**Introduction**

Confluent for Kubernetes (CFK) is a Kubernetes-native platform designed to deploy, manage, and scale Confluent services, including Kafka and Confluent Connectors, within a Kubernetes environment. CFK simplifies the management of these services through Kubernetes-native constructs, providing features like scalability, resilience, and seamless integration with other cloud-native tools and technologies.

One challenge to using Confluent for Kubernetes is a need for topic backups. Without deploying Kafka Mirror Maker, in order to leverage topic-level backups one must deploy Kafka Connector to manage the transition of data in-and-out of the Kafka cluster.

This document will provide an overview of Confluent for Kubernetes Declarative Connectors, including their benefits, key features, how they work, and how to implement and manage them within a Kubernetes ecosystem. This document will also provide a reference configuration for moving topic data to S3 from Kafka for long-term data retention purposes.

**Key Concepts**

**1. Kafka Connectors**

Kafka Connectors are used to integrate Kafka with external systems such as databases, cloud services, file systems, and other messaging systems. They allow Kafka to import or export data to and from these systems.

One of the key features of CFK is its support for **Declarative Connectors**. Declarative connectors enable users to define and manage their Kafka Connect connectors declaratively, using Kubernetes-style YAML configuration files. This approach enhances automation, versioning, and consistency when managing connectors in a Kubernetes environment.

Kafka Connectors are of two types:

* **Source Connectors**: These connectors pull data from an external system into Kafka topics.
* **Sink Connectors**: These connectors push data from Kafka topics into external systems.
  + This document will focus on Sink connectors as the use case here is backing up Topic data to S3 buckets.

**2. Declarative Management**

With declarative management, infrastructure and application configurations are described in a declarative manner—typically in YAML files. This allows Kubernetes to manage resources based on these declarations, ensuring that the desired state of the system is continuously maintained.

In the case of Confluent for Kubernetes, declarative management of connectors allows users to define Kafka Connect connectors through YAML manifests, which Kubernetes will use to deploy and maintain connectors in the desired state.

**3. Confluent for Kubernetes (CFK)**

CFK provides Kubernetes-native resources to manage and deploy Confluent services such as Kafka brokers, Kafka Connect, Schema Registry, and more. It offers a set of operators that interact with Kubernetes resources and enable easy management of Confluent components.

The Confluent Operator, a key component of CFK, is responsible for managing these resources and handling operations such as scaling, rolling updates, and ensuring health checks.

**Benefits of Declarative Connectors**

1. **Consistency**: Declarative configuration ensures that connectors are always deployed with the same settings across different environments, reducing the risk of configuration drift.
2. **Automation**: Declarative connectors can be automated with CI/CD pipelines. Changes to connector configurations can be tracked and deployed automatically, improving operational efficiency.
3. **Version Control**: Connector configurations can be stored in version-controlled repositories, enabling teams to manage, review, and roll back changes easily.
4. **Kubernetes-Native**: Since connectors are defined using Kubernetes YAML files, they benefit from the rich ecosystem of Kubernetes tools for monitoring, scaling, and fault tolerance.
5. **Self-Healing**: Kubernetes will automatically ensure that connectors are running in the desired state, making use of its self-healing capabilities. If a connector fails, Kubernetes will attempt to restart or redeploy it as needed.
6. **Simplified Resource Management**: With declarative connectors, Kafka Connect resources are simplified and easily managed as Kubernetes resources (e.g., pods, deployments, services).

**Using Declarative Connectors To build Kafka-S3 Connectivity**

**1. Kafka Connect Custom Resources**

In CFK, Kafka Connect is broken down into two different **Custom Resources** (CRs). These CRs represent the Kafka Connect Cluster, and its respective connectors.

The custom resource for declarative connectors is the Confluent Connector CR. This CR contains specifications for the connectors to be deployed within the Kafka Connect cluster.

A typical declarative connector YAML file defines:

* The **namespace** in which the connector should run.
* The **task max** which is essentially the number of concurrent tasks to spawn to perform the required partition reads necessary for the topics the connector is connected to (for example, if there were 6 partitions for a topic and task max was 3 then each task would read from 2 partitions to streamline operations)
* The **connector class** and its configuration (e.g., connector type, tasks, topics).
* Optional **configuration overrides**, such as authentication settings, connector-specific configurations, and parameters for the connector's behavior.

The custom resource for a kafka connect cluster is the Confluent Connect CR. This CR contains specifications for the connect cluster to be deployed in the designated confluent operator namespace.

A typical declarative connect cluster YAML file defines:

* The **namespace** in which the connect cluster should run
* The **replicas** the cluster should operate within (1 for testing, but usually 3 for prod)
* The **plugins** which the cluster will use for various connectors that are provisioned (e.g. S3 sink connector plugin)

**2. Kubernetes Manifests**

To deploy and manage connectors, users create YAML manifests that define the desired state of the Kafka Connect deployment, including the connector specifications. These manifests are applied to the Kubernetes cluster using kubectl commands.

In the case of this developmental scenario, the following YAML files were deployed for the connect cluster as well as its S3 Sink connector counterpart:

<https://linktoyamlfiles>

In this example:

* A Confluent Connect cluster is defined with 1 replica and a single plugin for the S3 Sink Connector.
* A Confluent Connect Connector is defined with 3 task max to use the specified S3 Sink connector, which will export specified topic data to a designated S3 buckets directory at a rate of every 5 new topic messages.

**3. Applying the YAML Files**

Once the YAML files are prepared, they are applied using standard Kubernetes tooling:

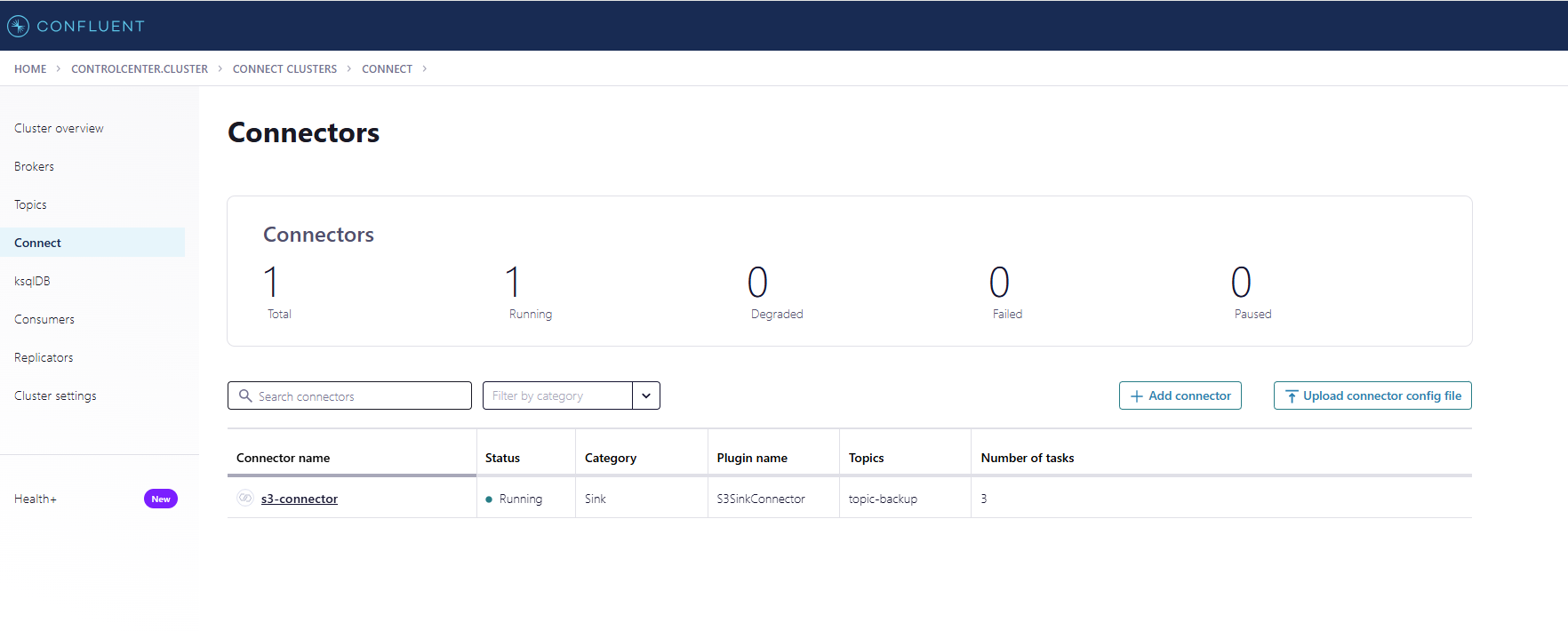
bash

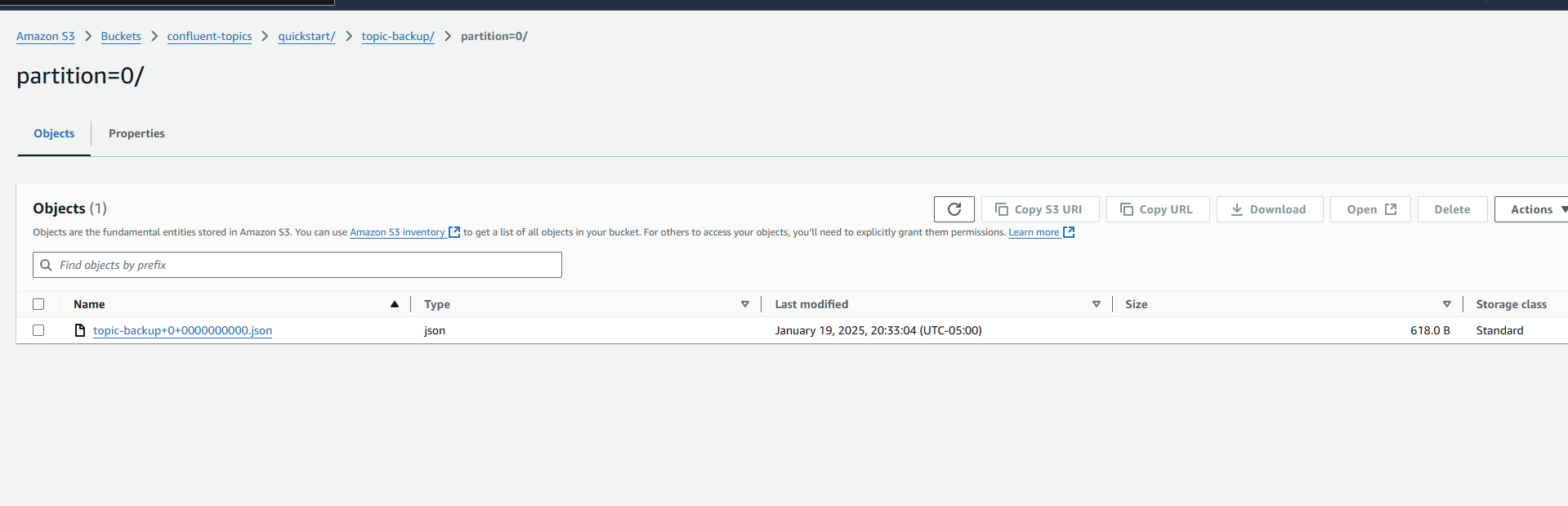
Copy

kubectl apply -f connector-definition.yaml

Kubernetes will process the YAML files and deploy the connector within the specified Kafka Connect cluster. If the connector configuration needs to be updated, users can modify the YAML and reapply it, allowing for seamless updates.

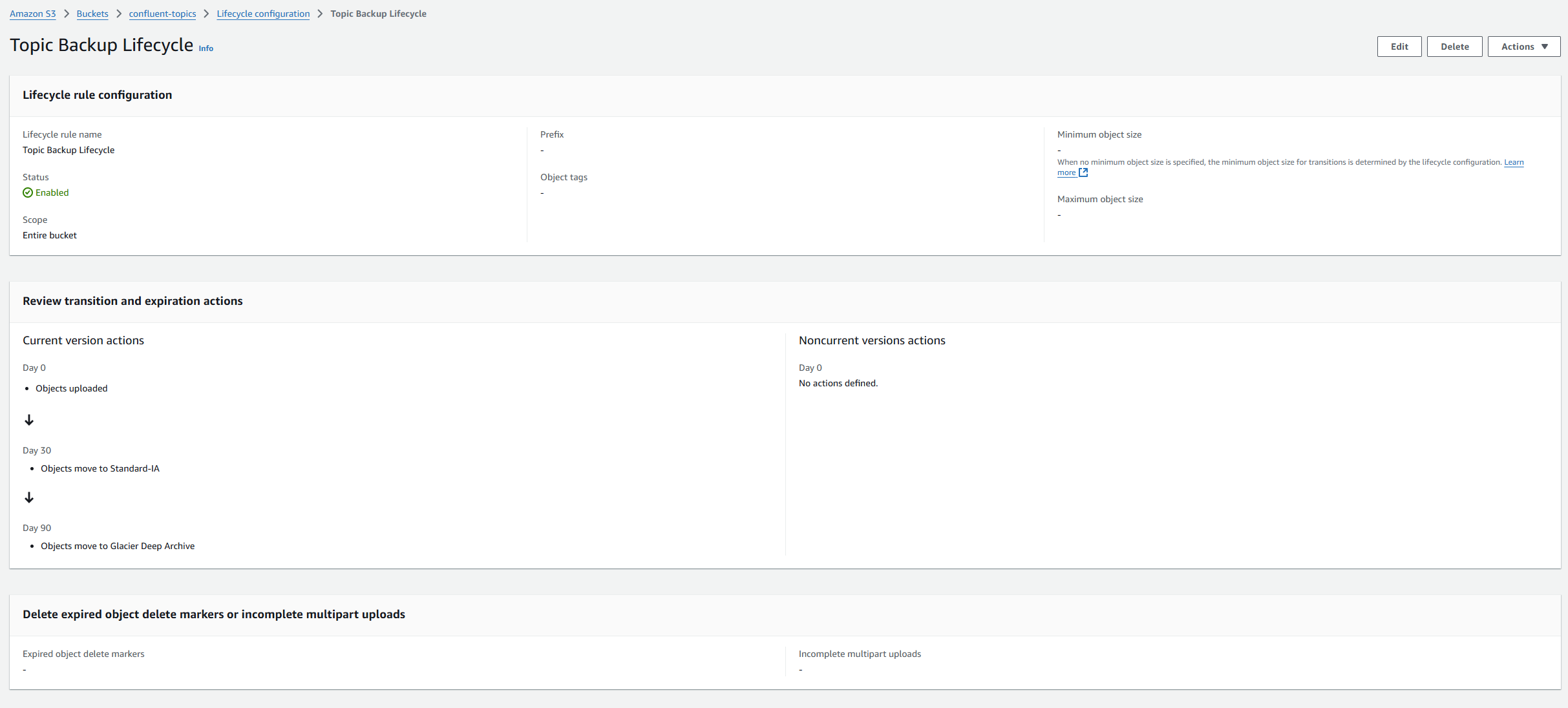
Once applied, the connector can be validated either via the CLI using kubectl (kubectl get connectors) or the Confluent Control Center UI





**4. Configuring S3 Lifecycle Policies**

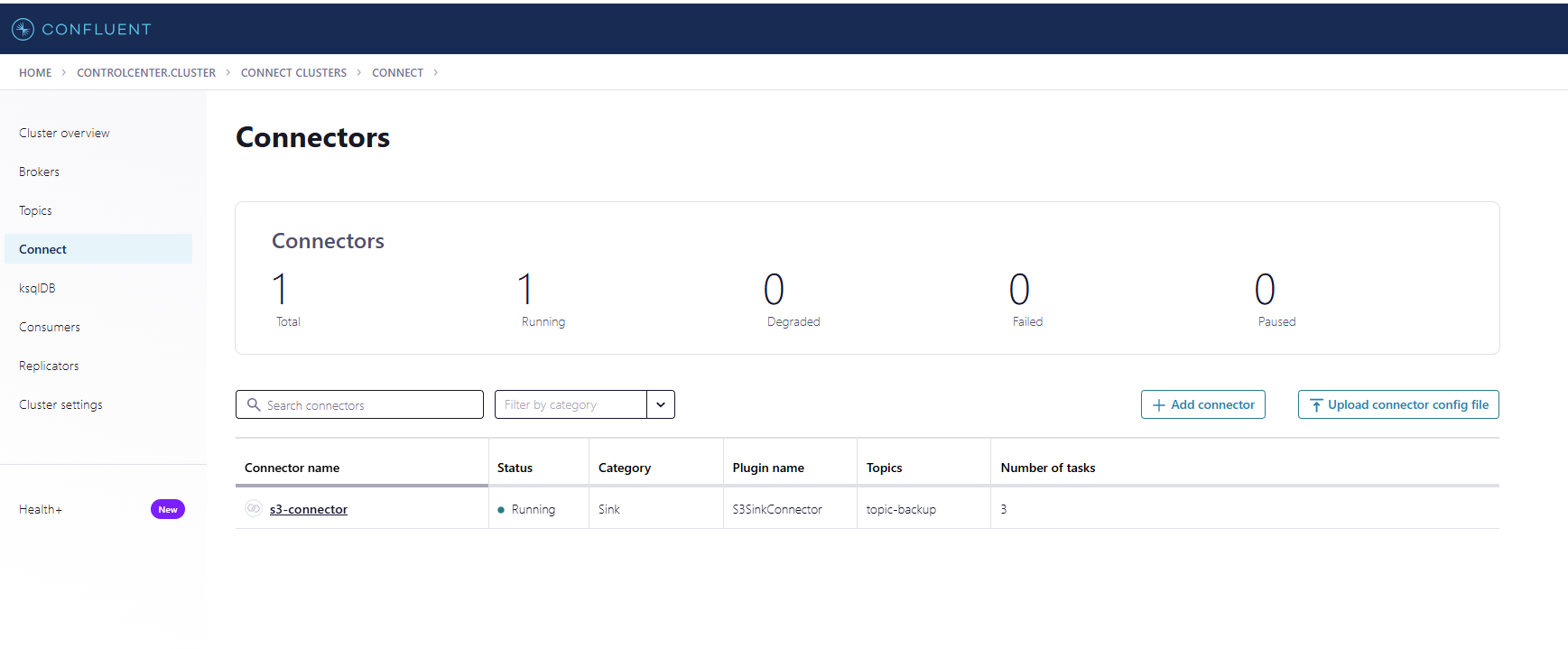
Now that the connectors are deployed, Kubernetes will ensure they are running as expected. Having data in S3 now, we can leverage S3 Lifecycle Policies to manage cost of retaining historical topic data. The current strategy is 0-30 Standard, 30-90 Standard-IA, 90+ Glacier Deep Archive.



**5. Troubleshooting**

If for any reason you are not seeing backups in S3, you will need to investigate at what point of the integration there is an issue.

You can begin by looking at the Confluent Control Center UI for any issues with the connector there.



In the case here, the connector reflects a “Running” state which is a healthy state. If it were degraded or failed then you would investigate connector logs.

To check collector logs simply run “kubectl describe connector **CONNECTOR** -n **NAMESPACE**”

The connector can also show running state and still not send data to S3. Usually in this scenario there is a problem with the message format, as kafka is strict about message formatting when performing any sort of processing.

In the configuration I have listed above, we have the Dead Letter Queue (DLQ) enabled for failed message processing. This is very useful for troubleshooting a failed Kafka Connect integration. What this does is essentially move failed messages to another topic for inspection and puts the error message in the header value.

More Info: <https://www.confluent.io/blog/kafka-connect-deep-dive-error-handling-dead-letter-queues/>

Example (DLQ Topic):

