Deploying Kafka Using Strimzi

*Introduction*

**Overview**: Strimzi is an open-source project designed to manage Apache Kafka on Kubernetes and OpenShift clusters, making it easier for organizations to deploy, operate, and scale Kafka in cloud-native environments. It simplifies the process of deploying Kafka brokers, Zookeeper clusters, Kraft clusters, and Kafka Connectors, automating critical tasks such as scaling, rolling updates, and monitoring. Strimzi integrates with Kubernetes-native tools like Helm and Operators, providing a robust solution for event-driven architectures. Competitors to Strimzi include Confluent (which offers a managed Kafka service and its own Kubernetes operator), and Amazon MSK, which also aim to simplify Kafka deployment and management on containerized platforms.

*Core Features*

**Kubernetes Integration:** Strimzi is designed to work seamlessly with Kubernetes and OpenShift, taking full advantage of these container orchestration platforms. It uses Kubernetes-native constructs such as Operators, Custom Resource Definitions (CRDs), and StatefulSets to manage Kafka deployments in a consistent and declarative way.

**Strimzi Operator:** The Strimzi Kafka Operator is the central component of the Strimzi ecosystem. It automates the management of Kafka clusters by monitoring the state of the resources and making adjustments as needed. This includes handling Kafka broker provisioning, configuration, monitoring, and scaling.

**KRaft Metadata Management:** "Kraft controllers" refer to the nodes within a Kafka cluster that manage cluster metadata and facilitate leader elections, essentially acting as the control plane for the Kafka cluster, all while utilizing the KRaft (Kafka Raft) consensus protocol, which allows the Kafka cluster to operate without relying on an external ZooKeeper service for metadata management.

**Multi-Cluster and Multi-Tenant Support:** Strimzi supports both multi-cluster and multi-tenant configurations, making it suitable for organizations with complex deployment needs. With Strimzi, operators can set up Kafka clusters that span multiple Kubernetes clusters, and the operator can manage the state of Kafka across all clusters.

**Auto-Scaling and Rolling Updates:** Strimzi supports automatic scaling of Kafka brokers and other components, ensuring that resources can be adjusted dynamically as workloads change. It also enables rolling updates, ensuring that the system remains operational while updates are being applied.

**Kafka Connect Integration:** Strimzi simplifies the deployment of Kafka Connect clusters for integrating external data sources and sinks with Kafka. Kafka Connect is a framework for connecting Kafka with other systems like databases, file systems, and cloud services, and Strimzi makes it easier to deploy and manage these connectors in Kubernetes environments.

**Security and Encryption:** Strimzi supports multiple security features, such as TLS encryption, authentication, and authorization for Kafka brokers. It integrates with Kubernetes secrets management to handle sensitive information securely, ensuring that Kafka clusters are protected in production environments.

**Monitoring and Logging:** Strimzi integrates with common monitoring tools such as Prometheus and Grafana, allowing operators to monitor Kafka clusters and visualize key performance metrics. It also supports centralized logging solutions, making it easier to track and troubleshoot issues in Kafka deployments.

*Advantages*

**Cost-Effective**: Since Strimzi is open-source and free to use, it offers a cost-effective way to deploy and manage Kafka without needing a commercial license, unlike Confluent's offerings.

**Kubernetes-First**: Strimzi is a natural fit because it leverages Kubernetes' native capabilities for scaling, self-healing, and managing resources. This reduces operational overhead and aligns with modern DevOps practices.

**Highly Extensible**: Being open-source, Strimzi allows for easy customization, integration with other Kubernetes-native tools, and contributions from the community. You can extend its functionality to fit specific needs.

**Unified Management with Kubernetes CRDs**: Strimzi uses Kubernetes Custom Resource Definitions (CRDs) for managing Kafka resources, which means it integrates well with Kubernetes' declarative management model. This allows Kubernetes operators to manage Kafka just like other Kubernetes resources.

**Strong Community Support**: Strimzi has an active open-source community that contributes to the project. There are multiple resources available for troubleshooting, extending functionality, and best practices, making it a flexible choice.

**Multi-Cloud Compatibility**: Strimzi is cloud-agnostic, meaning you can deploy Kafka on any Kubernetes cluster regardless of whether it's on AWS, Azure, Google Cloud, or on-premises.

*Disadvantages*

**Requires Kubernetes Expertise**: There is a steep learning curve if Kubernetes information

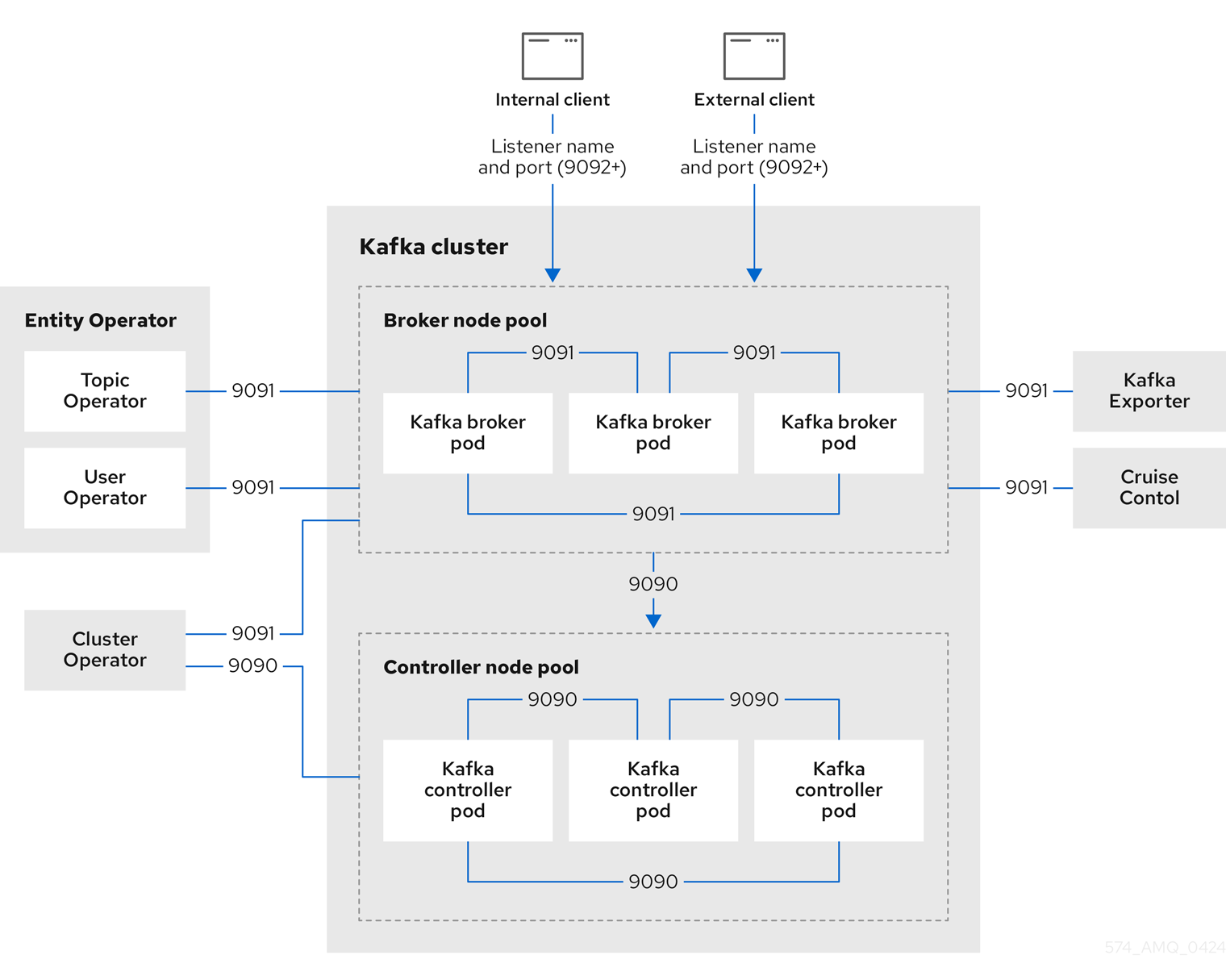
**Less Out-of-the-Box Features**: While Strimzi is feature-rich, it doesn’t provide as many out-of-the-box enterprise features like **Confluent Platform**. For example, Confluent offers more advanced security features, robust monitoring, and schema registry services right out of the box, which require extra setup in Strimzi (including an additional operator for registry services).

**Limited Enterprise Support**: Strimzi does not come with commercial support unless you leverage third-party managed services like Red Hat OpenShift or other managed Kafka providers. This is a major contrast to **Confluent**, which offers enterprise-grade support with SLAs.

**Limited Tools for Kafka Operations**: Strimzi does not provide as extensive a set of management tools as **Confluent**, which has built-in solutions for managing Kafka connectors, stream processing, and other tasks.

**Fewer Pre-Built Connectors**: While Strimzi supports Kafka Connect, the number of pre-built connectors in the open-source version is less than what Confluent offers through its commercial offerings.

*Architecture*



*Sample Deployment*

To deploy a basic Strimzi-based Kafka Cluster, you will need the following pre-requisites in place:

* Kubernetes cluster created
* Necessary permissions/access to be able to provision resources in Kubernetes
* Sample YAML files downloaded for desired Strimzi release (<https://github.com/strimzi/strimzi-kafka-operator/releases/>)

Once you have the pre-requisites in place, you can use kubectl to apply the various YAML files to install relevant Kafka components.

Firstly, Create your namespace within the Kubernetes cluster.

* kubectl create namespace strimzi-kafka

Next, install the Strimzi operator and resources in your newly created namespace.

* kubectl apply -f <https://strimzi.io/install/latest?namespace=strimzi-kafka>

Finally, apply your yaml file which will install all necessary kafka components in your newly created namespace using the strimzi operator.

* kubectl apply -f strimzi-kafka.yaml -n strimzi-kafka

A sample YAML file is provided below which deploys 3 KRaft controllers as well as 3 kafka brokers and 2 entity operators for topic/user management:



Additional YAML files provided below which provide a sample set of topics and a sample user with access to these topics:



### **Kafka Cluster Data Protection**

Strimzi helps users implement reliable backup and restore processes for Kafka clusters to ensure data integrity and availability. It leverages Kubernetes-native tools and concepts to provide a seamless way to back up Kafka data, such as topics, partitions, and consumer offsets.

### **Backup/Restore with Kafka Connect**

Strimzi supports Kafka Connect for integrating backup and restore functionality with external systems. Kafka Connect can be used to export data to various storage solutions like Amazon S3, Google Cloud Storage, or distributed file systems.

* **Backup**: You can set up connectors to periodically export Kafka data, including topics and their content, to external systems.
* **Restore**: In case of a failure, Strimzi allows you to restore data back into your Kafka cluster from these external backup locations.

### **Automated Cluster Backup/Restore**

* Strimzi’s Kafka operator can automate the deployment of backup and restore configurations. This means that users can define backup schedules and manage the entire process via Kubernetes manifests.
* The backup process can be configured to ensure consistency and minimal disruption to the running Kafka services.

### **Snapshot-based Backups**

* Strimzi enables the use of **snapshot-based backups**, which are taken at a specific point in time, ensuring consistency across Kafka's data stores. This is particularly useful when working with **stateful sets** in Kubernetes where backup consistency must be ensured across the entire system. This is done via the CRD KafkaConnect.

### **Disaster Recovery and High Availability**

* With the backup and restore features, Strimzi helps implement disaster recovery (DR) strategies by ensuring that data can be quickly restored in case of failure through use of Kafka MirrorMaker Dr capability.
* It provides options for active standby kafka cluster configurations that span geographical regions.

### **Restore Granularity**

* You can restore entire Kafka clusters or specific topics, partitions, or offsets. This allows users to tailor the restore process based on the need (e.g., restoring only a specific topic to a previous state).

### **Consistency with Multiple Kafka Brokers**

* Strimzi supports consistent backup and restore even across multi-broker Kafka clusters. This is important because Kafka clusters often run in a distributed manner, and ensuring that backups are consistent across brokers is critical for recovery.

### **Backup Configuration Flexibility**

* Strimzi allows backup configurations to be highly flexible. This includes the ability to adjust backup frequencies, data retention policies, and integrate with custom backup strategies based on organization needs (e.g., storing backups on-premises vs. in the cloud).

### **Custom Backup Solutions Integration**

* Strimzi can integrate with custom backup solutions or third-party tools like **Velero**, which is widely used for Kubernetes backups. This integration gives users more flexibility and control over their backup strategies.

### **Cross-Cluster Backup/Restore**

* For multi-cluster Kafka environments, Strimzi can facilitate cross-cluster backup and restore. It allows backing up Kafka clusters in one Kubernetes environment and restoring them in another, which can be important for multi-region deployments or for creating replica clusters.

### **Simplified Restore with Kafka Topic Replication**

* Kafka’s built-in replication features help in ensuring that the data is available across different brokers, and if a failure occurs, Strimzi can restore the data with minimal downtime by leveraging topic replication and offsets management.

### **Integration with Monitoring and Alerts**

* Strimzi provides monitoring features (through Prometheus, Grafana) that track the backup/restore health. Alerts can be configured to notify users of any issues during backup or restore processes, allowing for proactive measures to be taken.

*Storage Types*

In Strimzi, you can configure various **storage options** for Kafka brokers. These options are defined in the **Kafka custom resource** (CR) manifest, allowing you to specify how Kafka data is stored. The storage configuration impacts the performance, durability, and scalability of the Kafka cluster.

Strimzi offers multiple storage types, which are selected based on your infrastructure and requirements. These options are defined under the spec.kafka.storage section of the Kafka CR manifest.

### 1. **Ephemeral Storage** (ephemeral)

Ephemeral storage is used for temporary, non-persistent storage. This storage type is appropriate for use cases where the Kafka broker does not need to retain data across pod restarts or failures.

* **Characteristics**:
  + Data is lost when the Kafka pod is terminated or restarted.
  + Typically used in development environments or for testing purposes.
  + No persistent volume (PV) is created.
  + Suitable for non-production environments where data durability is not a concern.
* **Example Manifest**:

yaml

Copy code

spec:

kafka:

storage:

type: ephemeral

### 2. **Persistent Storage** (persistent-claim)

Persistent storage is used when you need data to persist across pod restarts and crashes. This option uses Kubernetes Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) to manage the storage lifecycle.

* **Characteristics**:
  + Data is retained even if the broker pod restarts or fails.
  + Requires a dynamic or static volume provisioner (like NFS, Ceph, or cloud storage providers such as AWS EBS, GCP Persistent Disks).
  + This is the most common choice for production environments, where durability and high availability are essential.
* **Storage Options**:
  + **Size**: You can specify the storage size.
  + **StorageClass**: Allows specifying a specific StorageClass for the PVC.
  + **VolumeMode**: Set to Filesystem (default) or Block for block storage.
* **Example Manifest**:

yaml

Copy code

spec:

kafka:

storage:

type: persistent-claim

size: 100Gi

class: standard

volumeMode: Filesystem

### 3. **Jbod Storage** (jbod)

JBOD (Just a Bunch Of Disks) allows you to configure multiple disks (volumes) for each Kafka broker. This option is suitable when you need to scale storage across multiple volumes, increasing the storage capacity or isolating data (e.g., separating logs and data on different disks).

* **Characteristics**:
  + Provides fine-grained control over storage configuration, allowing you to specify multiple disks (e.g., disk1, disk2).
  + Each disk can have different sizes or storage classes.
  + Useful for large-scale Kafka clusters, where you want to distribute data across multiple disks for performance reasons.
* **Example Manifest**:

yaml

Copy code

spec:

kafka:

storage:

type: jbod

volumes:

- id: 0

type: persistent-claim

size: 100Gi

- id: 1

type: persistent-claim

size: 100Gi

### 4. **Custom Storage** (custom)

The custom storage option allows you to specify a custom storage configuration beyond the defaults, typically for more advanced scenarios. This is useful when you want more control over the underlying storage, like using specific mount paths, using custom provisioning logic, or integrating with third-party storage solutions.

* **Characteristics**:
  + Allows fine-grained customization, including mounting external volumes or integrating custom storage backends.
  + More flexibility and complexity compared to the other storage types.
* **Example Manifest**:

yaml

Copy code

spec:

kafka:

storage:

type: custom

volumes:

- name: custom-volume

mountPath: /var/lib/kafka

persistentClaim:

claimName: my-pvc

### Key Considerations for Storage Configuration

1. **Performance**: Persistent storage types can affect Kafka performance depending on the type of backend storage used. Consider high-performance disk types for production deployments.
2. **Resilience**: Choosing persistent storage (especially with multiple disks in a JBOD configuration) improves data durability and ensures Kafka brokers remain resilient in case of node failure.
3. **Cloud-Native**: If you're using a cloud-based Kubernetes environment (like AWS, GCP, or Azure), it's common to use dynamic provisioning with cloud-specific persistent storage (like AWS EBS, GCP Persistent Disk, etc.).
4. **Size and Scalability**: Ensure that the storage size is appropriate for your Kafka workload, considering factors like retention periods and message volume.

*Role-Based Access*

In Strimzi, Role-Based Access Control (RBAC) permissions are essential to manage and secure access to the resources associated with Kafka clusters, Kafka Connect clusters, and other Strimzi components. These permissions ensure that the right users and services have appropriate access to Kafka resources within a Kubernetes environment. RBAC policies are defined in Kubernetes manifests and control access to resources like brokers, topics, consumers, producers, and more.

1. **Service Accounts**

* Service accounts are used to provide identity to the components (like Kafka brokers, Kafka Connect, etc.) within the Kubernetes cluster. These identities are associated with specific roles and permissions through RBAC.
* You can define a custom **service account** for Kafka components, or use the default ones.

**Example:**

yaml

Copy code

spec:

kafka:

authorization:

superUsers:

- "CN=admin,O=example"

1. **Roles and RoleBindings**

Strimzi uses Kubernetes Role-based Access Control (RBAC) to define permissions. The two most commonly used RBAC resources in the context of Strimzi are:

* **Role**: Defines a set of permissions (e.g., access to Kafka resources).
* **RoleBinding**: Associates a Role with a specific user or service account.

Strimzi provides a default **KafkaRole** and **KafkaRoleBinding** for managing access at the cluster and topic levels. You can customize these to fit your use case.

**Example:**

yaml

Copy code

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

namespace: my-kafka-namespace

name: kafka-producer-role

rules:

- apiGroups: ["strimzi.io"]

resources: ["kafkas"]

verbs: ["get", "list", "create"]

yaml

Copy code

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: kafka-producer-role-binding

namespace: my-kafka-namespace

subjects:

- kind: ServiceAccount

name: kafka-producer-sa

namespace: my-kafka-namespace

roleRef:

kind: Role

name: kafka-producer-role

apiGroup: rbac.authorization.k8s.io

1. **Kafka Authorization Configuration**

Strimzi provides an **authorization** section where you can specify the level of access that users and service accounts have over Kafka resources, such as topics, consumer groups, and brokers. This is commonly used to configure access control at the Kafka broker level. The three main types of authorization supported by Strimzi are:

* **Simple (ACL-based)**: Uses Kafka’s internal Access Control Lists (ACLs) to define which users or clients can perform certain operations on Kafka resources (e.g., produce, consume, create topics).
* **RBAC for Kubernetes-native resources**: Kubernetes RBAC is used to control access to the Kafka cluster itself (e.g., create/delete topics, describe Kafka resources).
* **Super Users**: This is a list of Kafka users that have unrestricted access to all Kafka resources.

**Example (ACL-based authorization for Kafka):**

yaml

Copy code

spec:

kafka:

authorization:

type: kafka

superUsers:

- "CN=kafka-admin,O=example"

acls:

- principal: "User:kafka-producer"

operation: "Read"

resource:

type: "Topic"

name: "my-topic"

- principal: "User:kafka-consumer"

operation: "Write"

resource:

type: "Topic"

name: "my-topic"

1. **Resource Access Control (KafkaTopic/Producer/Consumer)**

Strimzi allows fine-grained control over permissions at the **topic** and **producer/consumer** levels. You can specify which users are allowed to produce or consume messages from specific topics using the acls section in the Kafka CR.

The following permissions can be specified for Kafka topics:

* **Read**: Allow consuming messages from a topic.
* **Write**: Allow producing messages to a topic.
* **Create**: Allow creating topics.
* **Delete**: Allow deleting topics.
* **Alter**: Allow altering topic configurations.

**Example:**

yaml

Copy code

spec:

kafka:

authorization:

type: kafka

acls:

- principal: "User:kafka-producer"

operation: "Write"

resource:

type: "Topic"

name: "my-topic"

1. **Kafka Connect Authorization**

Strimzi allows you to configure Kafka Connect components with specific access to resources like Kafka brokers, connectors, and their configurations.

* You can configure a service account for Kafka Connect, and associate appropriate roles with it to control access.
* Kafka Connect configurations (like connectors) can also be restricted to specific users or service accounts.

**Example:**

yaml

Copy code

spec:

kafkaConnect:

authorization:

type: simple

superUsers:

- "CN=kafka-connect-admin,O=example"

1. **Custom Role Configurations**

You can also define custom roles in Strimzi using Kubernetes RBAC objects such as Role and ClusterRole. This allows fine-grained control over specific resources, especially when managing resources beyond Kafka brokers (e.g., Kafka Connect, Kafka MirrorMaker, etc.).

**Example:**

yaml

Copy code

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

namespace: my-kafka-namespace

name: kafka-cluster-admin

rules:

- apiGroups: ["strimzi.io"]

resources: ["kafkas"]

verbs: ["get", "list", "create", "update", "delete"]

1. **RBAC for Kafka CRDs (Custom Resources)**

Strimzi custom resources (CRs) such as Kafka, KafkaConnect, and KafkaTopic can be controlled via Kubernetes RBAC as well. You can configure access to these CRs by associating them with specific roles and role bindings.

**Example:**

yaml

Copy code

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

name: kafka-topic-reader

namespace: my-kafka-namespace

rules:

- apiGroups: ["strimzi.io"]

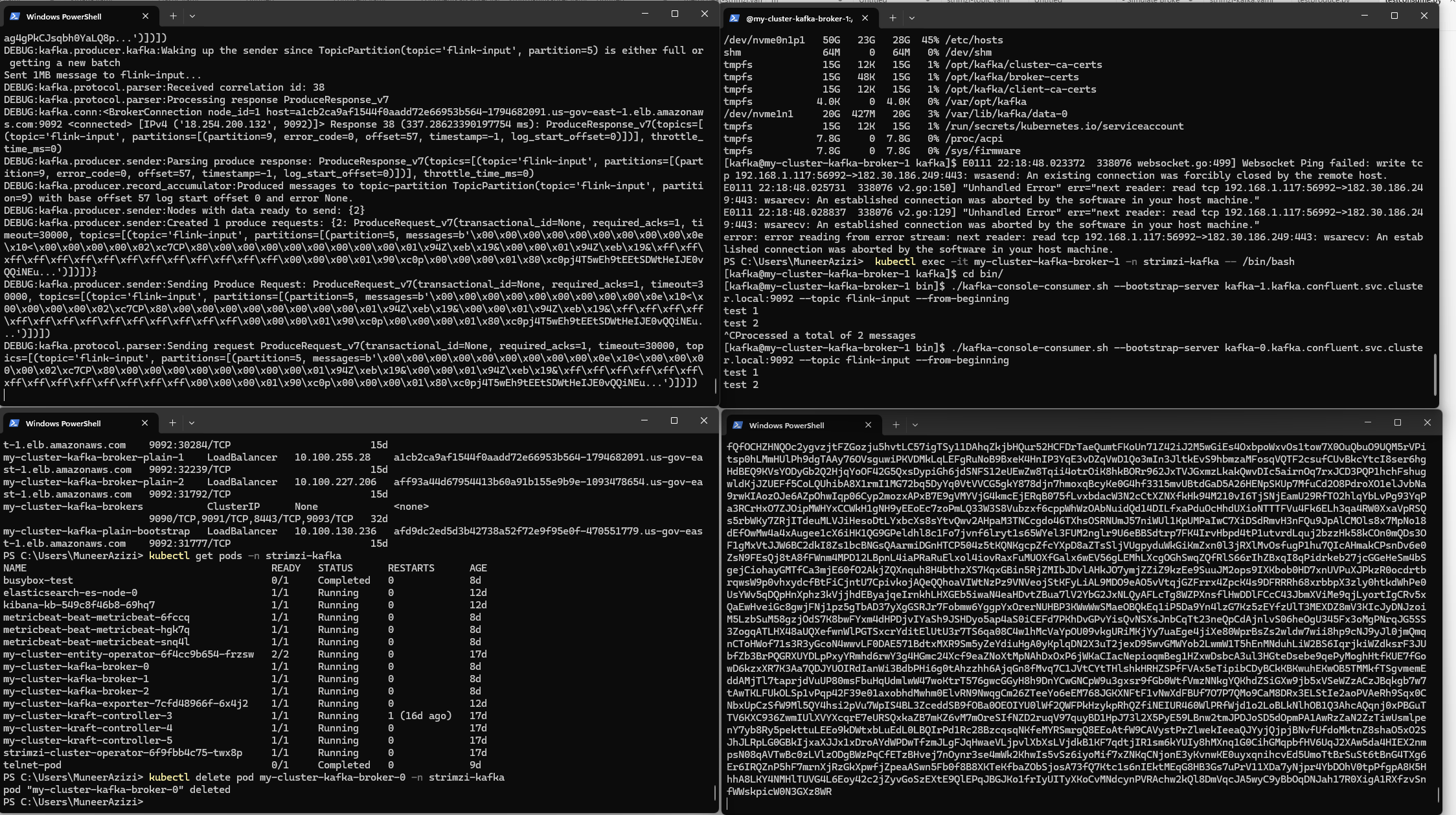
resources: ["kafkatopics"]

verbs: ["get", "list"]

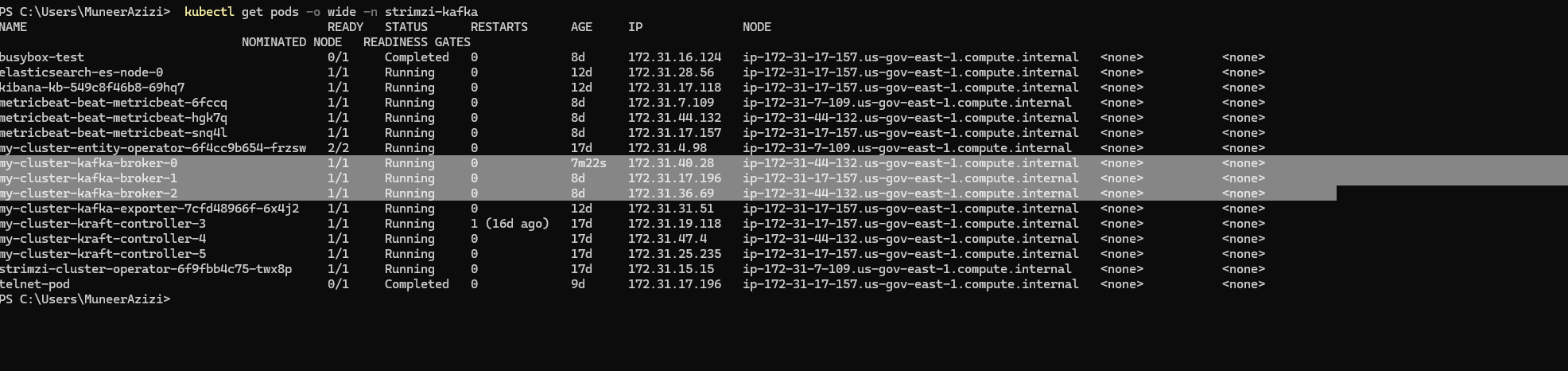
*Test Scenarios*

While deploying Strimzi Kafka-clusters in the RDT&E EKS stack several test scenarios were covered.

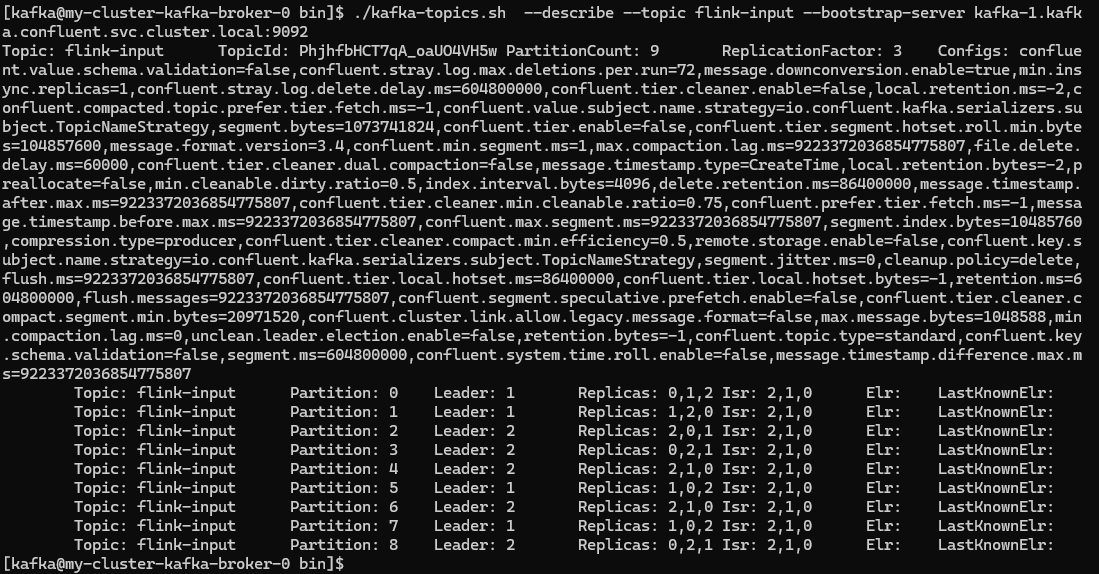
**Broker Failure Simulation**



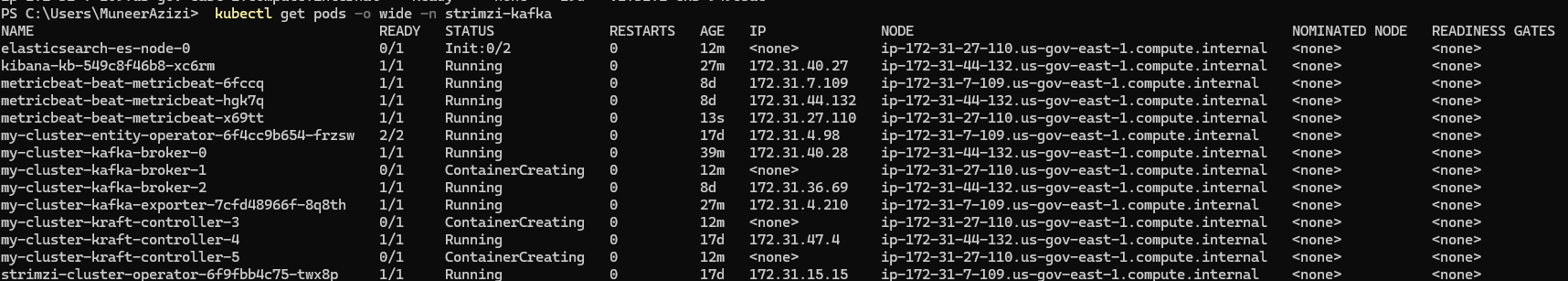
Strimzi Operator auto restarts the pod quickly (within seconds) when a single broker pod is manually deleted. Does not impact topic test production as well as topic test consumption.



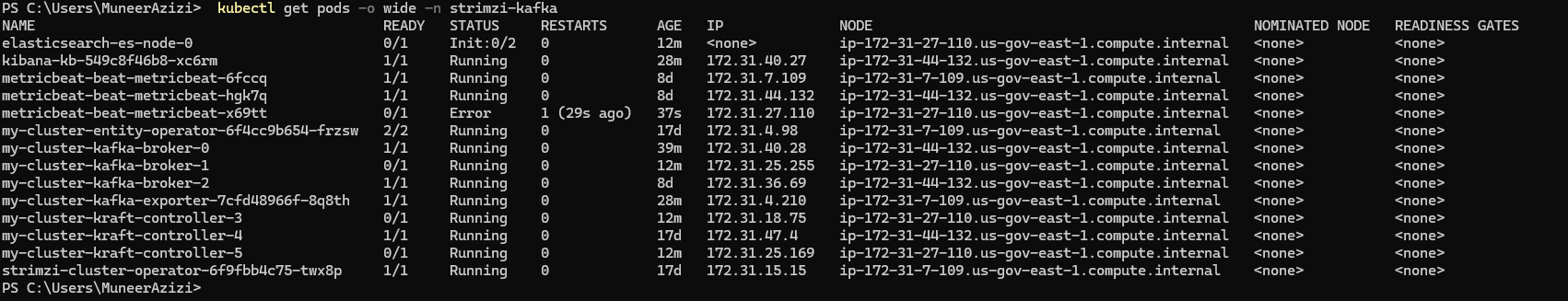
Shutting down the EC2 instance while writing to a topic and consuming from the topic resulted in about a ~20 second delay in reading new messages from the topic as messages began queuing from the producer end due to migration/rebalance of partitions from broker-1 being unavailable. All partitions are now spread across the 2 remaining kafka brokers.



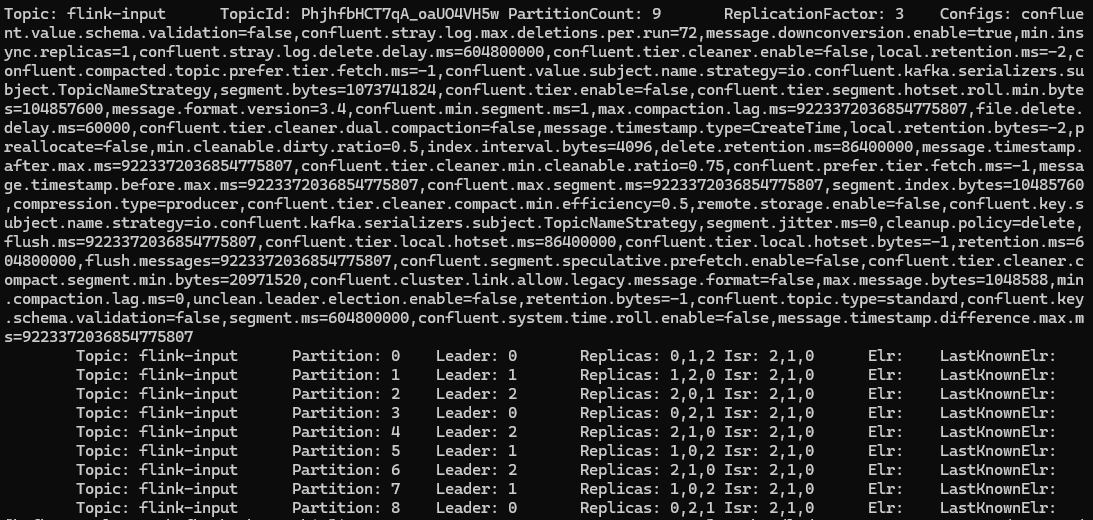
After a few minutes Kubernetes terminated the pods on the stopped EC2 instance and began deploying new pods on the newly provisioned EC2 instance (from auto-scaling rules defined in AWS). Kafka connectivity is still available during this entire time even while broker-1 is in a terminating state.



New Kafka-related pods are now running on new EC2 instance provisioned from auto-scaling group.



Topic partitions also auto-rebalanced across all 3 nodes now spreading partitions to the newly redeployed pod running on the new EC2 instance.



**Velero Namespace Backup for Strimzi Operator**

The namespace backup for the Strimzi operator using velero was relatively quick due to the size of the environment, and restoration is as simple as executing a single command:

velero restore create --from-backup <backup-name> --namespace <namespace>

You can monitor restoration progress as follows

velero restore describe <restore-name> --details

**Strimzi Schema Registry**

Strimzi uses confluents schema registry, which begins with installing the registry operator

helm repo add lsstsqre https://lsst-sqre.github.io/charts/

helm repo update

helm install lsstsqre/strimzi-registry-operator --name ssr --set clusterNamespace="...",clusterName="..."

Next, deploy the topic the registry will use for primary storage

apiVersion: kafka.strimzi.io/v1beta2

kind: KafkaTopic

metadata:

name: registry-schemas

labels:

strimzi.io/cluster: events

spec:

partitions: 1

replicas: 3

config:

# http://kafka.apache.org/documentation/#topicconfigs

cleanup.policy: compact

Next, deploy a kafka user for the registry

apiVersion: kafka.strimzi.io/v1beta2

kind: KafkaUser

metadata:

name: confluent-schema-registry

labels:

strimzi.io/cluster: events

spec:

authentication:

type: tls

authorization:

# Official docs on authorizations required for the Schema Registry:

# https://docs.confluent.io/current/schema-registry/security/index.html#authorizing-access-to-the-schemas-topic

type: simple

acls:

# Allow all operations on the registry-schemas topic

# Read, Write, and DescribeConfigs are known to be required

- resource:

type: topic

name: registry-schemas

patternType: literal

operation: All

type: allow

# Allow all operations on the schema-registry\* group

- resource:

type: group

name: schema-registry

patternType: prefix

operation: All

type: allow

# Allow Describe on the \_\_consumer\_offsets topic

- resource:

type: topic

name: \_\_consumer\_offsets

patternType: literal

operation: Describe

type: allow

Finally, deploy the registry itself

apiVersion: roundtable.lsst.codes/v1beta1

kind: StrimziSchemaRegistry

metadata:

name: confluent-schema-registry

spec:

strimziVersion: v1beta2

listener: tls

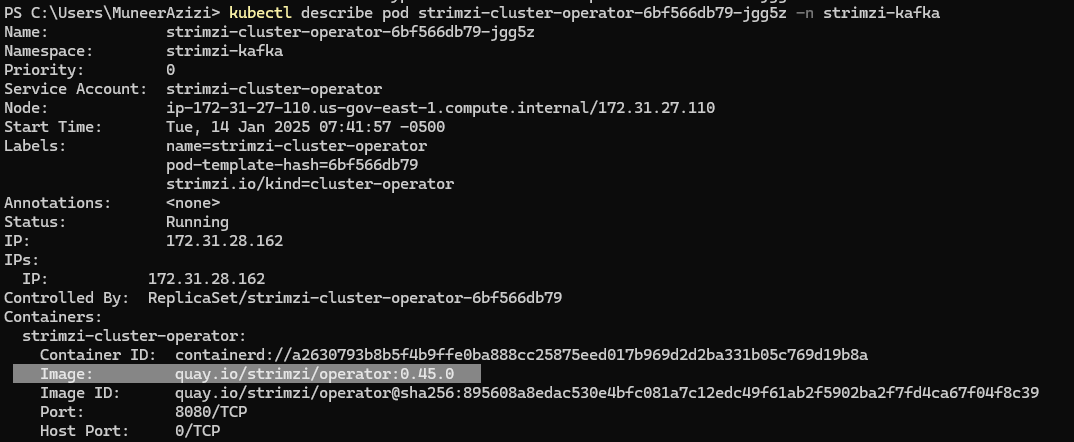
**Rolling Upgrade**

Upgrades to Kafka clusters using the Strimzi Operator are straight-forward. Simply upgrade the operator first, then modify the kafka version in your main kafka deployment manifest to the new kafka version you desire and the newly upgraded operator will handle the rolling upgrade of each kafka node.

Same command as ran above (new version is now available),

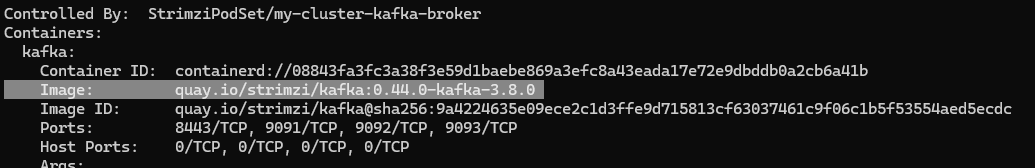
kubectl apply -f <https://strimzi.io/install/latest?namespace=strimzi-kafka>

Check new operator pod and validate new version

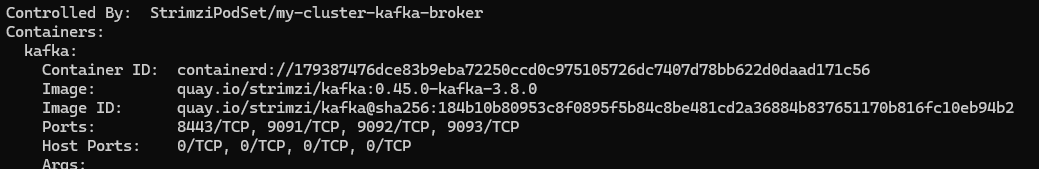


This triggers a rolling restart of all kafka nodes to use the same kafka version image, but new operator version with it.

For example:

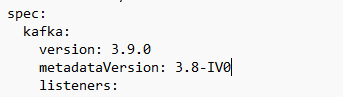


Now shows:

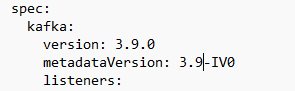


Same kafka version, but different operator version now.

Next, simply change the version of kafka your deployment manifest and reapply it



Once the rolling update of Kafka cluster is complete, update your deployment manifest once again to update the KRaft clusters metadata version.



Then reapply once again, and wait for KRaft pods to get upgraded.