

Università degli Studi di Milano

Subject : Cloud computing

Project : Secure and Resilient Kafka-Based
Messaging System with Istio-Driven Security
and HA.

Developed by : Banda Tharun Reddy 51793A ,
Tesfaamanuel Dekebo 41533A.

1. Project Overview

This project focuses on building and deploying a **secure, fault-tolerant, and highly available distributed messaging system** using **Apache Kafka** on a **Kubernetes cluster**. The core objective is to demonstrate key **non-functional properties** (NFRs) of distributed systems—such as resilience, scalability, security, and observability—through a practical microservice architecture.

Kafka serves as the backbone for asynchronous communication between two microservices: a **producer** and a **consumer**, both running as **Kubernetes deployments** to ensure high availability and auto-recovery. Kafka itself is deployed in a **highly available setup** with **three controller nodes**, coordinated using the **Raft consensus protocol** to maintain cluster state and ensure fault tolerance during broker failures.

To secure inter-service communication and simplify certificate management, the project integrates **Istio** as a service mesh. Istio enforces **mutual TLS (mTLS)** across the cluster, handles **automatic certificate provisioning**, and applies **fine-grained access control** via AuthorizationPolicies—eliminating the need for manual TLS setup in each microservice.

For **data persistence**, Kafka brokers are provisioned with **PersistentVolumeClaims (PVCs)** managed via **Helm charts**, ensuring reliable storage across restarts. Additionally, a robust **observability stack**—including **Prometheus**, **Grafana**, and **Kiali** (deployed using Istio add-ons)—provides real-time monitoring, metrics collection, and mesh-level traffic visualization.

This end-to-end setup demonstrates how modern cloud-native tools can be combined to achieve a production-grade message queue system with strong guarantees around **reliability**, **security**, and **operational transparency**.

2. Objectives

- Deploy a fault-tolerant, secure Kafka cluster on Kubernetes.
 - Demonstrate Kafka message flow between producer and consumer microservices.
 - Apply Istio to enforce **mTLS**, **encryption**, and **authentication** without manually managing TLS certificates.
 - Leverage observability tools to **monitor cluster state**, **message flow**, and **system health**.
 - Simulate failure conditions to validate resilience and HA mechanisms.
-

3. Architecture and Technology Stack

Technologies Used

Component	Technology
Messaging Queue	Apache Kafka (3 controller nodes, Raft)
Container Orchestration	Kubernetes
Service Mesh	Istio
Microservices	Kafka Producer/Consumer (Bitnami image)
Kafka Docker Image	<code>docker.io/bitnami/kafka:latest</code>
Deployment	Helm charts, YAML configs
Storage	Kubernetes PVCs
Observability	Kiali, Prometheus, Grafana

4. System Architecture

The Kafka cluster comprises:

- **3 Kafka controller nodes** coordinated using the **Raft protocol** to ensure high availability and consistent metadata management.
- **Persistent Volumes (PVs)** provisioned using **PVCs** for Kafka broker and controller state persistence.
- Kafka deployed via **Helm charts**, enabling modular and scalable configurations.
- **Two Kafka microservices: a producer and a consumer**, deployed as deployment files in K8's each of them to ensure auto healing and fault tolerance , when interacting with Kafka .
- Istio sidecars injected into all pods for secure mesh communication.
- **Monitoring stack with Prometheus, Grafana, and Kiali** for observability.

5. Kafka Deployment Details

5.1 Helm-Based Deployment

- Kafka was installed using **Bitnami Helm charts**, allowing:
 - Parameterized customization (replica count, storage size, etc.)
 - Declarative configuration of brokers and controllers
 - Simplified PVC setup for persistent data

```

# Steps for deployment of kafka KRaft using Helm

#1. Bitnami Helm Repo

helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo update

#kafka-values.yaml
# Creating a values.yaml for Kafka with KRaft

replicaCount: 2
persistence:
  enabled: true
  size: 8Gi
  storageClass: standard
kraft:
  enabled: true
  controllerQuorumVoters: "0@kafka-0.kafka-headless.default.svc.cluster.local:9093,1@kafka-1.kafka-headless.default.svc.cluster.local:9093"
  listeners:
    client:
      protocol: PLAINTEXT
    controller:
      protocol: PLAINTEXT
  auth:
    enabled: false
  zookeeper:
    enabled: false

# 3 Installing Kafka using Helm in Default Namespace
helm install kafka bitnami/kafka -n default -f kafka-values.yaml

```

5.2 Raft-Based Controller Nodes

- Kafka cluster deployed with **3 dedicated controller nodes** running **Raft protocol**, providing:
 - Metadata fault tolerance
 - Improved failover during controller node failure
 - Centralized controller architecture

```
(base) THARUN@MacBook-Pro-M2 ~ % kubectl get pods
NAME                               READY   STATUS    RESTARTS   AGE
kafka-client-5dbb5f6cc9-596gz     2/2     Running   7 (48m ago)  13h
kafka-client-5dbb5f6cc9-76k21     2/2     Running   8 (48m ago)  13h
kafka-controller-0                2/2     Running   20 (13h ago) 12d
kafka-controller-1                2/2     Running   20 (13h ago) 12d
kafka-controller-2                2/2     Running   20 (13h ago) 12d
kafka-producer-fc49cccc8-jfj7f    2/2     Running   0          28m
(base) THARUN@MacBook-Pro-M2 ~ %
```

5.3 Persistent Storage

- Used **PVCs** for both brokers and controllers to ensure:
 - **Data durability** in the event of pod rescheduling
 - StatefulSets ensured consistent pod identities
 - Underlying storage provisioned using dynamic volume claims

```
(base) THARUN@MacBook-Pro-M2 ~ % kubectl get pv
NAME           CAPACITY   ACCESS MODES  RECLAIM POLICY  STATUS   CLAIM                                     STORAGECLASS  VOLUMEATTRIBUTESCLASS  REASON  AGE
pvc-0f1afa8f-4b35-46f8-8109-f3bb7cd22f77  8Gi        RWO          Delete        Bound    default/data-kafka-controller-1  standard    <unset>          12d
pvc-7c12a68b-d977-46c8-b43e-6afe8036ba9b  8Gi        RWO          Delete        Bound    default/data-kafka-controller-0  standard    <unset>          12d
pvc-f43066f3-5e15-4d9d-bf36-5ba04f02e8d5  8Gi        RWO          Delete        Bound    default/data-kafka-controller-2  standard    <unset>          12d
(base) THARUN@MacBook-Pro-M2 ~ % kubectl get pvc
NAME           STATUS   VOLUME                                     CAPACITY   ACCESS MODES  STORAGECLASS  VOLUMEATTRIBUTESCLASS  AGE
data-kafka-controller-0  Bound   pvc-7c12a68b-d977-46c8-b43e-6afe8036ba9b  8Gi        RWO          standard    <unset>          12d
data-kafka-controller-1  Bound   pvc-0f1afa8f-4b35-46f8-8109-f3bb7cd22f77  8Gi        RWO          standard    <unset>          12d
data-kafka-controller-2  Bound   pvc-f43066f3-5e15-4d9d-bf36-5ba04f02e8d5  8Gi        RWO          standard    <unset>          12d
```

6. Security with Istio

6.1 Istio Service Mesh Integration for Secure Kafka Communication

To enforce secure communication and minimize configuration overhead, **Istio** was integrated as a service mesh within the Kubernetes cluster. This enabled a **zero-trust architecture** by default and provided seamless encryption, authentication, and traffic control for all microservices—including Kafka producer, consumer, and the Kafka brokers themselves.

6.2 Mutual TLS (mTLS) for Pod-to-Pod Encryption

- **Istio's automatic mTLS** was enabled at the namespace level, ensuring that **all traffic between services was transparently encrypted** using TLS.
- Sidecar proxies (Envoy) were automatically injected into each pod and managed the encryption and decryption of traffic.
- This ensured **data confidentiality and integrity** for Kafka messages exchanged between producer, consumer, and Kafka brokers—even within the internal cluster network.

6.3 Authentication and Authorization

- **Workload identity was enforced** using Istio's strong service-to-service authentication based on **SPIFFE (Secure Production Identity Framework for Everyone)**.
- An **Istio AuthorizationPolicy** was configured to allow access only to specific default namespace (e.g., Kafka producer and consumer), blocking unauthorized traffic from other namespaces.
- Attempted communication from an unauthorized pod was **successfully denied** during testing, proving the effectiveness of the RBAC policies.

6.4 Simplified Certificate Management

With Istio, complex TLS configurations were eliminated:

- **No manual certificate generation or distribution** was required.
- **No custom TLS setup** needed in Kafka configuration files for producer or consumer.
- Certificates were **automatically provisioned, rotated, and expired** by Istio's Citadel component (or Istiod in newer versions), significantly **reducing operational overhead and human error**.

6.5 Benefits Gained

- **End-to-end secure channel for all Kafka communications**
- **Fine-grained access control** with minimal effort
- **Automatic certificate management**, saving configuration time and reducing security risks
- Enhanced observability through Istio's telemetry integration with Prometheus, Kiali, and Grafana

```
Installing Istio on k8's ,
```

```
# Download Istio
```

```
curl -L https://istio.io/downloadIstio | sh -
cd istio-*
export PATH=$PWD/bin:$PATH
```

```
# Install Istio with mTLS enabled by default
istioctl install --set profile=demo -y
```

```
To configure the istio to automatically create sidecar container deployed in the default namespace,
```

```
# To Enable automatic sidecar injection for default namespace
kubectl label namespace default istio-injection=enabled
```

```
(base) THARUN@MacBook-Pro-M2 ~ % kubectl get pods -n istio-system
NAME                               READY   STATUS    RESTARTS   AGE
grafana-65bfb5f855-khl9n          1/1     Running   7 (14h ago)  9d
istio-ingressgateway-7f7bcf6bb9-xd7v9 1/1     Running   11 (14h ago) 13d
istiod-85c68488c4-fftwm           1/1     Running   11 (14h ago) 13d
kiali-6d774d8bb8-76gxt            1/1     Running   10 (14h ago) 12d
prometheus-689cc795d4-bgp56       2/2     Running   14 (14h ago) 9d
...
```

7. Microservice Demonstration

To validate the core Kafka functionality and its integration with the distributed system infrastructure, two microservices were deployed on the Kubernetes cluster using the Bitnami Kafka image. These services were designed to simulate a producer-consumer architecture within a secure and fault-tolerant environment.

7.1 Kafka Producer Microservice

- **Image:** docker.io/bitnami/kafka:latest
- **Functionality:** Periodically sends test messages to a Kafka topic.
- **Deployment Method:** Deployed as a **Kubernetes Deployment** to ensure:
 - **Auto-healing:** In case of a pod failure, a new pod is automatically recreated by the deployment controller.
 - **Scalability:** Can be scaled horizontally to run multiple producer instances.
- **Security:** Configured to securely connect with the Kafka cluster inside the Istio service mesh using mutual TLS and sidecar injection.

```

kubectl apply -f - <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kafka-producer
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kafka-producer
  template:
    metadata:
      labels:
        app: kafka-producer
    spec:
      containers:
        - name: kafka-producer
          image: docker.io/bitnami/kafka:latest
          command:
            - "/bin/bash"
            - "-c"
            - |
              echo "Kafka producer pod is running. Exec into this pod to use kafka-console-producer.sh"
              tail -f /dev/null
  restartPolicy: Always
EOF

```

```
(base) THARUN@MacBook-Pro-M2 ~ % kubectl get deployments
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
kafka-client	2/2	2	2	14h
kafka-producer	1/1	1	1	78m

```
(base) THARUN@MacBook-Pro-M2 ~ % kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
kafka-client-5dbb5f6cc9-596gz	2/2	Running	7 (98m ago)	14h
kafka-client-5dbb5f6cc9-76k21	2/2	Running	8 (98m ago)	14h
kafka-controller-0	2/2	Running	20 (14h ago)	12d
kafka-controller-1	2/2	Running	20 (14h ago)	12d
kafka-controller-2	2/2	Running	20 (14h ago)	12d
kafka-producer-fc49cccc8-jfj7f	2/2	Running	0	78m

7.2 Kafka Consumer Microservice

- **Image:** docker.io/bitnami/kafka:latest
- **Functionality:** Subscribes to the Kafka topic and logs received messages.
- **Consumer Group:** Part of a Kafka Consumer Group to enable:
 - **Load Balancing:** Distributes message consumption across multiple instances.
 - **Parallelism:** Enables parallel processing of messages from different partitions.
- **Deployment Method:** Also deployed as a **Kubernetes Deployment**, ensuring:
 - **Auto-recovery** of the consumer pod in the event of failure.
 - **High availability** through support for multiple replicas.
- **Security:** Communicates over a secure channel within the service mesh and adheres to Istio's access control and encryption policies.

```

kubectl apply -f - <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kafka-client
  namespace: default
spec:
  replicas: 2 # Runs 2 consumer pods
  selector:
    matchLabels:
      app: kafka-client
  template:
    metadata:
      labels:
        app: kafka-client
  spec:
    containers:
      - name: kafka-client
        image: docker.io/bitnami/kafka:latest
        command:
          - "/bin/bash"
          - "--c"
          - |
            echo "Starting Kafka consumer in consumer group: my-consumer-group"
            kafka-console-consumer.sh \
              --bootstrap-server kafka.default.svc.cluster.local:9092 \
              --topic my-topic \
              --group my-consumer-group
EOF

```

(base) THARUN@MacBook-Pro-M2 ~ % kubectl get deployments

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
kafka-client	2/2	2	2	14h
kafka-producer	1/1	1	1	83m

(base) THARUN@MacBook-Pro-M2 ~ % kubectl get pods

NAME	READY	STATUS	RESTARTS	AGE
kafka-client-5dbb5f6cc9-596gz	2/2	Running	7 (103m ago)	14h
kafka-client-5dbb5f6cc9-76k21	2/2	Running	8 (103m ago)	14h
kafka-controller-0	2/2	Running	20 (14h ago)	12d
kafka-controller-1	2/2	Running	20 (14h ago)	12d
kafka-controller-2	2/2	Running	20 (14h ago)	12d
kafka-producer-fc49cccc8-jfj7f	2/2	Running	0	83m

8. Observability and Monitoring

To ensure the system is **transparent, observable, and easy to troubleshoot**, a robust observability stack was deployed. This monitoring setup covered Kafka's operations, Kubernetes resource usage, and Istio's service mesh traffic. All observability tools were installed and configured using the **Istio-provided add-ons**, ensuring seamless integration and minimal manual setup.

8.1 Prometheus

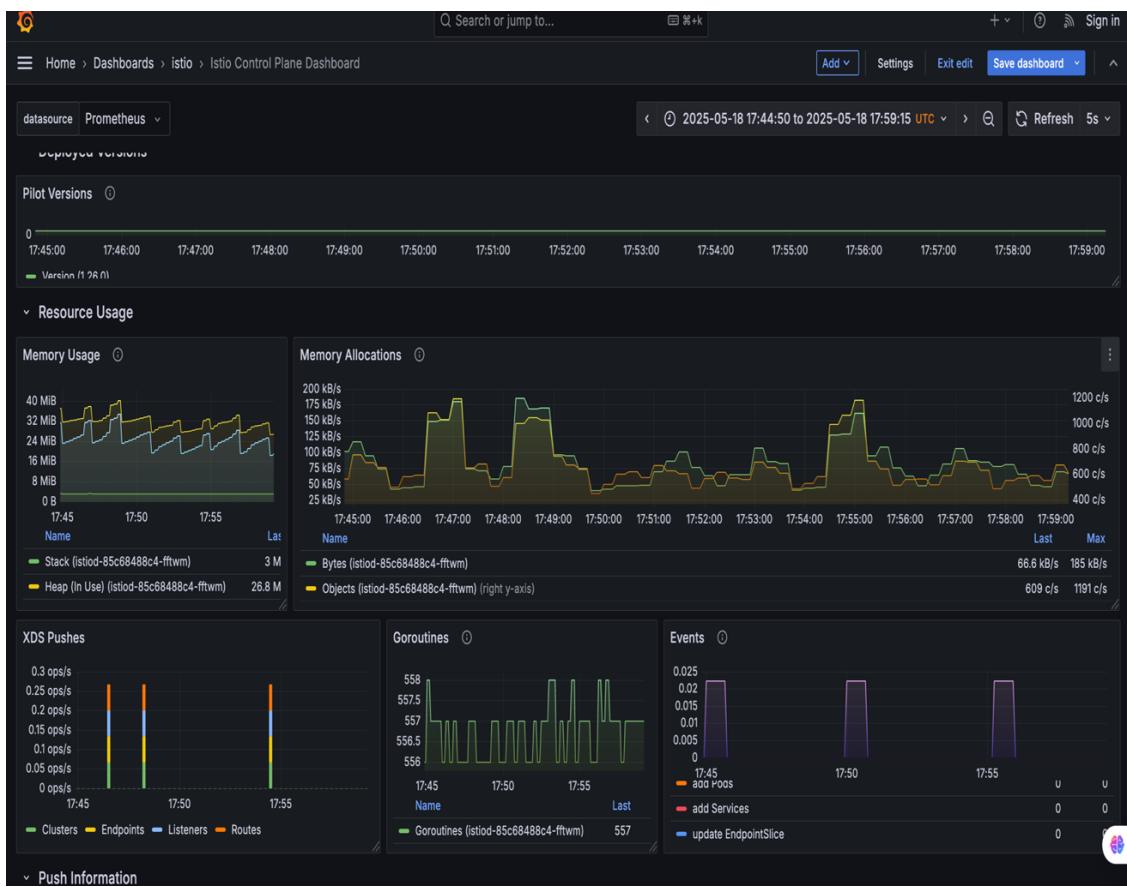
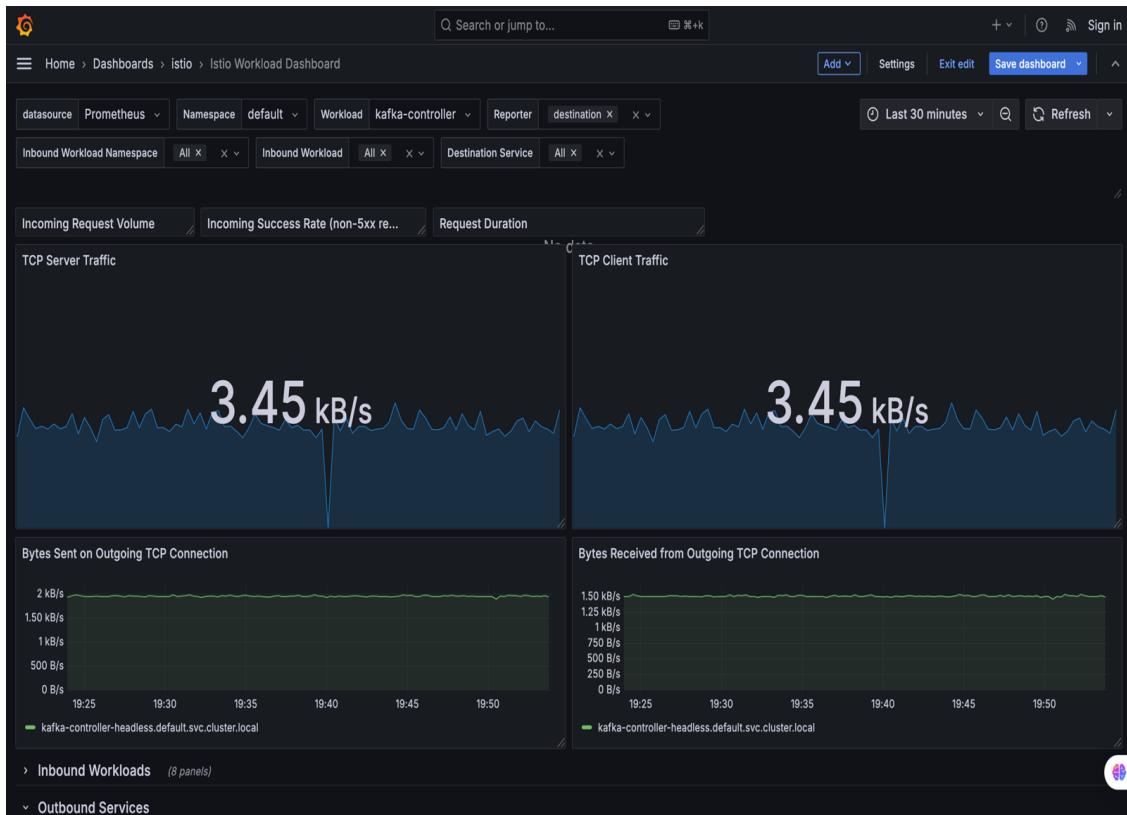
- **Installation:** Deployed from the official **Istio add-ons**, which included a pre-configured Prometheus setup tailored to Istio's metrics.
- **Functionality:**
 - **Scrapes metrics** from multiple sources, including:
 - Kafka brokers and pods (via JMX exporters or sidecar scraping)
 - Kubernetes nodes and system components
 - Istio proxies and control plane
 - **Metrics collected** included:
 - Kafka topic throughput, partition lag, and broker health
 - Pod CPU/memory utilization
 - mTLS status, request latency, error rates, and policy enforcement data

```
(base) THARUN@MacBook-Pro-M2 ~ % kubectl get pods -n istio-system
NAME                               READY   STATUS    RESTARTS   AGE
grafana-65bfb5f855-kh19n          1/1     Running   7 (15h ago)  9d
istio-ingressgateway-7f7bcf6bb9-xd7v9  1/1     Running   11 (15h ago) 13d
istiod-85c68488c4-fftwm           1/1     Running   11 (15h ago) 13d
kiali-6d774d8bb8-76gxt            1/1     Running   10 (15h ago) 12d
prometheus-689cc795d4-bgp56       2/2     Running   14 (15h ago) 9d
```

8.2 Grafana

- **Installation:** Also deployed using the **Istio bundled Helm configuration**.
- **Usage:**
 - Used to visualize real-time and historical performance metrics.
 - **Pre-configured dashboards** were customized to include:
 - **Kafka metrics:** Broker availability, topic activity, partition distribution, consumer lag
 - **Kubernetes resource dashboards:** Pod usage, node health, cluster capacity
 - **Istio dashboards:** mTLS status, success/error rates, request volumes, latency per service.
- **Benefits:**
 - Quick insights into the system's health
 - Immediate identification of bottlenecks or failures in the Kafka pipeline or service mesh

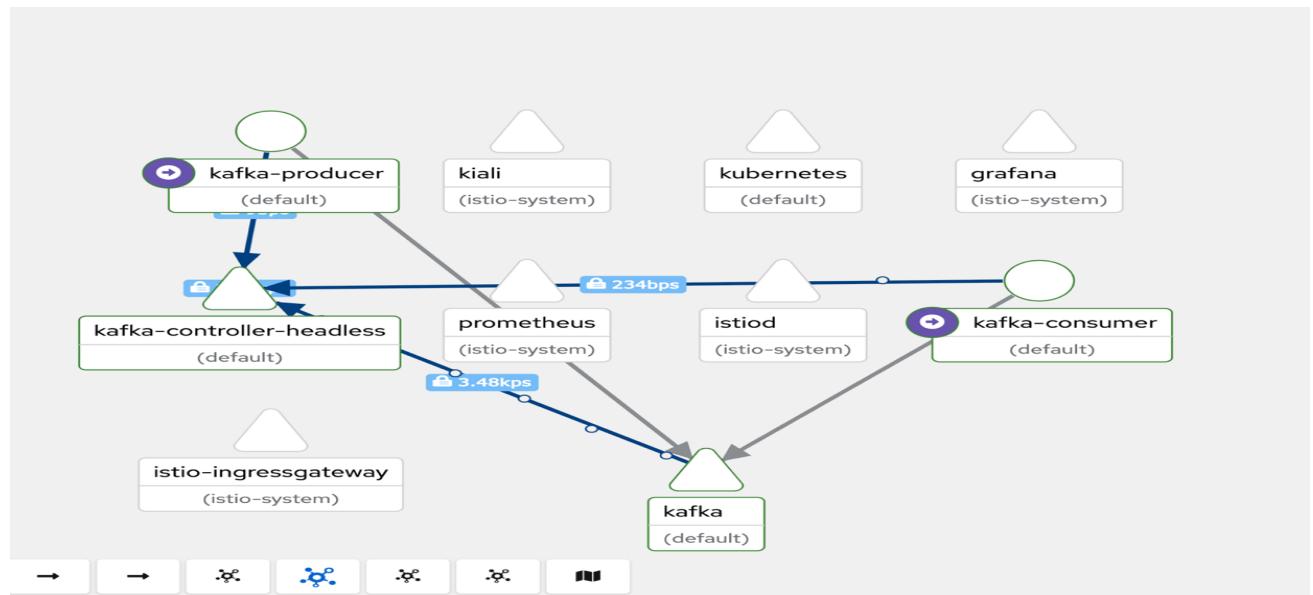
```
(base) THARUN@MacBook-Pro-M2 ~ % kubectl get pods -n istio-system
NAME                               READY   STATUS    RESTARTS   AGE
grafana-65bfb5f855-kh19n          1/1     Running   7 (15h ago)  9d
istio-ingressgateway-7f7bcf6bb9-xd7v9  1/1     Running   11 (15h ago) 13d
istiod-85c68488c4-fftwm           1/1     Running   11 (15h ago) 13d
kiali-6d774d8bb8-76gxt            1/1     Running   10 (15h ago) 12d
prometheus-689cc795d4-bgp56       2/2     Running   14 (15h ago) 9d
```

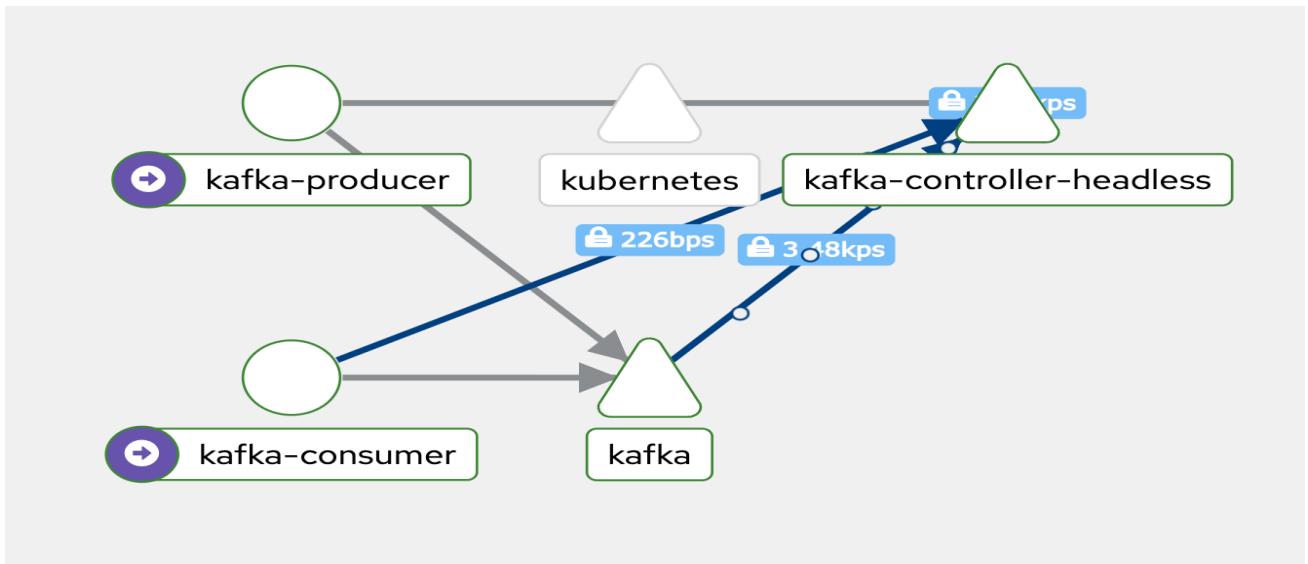


8.3 Kiali

- **Installation:** Installed using Istio's native add-on deployment (`samples/addons/kiali.yaml`), which integrates deeply with Istio's telemetry.
- **Capabilities:**
 - **Live visualization** of the **Kafka producer–broker–consumer** communication flow.
 - Highlights **mTLS status** using padlock indicators (green for encrypted communication).
 - Shows **request volume**, **error rates**, and **latency** in a service-to-service graph.
 - Displays **authorization policies**, routing decisions, and service versions if enabled.
- **Demonstration Use:**
 - Used to validate that Kafka traffic was **encrypted via mTLS**.
 - Confirmed that **unauthorized services** could not connect to Kafka producer/consumer.
 - Enabled visual tracing of Kafka message flow inside the cluster mesh.

```
(base) THARUN@MacBook-Pro-M2 ~ % kubectl get pods -n istio-system
NAME                               READY   STATUS    RESTARTS   AGE
grafana-65bfb5f855-khl9n          1/1     Running   7 (15h ago)  9d
istio-ingressgateway-7f7bcf6bb9-xd7v9 1/1     Running   11 (15h ago)  13d
istiod-85c6848c4-fftwm            1/1     Running   11 (15h ago)  13d
kiali-6d774d8bb8-76gxt           1/1     Running   10 (15h ago)  12d
prometheus-689cc795d4-bgp56       2/2     Running   14 (15h ago)  9d
(base) THARUN@MacBook-Pro-M2 ~ % kubectl get svc -n istio-system
NAME        TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)          AGE
grafana     ClusterIP 10.102.20.196 <none>        3000/TCP        9d
istio-ingressgateway   LoadBalancer 10.100.173.159 127.0.0.1   15021:30940/TCP,80:32500/TCP,443:30586/TCP  13d
istiod      ClusterIP 10.109.144.100 <none>        15010/TCP,15012/TCP,443/TCP,15014/TCP  13d
kiali       ClusterIP 10.111.167.247 <none>        20001/TCP,9090/TCP  12d
prometheus  ClusterIP 10.102.64.206 <none>        9090/TCP        9d
(base) THARUN@MacBook-Pro-M2 ~ %
```





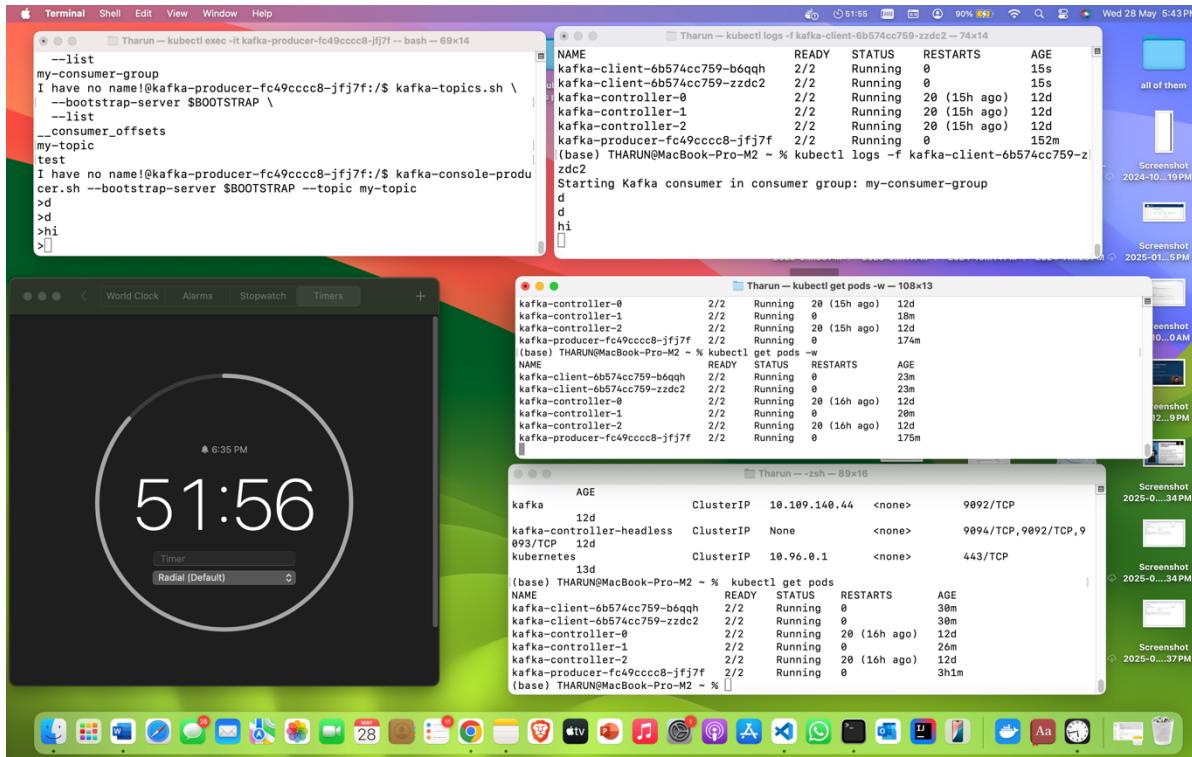
9. Demonstration of Non-Functional Properties

To validate the robustness, availability, and security of the Kafka-based microservice system, a live demonstration was conducted involving fault injection, security policy testing, and automatic certificate handling. Below are the scenarios showcased:

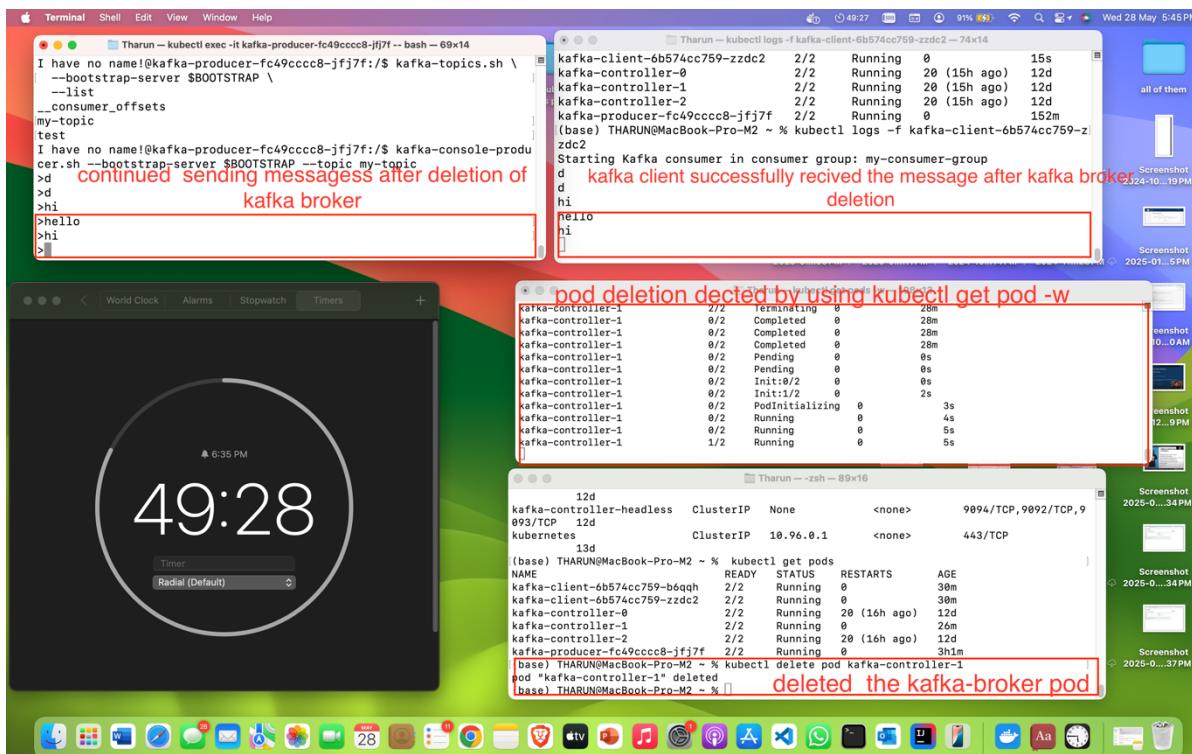
9.1 Fault Tolerance – Kafka Broker Failure

- A Kafka broker pod was **manually deleted** to simulate failure.
- Kafka's built-in replication and partitioning ensured that the cluster remained operational.
- Producers continued sending messages without errors.
- Consumers successfully received messages, proving **no message loss or downtime**.
- Kubernetes automatically respawned the deleted broker pod, validating **cluster resiliency**.

Screenshot showcasing client receiving messages from producer before the deleting the kafka pod



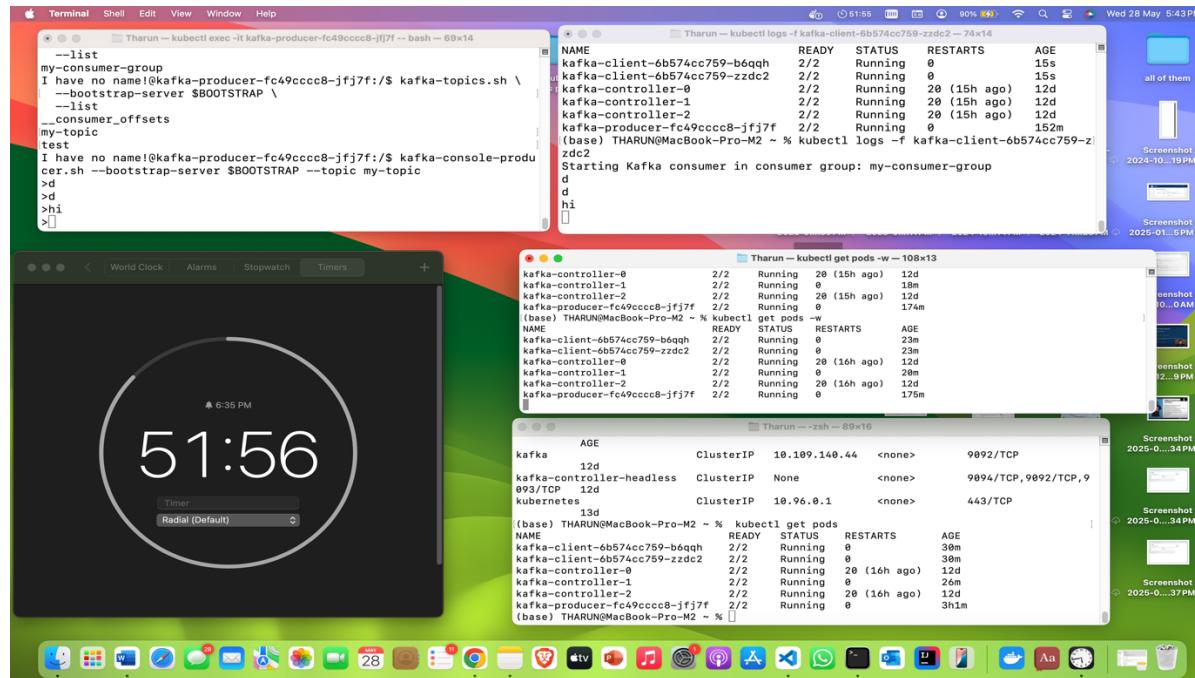
Screenshot showcasing client continue receiving messages from producer after deleting the kafka pod aswell



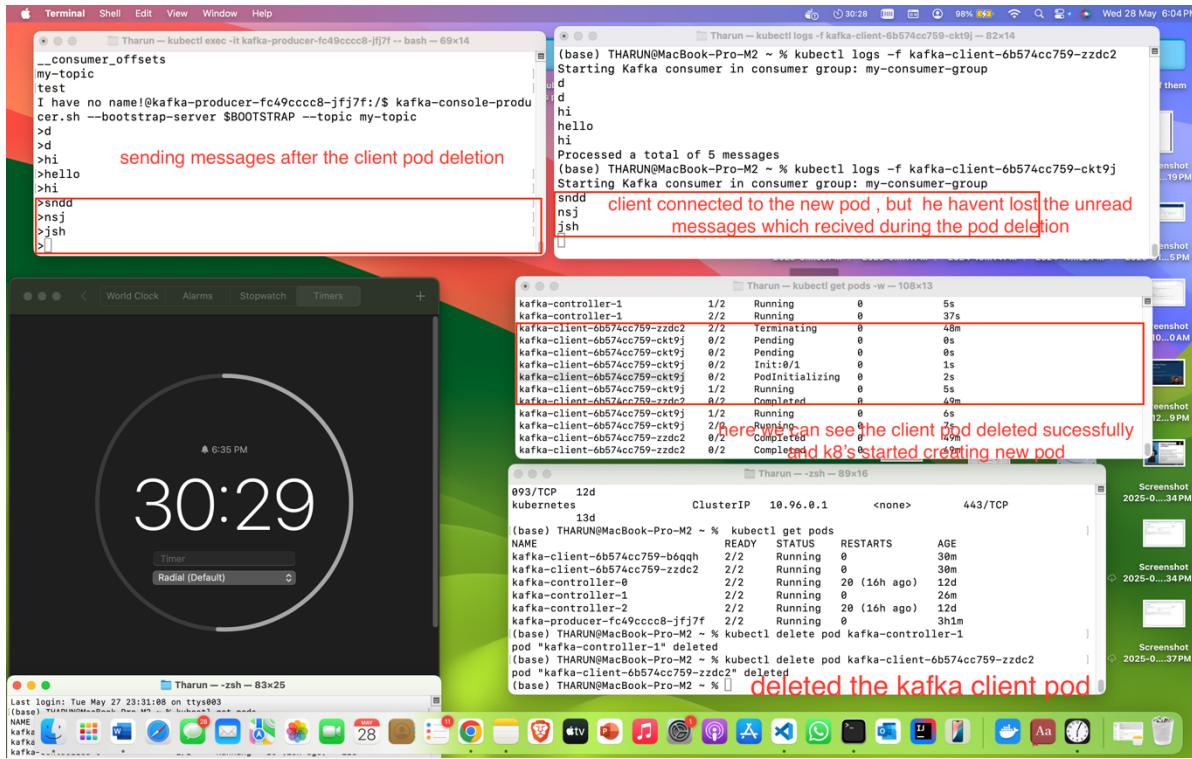
9.2 Self-Healing – Consumer Pod Crash Recovery

- A Kafka **consumer pod was deleted** intentionally.
- Kubernetes immediately recreated the pod using the existing deployment definition.
- After restarting, the new consumer resumed message consumption without data loss.
- Kafka's **durability and offset tracking** allowed the consumer to continue from where it left off.

Screenshot before client pod deletion



Screenshot after client pod deletion:



9.3 Security Enforcement – Unauthorized Access Prevention

- A rogue pod named `attacker` was deployed to test unauthorized access attempts.
- An **Istio AuthorizationPolicy** was configured to allow only authenticated service accounts to interact with the producer.
- When the attacker pod attempted to access the producer service, the connection was denied (403 error), confirming that **Istio RBAC and mTLS policies** were effectively enforced.

Screenshot of this demonstration launching the attacker pod and trying to access the Kafka-broker

The screenshot displays a macOS desktop environment with two terminal windows open in a 'Terminal' application.

Top Terminal Window:

```

(base) THARUN@MacBook-Pro-M2 ~ % kubectl create namespace attacker
namespace/attacker created
(base) THARUN@MacBook-Pro-M2 ~ % kubectl run attacker \
--restart='Never' \
--image=docker.io/bitnami/kafka:latest \
--namespace=attacker \
--command -- sleep infinity
pod/attacker created
(base) THARUN@MacBook-Pro-M2 ~ % kubectl get pods -n attacker
NAME      READY   STATUS    RESTARTS   AGE
attacker   1/1     Running   0          117s
(base) THARUN@MacBook-Pro-M2 ~ %

```

Bottom Terminal Window:

```

(base) THARUN@MacBook-Pro-M2 ~ % kubectl exec -it attacker -- bash
Error from server (NotFound): pods "attacker" not found
(base) THARUN@MacBook-Pro-M2 ~ % kubectl exec -it attacker -- bash
I have no name!@attacker:/# kafka-topics.sh \
--bootstrap-server kafka.default.svc.cluster.local:9092 \
--list
Error while executing topic command : Timed out waiting for a node assignment. Call: listTopics
[2025-05-28 16:38:41,728] ERROR org.apache.kafka.common.errors.TimeoutException: Timed out waiting for a node assignment. Call: listTopics
(org.apache.kafka.tools.TopicCommand)
I have no name!@attacker:/#

```

Annotations:

- A red box highlights the command and output in the top terminal window: "successfully deployed attacker pod in the new attacker namespace where istio is not configured for".
- A red box highlights the error message in the bottom terminal window: "tried to login into attacker pod and attempted to connect to the Kafka broker here and again see the connection faild ,where istio blocked the access".

10. Conclusion

The project demonstrated how Kafka, when deployed on Kubernetes with proper architectural choices, can meet the rigorous non-functional requirements of modern distributed systems. The integration with **Istio** simplified security enforcement, and the use of **Raft protocol**, **PVCs**, and **Helm** ensured system robustness, availability, and resilience.

This solution offers a **production-ready Kafka microservices architecture** capable of secure communication, real-time data streaming, and resilient behavior under adverse conditions.