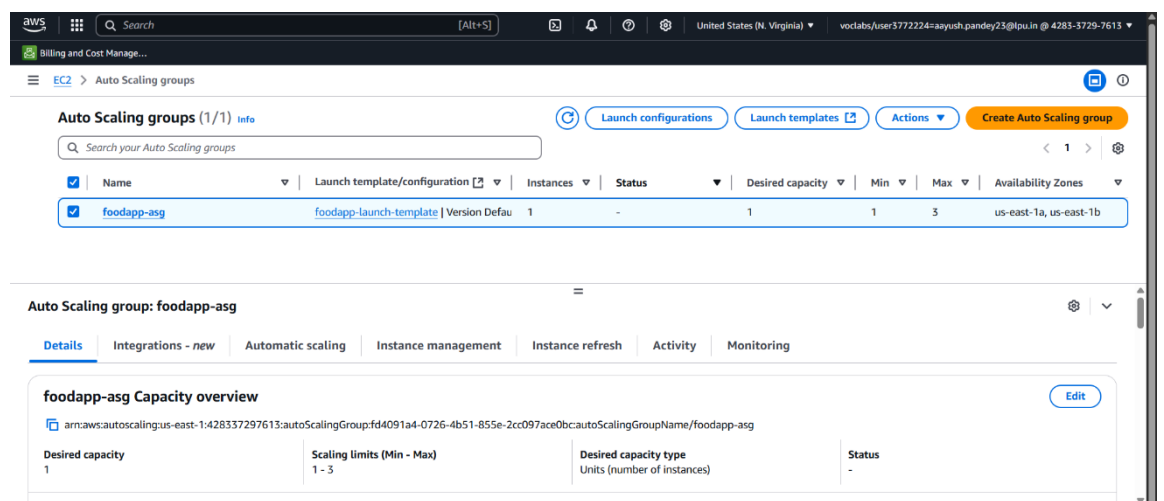
- Configure Amazon CloudWatch to monitor EC2 instances, Lambda functions, Kafka streams (MSK), and Redis cache performance, with dashboards and automated alarms.
- CloudWatch Monitor



- Event Logs

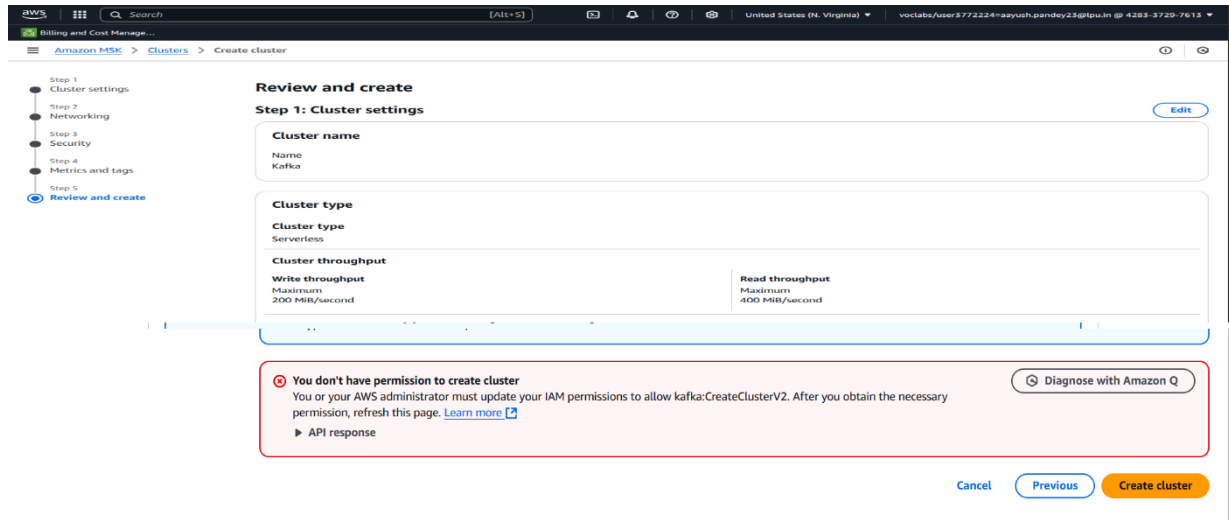


- Optimize costs using Auto Scaling for EC2 instances and AWS Lambda for on-demand, event-driven processes to reduce infrastructure overhead.
- Autoscaling



## 7. Security and Access Management:

- Enforce strict access control policies across all services using AWS IAM roles and policies to ensure secure service-to-service and user-to-service communications.
- IAM Restriction Demonstrated on Amazon MSK Cluster Creation



## ❖ Implementation: Serverless and Container-Based Approach

- This approach leverages serverless technologies (AWS Lambda) combined with containerized microservices on EC2 instances and real-time data streaming via Kafka (Amazon MSK) to achieve high scalability, flexibility, and cost optimization.

### ❖ Steps:

#### 1. Frontend and Content Delivery:

- Use Amazon CloudFront integrated with Amazon S3 for caching and accelerating the global delivery of static assets such as restaurant images, menus, and user-uploaded content.
- Configure Elastic Load Balancer (ALB) to distribute incoming API requests to backend microservices hosted on EC2 instances.

#### 2. Serverless & Compute Resources:

- Deploy backend microservices (e.g., order management, authentication, payment services) as Docker containers running on Amazon EC2 instances.
- Create an Auto Scaling Group to dynamically adjust EC2 instances based on real-time traffic demands.
- Use AWS Lambda to handle event-driven backend functions such as sending order notifications, triggering real-time updates, or lightweight background tasks.



### **3. Real-time Event Streaming:**

- Set up a Kafka Cluster (Amazon MSK) to stream real-time events such as order placement, rider location updates, and notifications.
- Use AWS Lambda to consume events from the Kafka cluster and process them asynchronously (e.g., notify customers of order status changes or trigger workflows).

### **4. Database Configuration:**

- Deploy Amazon RDS (Aurora) for managing relational data including customer profiles, order history, and payment transactions.
- Use Amazon ElastiCache (Redis) to cache frequently accessed data such as active orders, session data, and restaurant information to improve application performance.

### **5. Storage and Backup:**

- Store static assets and system backups in Amazon S3.
- Enable S3 Transfer Acceleration to enhance file transfer speeds for both uploads and downloads globally.
- 

### **6. Monitoring and Cost Optimization:**

- Configure Amazon CloudWatch to monitor EC2 instances, Lambda functions, Redis cache, and Kafka streaming pipelines.
- Use AWS Auto Scaling to manage EC2 instance capacity dynamically based on traffic and optimize costs by leveraging AWS Lambda for event-driven processes.

### **7. Security and Access Management:**

- Implement AWS IAM to apply role-based access control (RBAC) and securely manage permissions for all services and users.