splunk-guide-for-kafka-monitoring Documentation

Release 1

Guilhem Marchand

Nov 06, 2020

Requirements

The unified guide for Kafka and Confluent monitoring with Splunk provides a full step by step guidance for monitoring with Splunk, with the following main concepts:

- realtime event logging
- realtime and high performance metric store
- evolutive and efficient alerting
- scalability and resiliency
- compatibility with traditional bare metails/virtual machines deployments and Kubernetes containers deployments

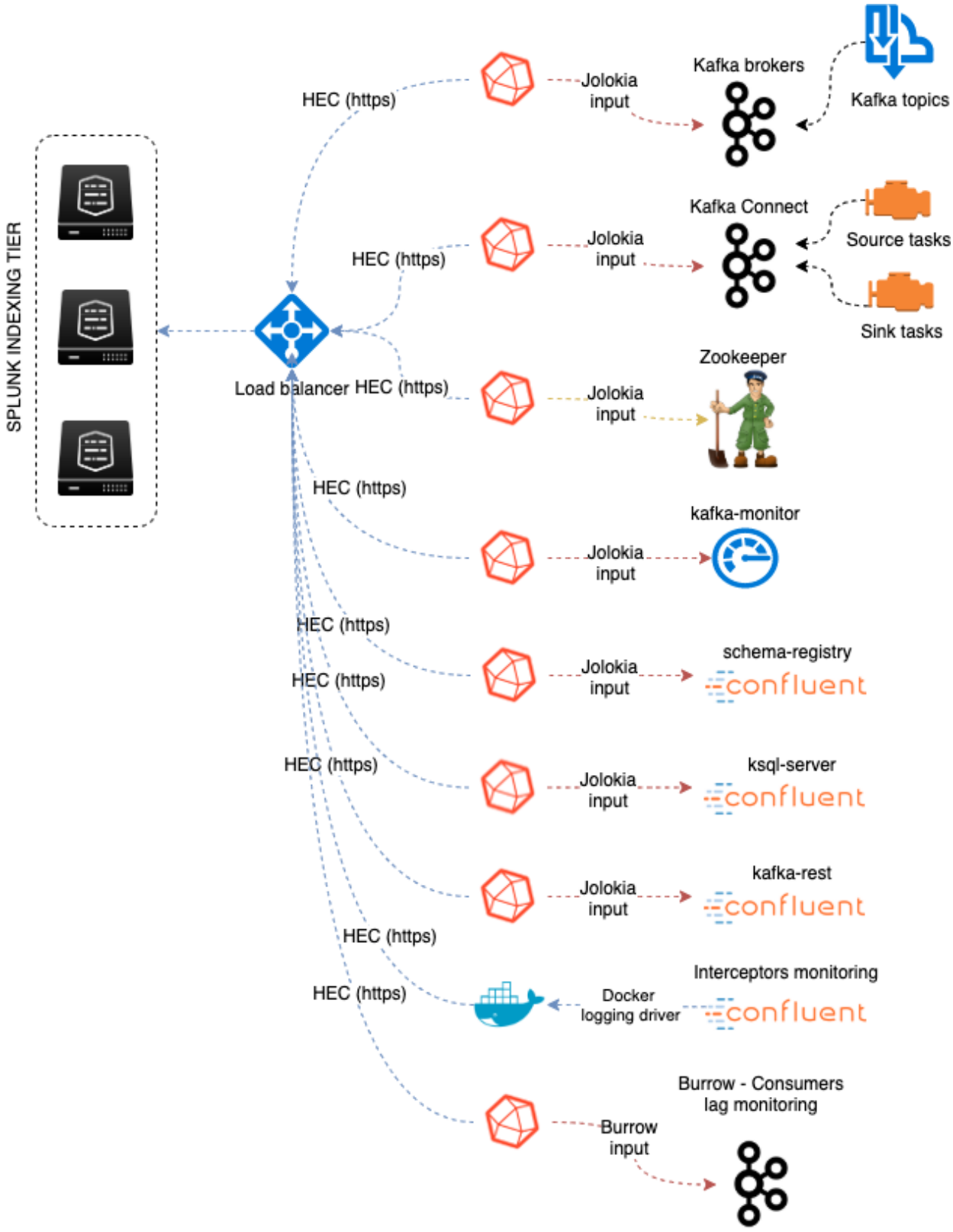
As a basis, the following components are being natively managed:

- Zookeeper
- Apache Kafka Brokers
- Apache Kafka Connect
- Confluent schema-registry
- Confluent ksql-server
- Confluent kafka-rest
- Kafka SLA and end to end monitoring with the Linkedin Kafka monitor

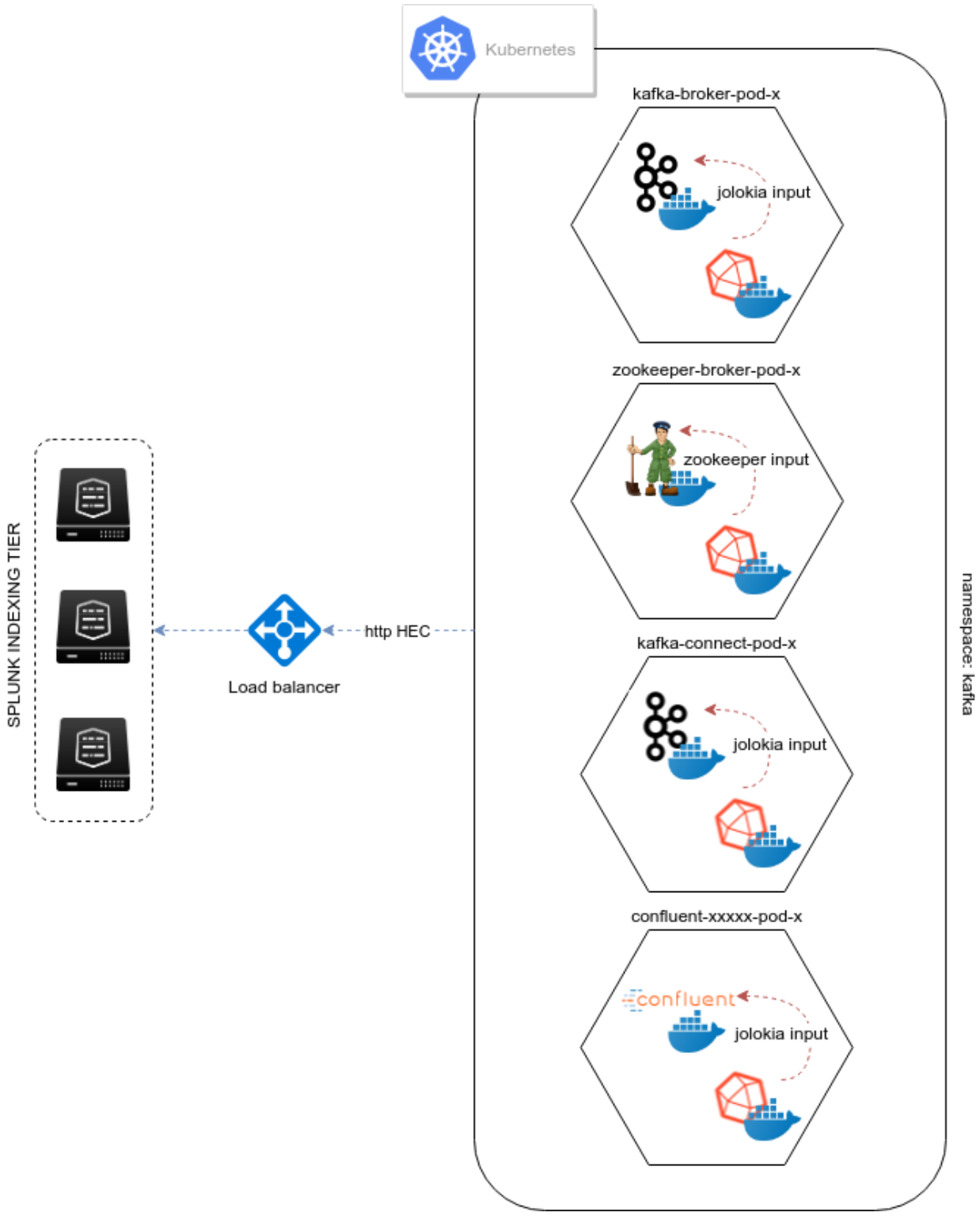
The following components are leveraged:

- Splunk (!)
- Jolokia, connector interface for JMX
- Telegraf, the plugin-driven server agent for collecting & reporting metrics

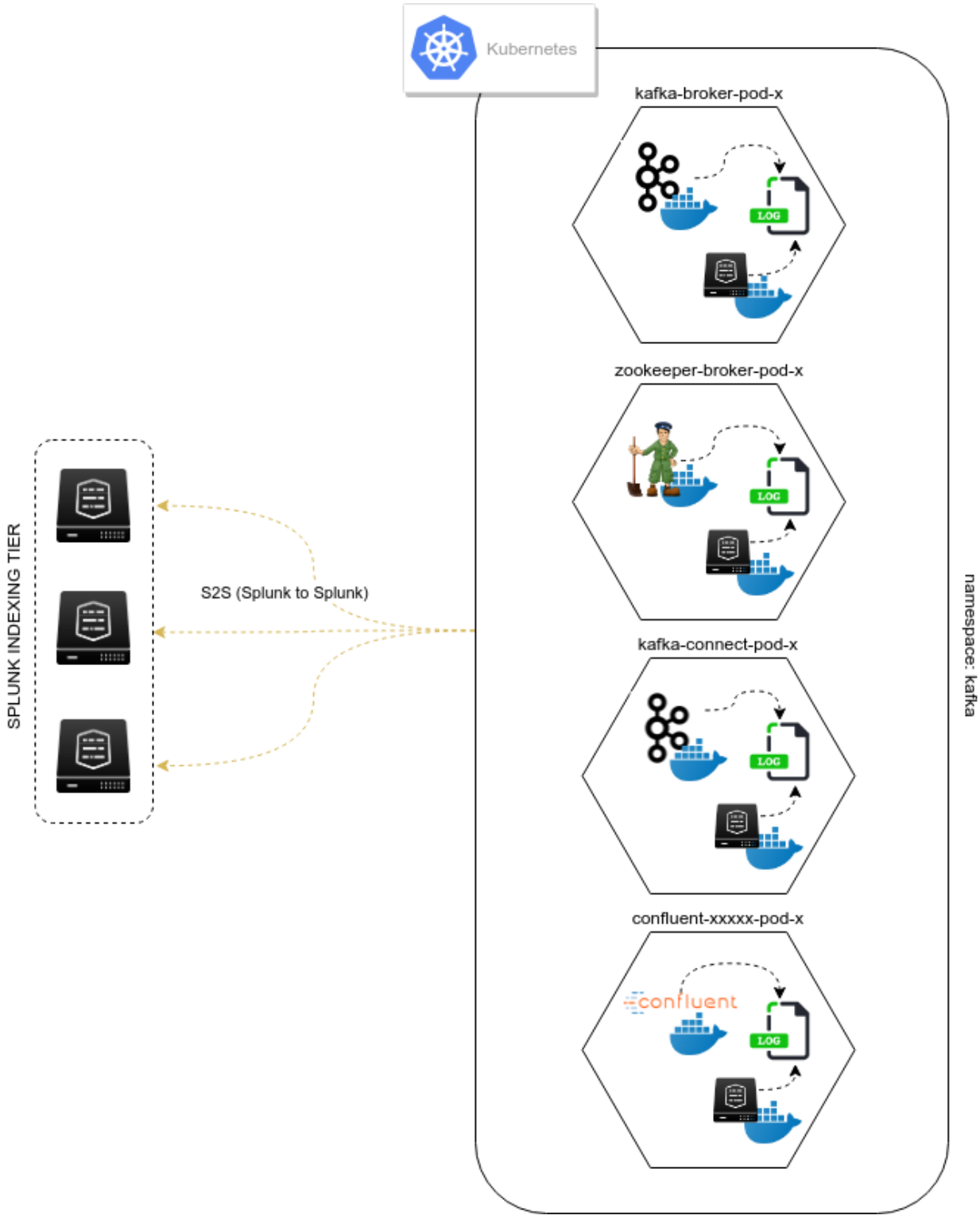
metrics collection diagram example



Kubernetes metrics collection diagram - sidecar containers metrics collection by Telegraf to Jolokia:



Kubernetes events logging ingestion diagram - sidecar containers Splunk Universal Forwarders reading logs in pod shared volumes:


Kubernetes
kafka-broker-pod-x
LOG
zookeeper-broker-pod-x
LOG
kafka-connect-pod-x
LOG
confluent-xxxxx-pod-x
confluent
LOG
namespace: kafka
SPLUNK INDEXING TIER
S2S (Splunk to Splunk)

CHAPTER 1

Implementation:

1.1 Pre-requisites

1.1.1 Splunk

We use the Splunk metric store that was introduced in the early version of 7.0, any Splunk version starting 7.0.x will comply with the requirements.

1.1.2 Splunk Universal Forwarder

A Splunk Universal Forwarder instance is deployed on each component to achieve monitoring and indexing of the application events.

In most cases, you will use a Splunk Deployment Server to manage configuration of the Universal Forwarder.

1.2 Running Kafka in Kubernetes

Deploying and running successfully Kafka in Kubernetes is not the purpose of the guide, however it is interesting to share some tips about this task which can be quite complex.

1.2.1 Confluent platform with helm

Confluent provides a very interesting set of configurations with helm that you can use to setup and build your infrastructure in Kubernetes:

- https://docs.confluent.io/current/installation/installing_cp/cp-helm-charts/docs/index.html
- https://github.com/confluentinc/cp-helm-charts

The templates provides in this guide are built on top of the Confluent platform and these configurations, which however can be adapted to run on any kind of deployment.

Testing with minikube

Make sure you start minikube with enough memory and cpu resources, example:

```
minikube start --memory 8096 --cpus 4
```

Before starting the deployment with helm, you can use the following configuration to create the require storage classes:

minikube_storageclasses.yml

```
---
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: kafka-broker
provisioner: k8s.io/minikube-hostpath
reclaimPolicy: Retain
---
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: kafka-zookeeper
provisioner: k8s.io/minikube-hostpath
reclaimPolicy: Retain
```

Then apply:

```
kubectl create -f minikube_storageclasses.yml
```

Modify the values.yml to include the storage classes and some restrictions on the containers to get it successful:

values.yaml

```
## ------------------------------------------------------
## Zookeeper
## ------------------------------------------------------
cp-zookeeper:
  enabled: true
  servers: 3
  image: confluentinc/cp-zookeeper
  imageTag: 5.0.1
  ## Optionally specify an array of imagePullSecrets. Secrets must be manually␣
↪created in the namespace.
  ## https://kubernetes.io/docs/concepts/containers/images/#specifying-
↪imagepullsecrets-on-a-pod
  imagePullSecrets:
  #  - name: "regcred"
  heapOptions: "-Xms512M -Xmx512M"
  persistence:
    enabled: true
    ## The size of the PersistentVolume to allocate to each Zookeeper Pod in the␣
↪StatefulSet. For
    ## production servers this number should likely be much larger.
    ##
    ## Size for Data dir, where ZooKeeper will store the in-memory database snapshots.
    dataDirSize: 5Gi
    dataDirStorageClass: "kafka-zookeeper"
```

(continues on next page)

(continued from previous page)

```
    ## Size for data log dir, which is a dedicated log device to be used, and helps␣
↪avoid competition between logging and snaphots.
    dataLogDirSize: 5Gi
    dataLogDirStorageClass: "kafka-zookeeper"
  resources:
  ## If you do want to specify resources, uncomment the following lines, adjust them␣
↪as necessary,
  ## and remove the curly braces after 'resources:'
    limits:
     cpu: 100m
     memory: 256Mi
    requests:
     cpu: 100m
     memory: 256Mi

## ------------------------------------------------------
## Kafka
## ------------------------------------------------------
cp-kafka:
  enabled: true
  brokers: 3
  image: confluentinc/cp-kafka
  imageTag: 5.0.1
  ## Optionally specify an array of imagePullSecrets. Secrets must be manually␣
↪created in the namespace.
  ## https://kubernetes.io/docs/concepts/containers/images/#specifying-
↪imagepullsecrets-on-a-pod
  imagePullSecrets:
  #  - name: "regcred"
  heapOptions: "-Xms512M -Xmx512M"
  persistence:
    enabled: true
    storageClass: "kafka-broker"
    size: 5Gi
    disksPerBroker: 1
  resources:
  ## If you do want to specify resources, uncomment the following lines, adjust them␣
↪as necessary,
  ## and remove the curly braces after 'resources:'
    limits:
     cpu: 200m
     memory: 512Mi
    requests:
     cpu: 200m
     memory: 512Mi

## ------------------------------------------------------
## Schema Registry
## ------------------------------------------------------
cp-schema-registry:
  enabled: true
  image: confluentinc/cp-schema-registry
  imageTag: 5.0.1
  ## Optionally specify an array of imagePullSecrets. Secrets must be manually␣
↪created in the namespace.
  ## https://kubernetes.io/docs/concepts/containers/images/#specifying-
↪imagepullsecrets-on-a-pod
```

(continues on next page)

(continued from previous page)

```
  imagePullSecrets:
  #  - name: "regcred"
  heapOptions: "-Xms512M -Xmx512M"
  resources:
  ## If you do want to specify resources, uncomment the following lines, adjust them␣
↪as necessary,
  ## and remove the curly braces after 'resources:'
    limits:
     cpu: 100m
     memory: 512Mi
    requests:
     cpu: 100m
     memory: 512Mi

## ------------------------------------------------------
## REST Proxy
## ------------------------------------------------------
cp-kafka-rest:
  enabled: true
  image: confluentinc/cp-kafka-rest
  imageTag: 5.0.1
  ## Optionally specify an array of imagePullSecrets. Secrets must be manually␣
↪created in the namespace.
  ## https://kubernetes.io/docs/concepts/containers/images/#specifying-
↪imagepullsecrets-on-a-pod
  imagePullSecrets:
  #  - name: "regcred"
  resources:
  ## If you do want to specify resources, uncomment the following lines, adjust them␣
↪as necessary,
  ## and remove the curly braces after 'resources:'
    limits:
     cpu: 100m
     memory: 256Mi
    requests:
     cpu: 100m
     memory: 256Mi

## ------------------------------------------------------
## Kafka Connect
## ------------------------------------------------------
cp-kafka-connect:
  enabled: true
  image: confluentinc/cp-kafka-connect
  imageTag: 5.0.1
  ## Optionally specify an array of imagePullSecrets. Secrets must be manually␣
↪created in the namespace.
  ## https://kubernetes.io/docs/concepts/containers/images/#specifying-
↪imagepullsecrets-on-a-pod
  imagePullSecrets:
  #  - name: "regcred"
  resources: {}
  ## If you do want to specify resources, uncomment the following lines, adjust them␣
↪as necessary,
  ## and remove the curly braces after 'resources:'
    #limits:
     #cpu: 100m
```

(continues on next page)

(continued from previous page)

```
     #memory: 512Mi
    #requests:
     #cpu: 100m
     #memory: 512Mi

## ------------------------------------------------------
## KSQL Server
## ------------------------------------------------------
cp-ksql-server:
  enabled: true
  image: confluentinc/cp-ksql-server
  imageTag: 5.0.1
  ## Optionally specify an array of imagePullSecrets. Secrets must be manually␣
↪created in the namespace.
  ## https://kubernetes.io/docs/concepts/containers/images/#specifying-
↪imagepullsecrets-on-a-pod
  imagePullSecrets:
  #  - name: "regcred"
  ksql:
    headless: false
```

Starting helm:

The templates provided are built on the naming convention of a helm installion called "confluent-oss" in a name space called "kafka":

helm3 command:

```
helm install confluent-oss confluentinc/cp-helm-charts
```

The following command is valid for helm2 only, and left for historical purposes

```
helm install cp-helm-charts --name confluent-oss --namespace kafka
```

The helm installation provided by Confluent will create:

- Zookeeper cluster in a statefulSet
- Kafka Brokers cluster in a statefulSet
- Kafka Connect in a Deployment
- Confluent Schema registry in a Deployment
- Confluent ksql-server in a Deployment
- Confluent kafka-rest in a Deployment

1.3 Running Kafka in Docker

Out of Kubernetes, you may as well simply run a Kafka infrastructure on Docker, see the following links for examples and configuration samples:

- https://docs.confluent.io/current/installation/docker/installation/index.html#cpdocker-intro
- https://github.com/confluentinc/cp-all-in-one/blob/latest/cp-all-in-one/docker-compose.yml
- https://github.com/guilhemmarchand/kafka-docker-splunk

This covers events logging and metric collections when running Kafka on containers.

1.4 Running Kafka on premise

You can run your Kafka infrastructure on a traditional deployment with dedicated bare metal servers or VMs, on premise infrastruture and private Cloud, see the following links:

- https://kafka.apache.org/quickstart
- https://docs.confluent.io/current/installation/installing_cp/index.html

This guide covers events logging and metric collection when running on Kubernetes.

1.5 Chapter 1: Events logging

1.5.1 Pre-requisites

Download the Technology Addon for Technology addon for Kafka streaming platform:

https://splunkbase.splunk.com/app/4302

The full and dedicated documentation site:

https://ta-kafka-streaming-platform.readthedocs.io

1.5.2 Splunk configuration

1. Index creation

Create a new index that will be used to store the events from the whole infrastructure. By default, the Technology Addon uses an event indexed named `kafka`.

Ensure that the index was created in your environment before you continue.

You can change the index name by customising the inputs provided by the Splunk application.

2. Deployment of the Technical Addon to the Splunk core infrastructure

Deploy the Technology Addon to:

- **The indexers** (cluster master then push the cluster bundle for clusters, as a normal application for a standalone server)
- **Heavy Forwarders** if they are being used as intermediate forwarders before reaching the indexers
- **Search Heads** (SHC deployer if running a search head cluster, as a normal application for a standalone server)
- **Deployment Server** in deployment-apps to be pushed to the Universal Forwarders

1.5.3 Events logging in dedicated servers (bare metal, VMs)

1. Verify the log4j format in used

The Technology Addon uses the most optimised configuration to guarantee a proper parsing of the events, specially with a perfect management of multi-line events. (such as Java stacktraces)

Sourcetype definitions assume the usage of the default format used by Kafka and Confluent Enterprise/OSS

Example:

```
[2018-11-20 22:02:15,435] INFO Registered kafka:type=kafka.Log4jController MBean␣
↪(kafka.utils.Log4jControllerRegistration$)
```

This relies of your log4j properties files having the following format: (again this is the default format)

```
DatePattern='.'yyyy-MM-dd-HH
layout.ConversionPattern=[%d] %p %m (%c)%n
```

If you are relying on a different log format, copy the default/props.conf to local/ and achieve the relevant customization.

Notes: The JVM garbage collector has its own format that is unlikely to be customized

4. Verify and collect log locations of the Kafka components

As logging directories can be customized, and some components may require some minimal configurations to log properly, you first action is to verify, collect and eventually achieve the required changes.

Apache Kaka - Confluent Enterprise / OSS

Zookeeper

By default, Confluent may use the same logging location for both Zookeeper and Kafka brokers, suggested configuration to avoid this:

Configuring the systemd for Zookeeper:

- Edit: `/lib/systemd/system/confluent-zookeeper.service`
- Configure the logs location with the LOG_DIR environment variable

```
[Unit]
Description=Apache Kafka - ZooKeeper
Documentation=http://docs.confluent.io/
After=network.target
```

(continues on next page)

(continued from previous page)

```
[Service]
Type=simple
User=cp-kafka
Group=confluent
ExecStart=/usr/bin/zookeeper-server-start /etc/kafka/zookeeper.properties
Environment="LOG_DIR=/var/log/zookeeper"
TimeoutStopSec=180
Restart=no

[Install]
WantedBy=multi-user.target
```

- Create the log directory:

```
sudo mkdir /var/log/zookeeper
sudo chown cp-kafka:confluent /var/log/zookeeper
```

- Restart Zookeeper and verify that logs are properly generated in the directory:

```
sudo systemctl daemon-reload
sudo systemctl status confluent-zookeeper
```

Kafka brokers

By default, the Confluent platform generates brokers logs in the following location:

```
/var/log/kafka
```

Kafka Connect

Kafka Connect does not log to a file by default, it only logs to the console.

To change this behaviour, you need to edit the log4j configuration:

Configuring the systemd service file for Connect:

- Edit: `/lib/systemd/system/confluent-kafka-connect.service`
- Configure the logs location with the LOG_DIR environment variable

```
[Unit]
Description=Apache Kafka Connect - distributed
Documentation=http://docs.confluent.io/
After=network.target confluent-kafka.target

[Service]
Type=simple
User=cp-kafka-connect
Group=confluent
ExecStart=/usr/bin/connect-distributed /etc/kafka/connect-distributed.properties
Environment="LOG_DIR=/var/log/connect"
TimeoutStopSec=180
Restart=no
```

(continues on next page)

(continued from previous page)

```
[Install]
WantedBy=multi-user.target
```

- Create the log directory:

```
sudo mkdir /var/log/connect
sudo chown cp-kafka-connect:confluent /var/log/connect
```

Configuring log4j:

- Edit: `/etc/kafka/connect-log4j.properties`
- Add a file appender:

```
log4j.rootLogger=INFO, stdout, FILE

log4j.appender.FILE=org.apache.log4j.DailyRollingFileAppender
log4j.appender.FILE.DatePattern='.'yyyy-MM-dd-HH
log4j.appender.FILE.File=${kafka.logs.dir}/connect.log
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.ConversionPattern=[%d] %p %m (%c)%n

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=[%d] %p %m (%c:%L)%n

log4j.logger.org.apache.zookeeper=ERROR
log4j.logger.org.I0Itec.zkclient=ERROR
log4j.logger.org.reflections=ERROR
```

- Restart Connect and verify that the log file is being created:

```
sudo systemctl daemon-reload
sudo systemctl restart confluent-kafka-connect
```

schema-registry

By default, the Confluent platform generates Schema registry log in the following location:

```
/var/log/confluent/schema-registry
```

ksql-server

ksql-server does not log to a file by default, it only logs to the console.

Notes: By default, the systemd already defines the log directory location, which should already be existing with the correct permissions.

Verifying the systemd service file for ksql:

- Edit: */lib/systemd/system/confluent-ksqldb.service*
- Verify the logs location with the LOG_DIR environment variable

```
[Unit]
Description=Streaming SQL engine for Apache Kafka
Documentation=http://docs.confluent.io/
After=network.target confluent-kafka.target confluent-schema-registry.target

[Service]
Type=simple
User=cp-ksql
Group=confluent
Environment="LOG_DIR=/var/log/confluent/ksql"
ExecStart=/usr/bin/ksql-server-start /etc/ksqldb/ksql-server.properties
TimeoutStopSec=180
Restart=no

[Install]
WantedBy=multi-user.target
```

- Verify and create the log directory if required:

```
sudo mkdir -p /var/log/confluent/ksql
sudo chown cp-kafka-connect:confluent /var/log/confluent/ksql
```

Configuring log4j:

- Edit: */etc/ksqldb/log4j.properties*
- Add a file appender:

```
log4j.rootLogger=INFO, stdout, FILE

log4j.appender.FILE=org.apache.log4j.DailyRollingFileAppender
log4j.appender.FILE.DatePattern='.'yyyy-MM-dd-HH
log4j.appender.FILE.File=${ksql.log.dir}/ksql-server.log
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.ConversionPattern=[%d] %p %m (%c)%n

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=[%d] %p %m (%c:%L)%n

log4j.appender.streams=org.apache.log4j.ConsoleAppender
log4j.appender.streams.layout=org.apache.log4j.PatternLayout
log4j.appender.streams.layout.ConversionPattern=[%d] %p %m (%c:%L)%n

log4j.logger.kafka=ERROR, stdout
log4j.logger.org.apache.kafka.streams=INFO, streams
log4j.additivity.org.apache.kafka.streams=false
log4j.logger.org.apache.zookeeper=ERROR, stdout
log4j.logger.org.apache.kafka=ERROR, stdout
log4j.logger.org.I0Itec.zkclient=ERROR, stdout
```

- Restart ksql-server and verify that the log file is being created:

```
sudo systemctl daemon-restart
sudo systemctl restart confluent-ksqldb
```

kafka-rest

By default, the Confluent platform generates kafka-rest logs in the following location:

```
/var/log/confluent/kafka-rest
```

4. Deployment to the Splunk Universal Forwarders

This assumes that:

- You have deployed a Splunk Universal Forwarder (UF) on each instance to be monitored (Zookeeper, brokers, etc)
- UFs are properly configured and forwarding to your Splunk Indexing layer (`index=_internal sourcetype=splunkd` returns Splunk internal events from the UFs)
- Manage deployment to the UFs via a Splunk Deployment server (although you could use any automation tool of your choice)

IMPORTANT: By default, all inputs are disabled and must be enabled depending on your needs

- Extract the content of the Technology Addon archive in your deployment server

example:

```
/opt/splunk/etc/deployment-apps/TA-kafka-streaming-platform
```

- Create a local directory, copy the default inputs.conf, enable each monitor input required and achieve any customization required, such as custom paths to log directories:

```
cd /opt/splunk/etc/deployment-apps/TA-kafka-streaming-platform
mkdir local
cp -p default/inputs.conf local/
```

- To enable an input monitor:

replace

```
disabled = true
```

by

```
disabled = false
```

- Finally, create a server class in the deployment server that matches your Kafka infrastructure hosts, associate with the Technology Addon. (ensure to restart splunkd !)
- Once the TA and its configuration has been deployed to the UFs, the logs collection will start immediately.

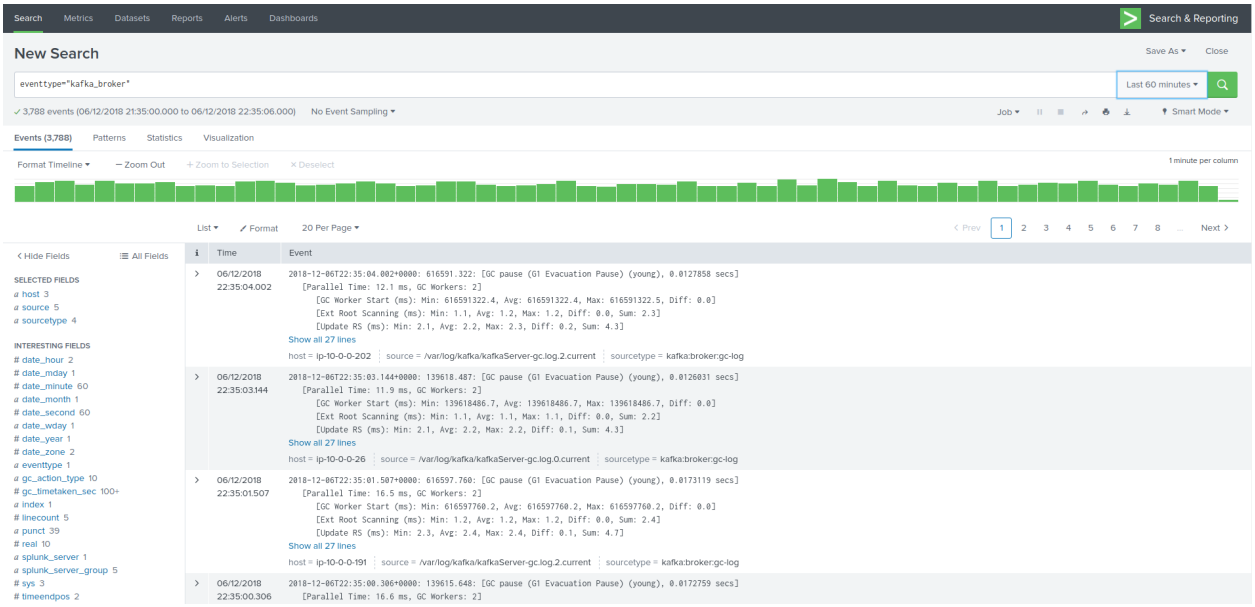
Verify

The easiest and first verification is obviously looking at the index content:

```
index=kafka
```

Next verification is verifying the eventtypes definition, example:

```
eventtype=kafka_broker
```



1.5.4 Events logging in Kubernetes and docker containers

A perfect events logging management requires a different approach in a Kubernetes deployment.

As a basis, each container produces output logging in its standard output, which you can index in Splunk using the Splunk Technical Addon for Kubernetes:

https://splunkbase.splunk.com/app/3991

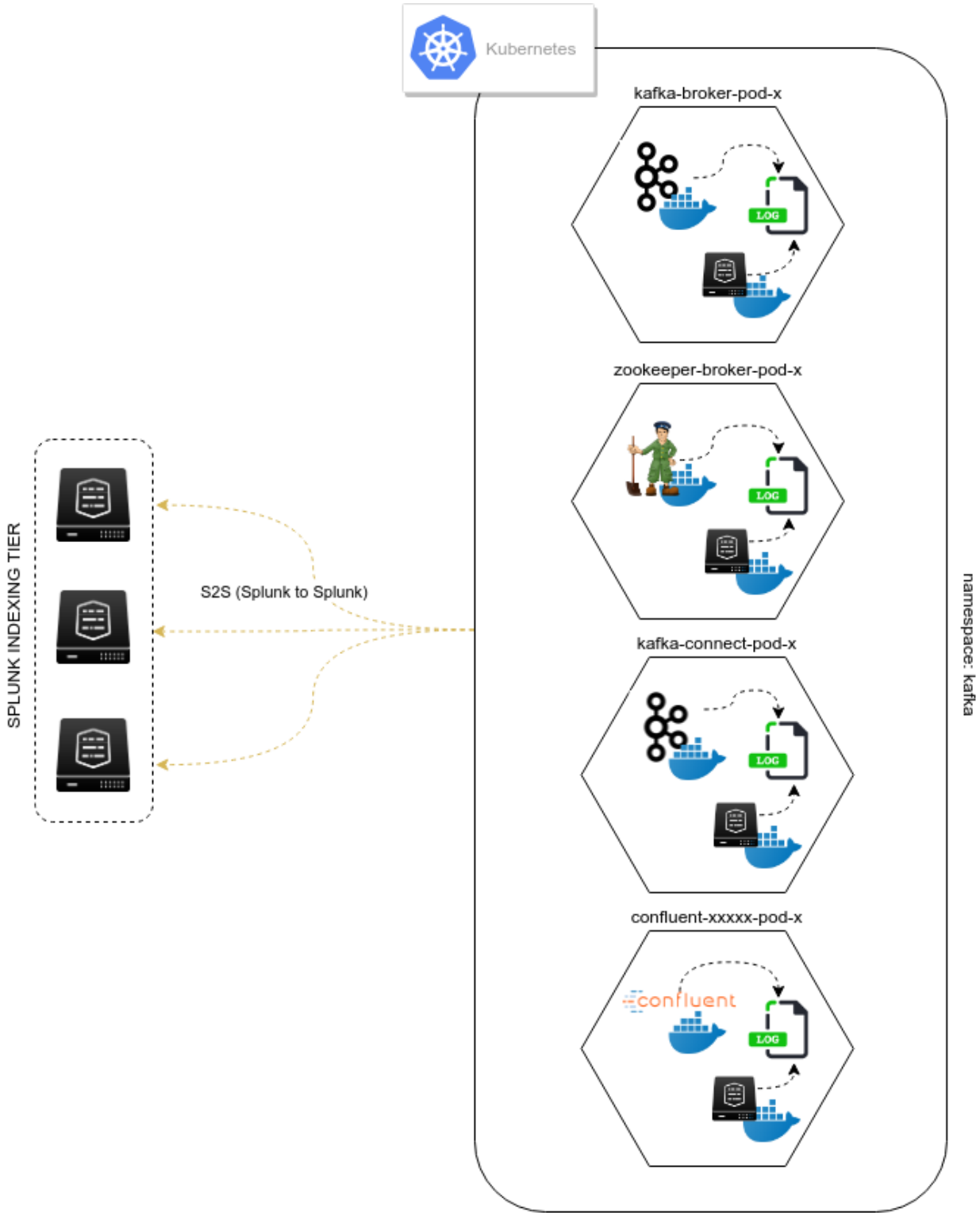
However, the multi-line management and the differentiation between the different parts of the sub systems logging is a dead end path. (Think about Java stacktraces, garbage collector logging, etc.)

The approach provided is a different approach that is entirely in the philosophy of Kubernetes and Splunk, by using the Kubernetes pods capabilities:

- Each Kafka or Confluent container running in a statefulSet or Deployment is updated to produce logs locally on the container (in addition with its standard output)
- A Splunk Universal Forwarder is created and configured to run in the each pod, which is called a sidecar container (running Splunk Forwarders in a container is now fully supported)
- Splunk Universal Forwarders are connected to your Splunk Deployment infrastructure, and managed just as usual
- The containers running in a same pod automatically share the log directory as a volume, Kafka component produces logs, Splunk monitors these
- Anytime the pod is destroyed and re-created, the Splunk containers is automatically re-created and configured

This is a resilient, scalable and reliable approach that is entirely compatible, relevant and standard with Kubernetes, Kafka and Confluent components, and Splunk.

events logging collection diagram - sidecar Splunk Universal Forwarder containers:



Zookeeper monitoring

Link: Zookeeper logging

Kafka Brokers monitoring

Link: Kafka Brokers logging

Kafka Connect monitoring

Link: Kafka Connect logging

Confluent schema-registry monitoring

Link: Confluent shema-registry logging

Confluent kafka-rest monitoring

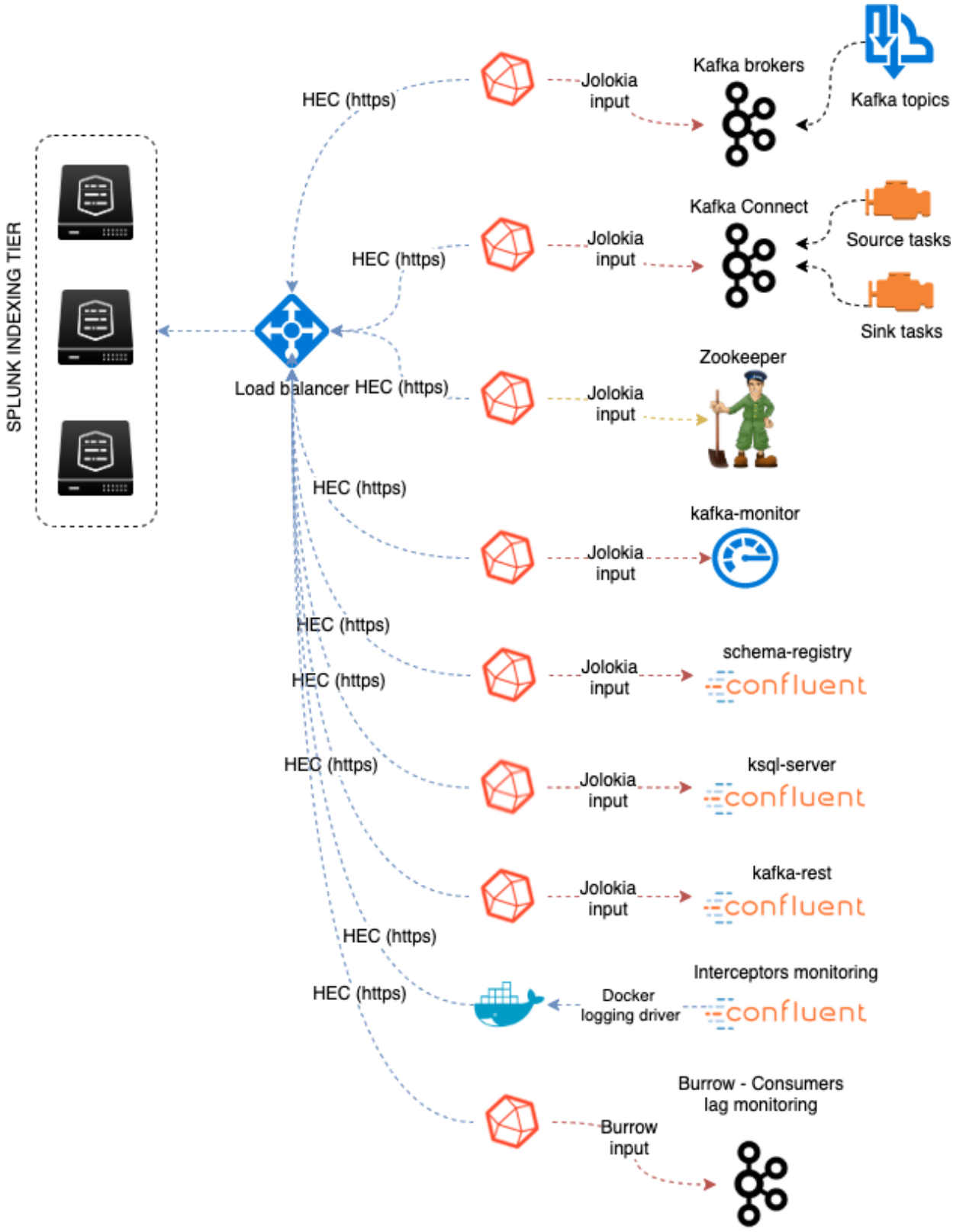
Link: Confluent kafka-rest logging

Confluent ksql-server monitoring

Link: Confluent ksql-server logging

1.6 Chapter 2: Metrics

Data collection diagram overview:



1.6.1 Splunk configuration

Index definition

The application relies by default on the creation of a metrics index called "telegraf_kafka":

indexes.conf example with no Splunk volume::

```
[telegraf_kafka]
coldPath = $SPLUNK_DB/telegraf_kafka/colddb
datatype = metric
homePath = $SPLUNK_DB/telegraf_kafka/db
thawedPath = $SPLUNK_DB/telegraf_kafka/thaweddb
```

indexes.conf example with Splunk volumes::

```
[telegraf_kafka]
coldPath = volume:cold/telegraf_kafka/colddb
datatype = metric
homePath = volume:primary/telegraf_kafka/db
thawedPath = $SPLUNK_DB/telegraf_kafka/thaweddb
```

In a Splunk distributed configuration (cluster of indexers), this configuration stands on the cluster master node.

All Splunk searches included in the added refer to the utilisation of a macro called **"telegraf_kafka_index"** configured in:

- telegraf-kafka/default/macros.conf

If you wish to use a different index model, this macro shall be customized to override the default model.

HEC input ingestion and definition

The default recommended way of ingesting the Kafka metrics is using the HTTP Events Collector method which requires the creation of an HEC input.

inputs.conf example:

```
[http://telegraf_kafka_monitoring]
disabled = 0
index = telegraf_kafka
token = 205d43f1-2a31-4e60-a8b3-327eda49944a
```

If you create the HEC input via Splunk Web interface, it is not required to select an explicit value for source and sourcetype.

The HEC input will be ideally relying on a load balancer to provides resiliency and load balancing across your HEC input nodes.

Other ingesting methods

There are other methods possible to ingest the Kafka metrics in Splunk:

- TCP input (graphite format with tags support)
- KAFKA ingestion (Kafka destination from Telegraf in graphite format with tags support, and Splunk connect for Kafka)
- File monitoring with standard Splunk input monitors (file output plugin from Telegraf)

Notes: In the very specific context of monitoring Kafka, it is not a good design to use Kafka as the ingestion method since you will most likely never be able to know when an issue happens on Kafka.

These methods require the deployment of an additional Technology addon: https://splunkbase.splunk.com/app/4193

These methods are heavily described here: https://da-itsi-telegraf-os.readthedocs.io/en/latest/telegraf.html

1.6.2 Monitoring Kafka in dedicated servers (bare metal, VMs)

Dedicated servers are bare metal servers or virtual machines that are dedicated to host one or more Kafka roles.

Monitoring the components metrics with Telegraf

Telegraf installation, configuration and start

If you are running Telegraf as a regular process in machine, the standard installation of Telegraf is really straightforward, consult:

- https://github.com/influxdata/telegraf

If you have a Splunk Universal Forwarder deployment, you can deploy, run and maintain Telegraf and its configuration through a Splunk application (TA), consult:

- https://da-itsi-telegraf-os.readthedocs.io/en/latest/telegraf.html#telegraf-deployment-as-splunk-application-deployed-by-splunk-

An example of a ready to use TA application can be found here:

- https://github.com/guilhemmarchand/TA-telegraf-amd64

For Splunk customers, this solution has various advantages as you can deploy and maintain using your existing Splunk infrastructure.

Telegraf is extremely container friendly, a container approach is very convenient as you can easily run multiple Telegraf containers to monitor each of the Kafka infrastructure components:

- https://hub.docker.com/r/_/telegraf/

Data collection environment design:

The most scalalable and highly available design in term of where placing the Telegraf instances is to deploy Telegraf locally on each server to be monitored (and collect locally the component) or running as a side car container for Kubernetes based environments.

It is to possible to collect multiple instances of multiple components via a unique Telegraf instance, however there will be a limit where issues can start, and this design will not provide high availability as the failure of this instance will impact the whole metric collection.

Telegraf output and minimal configuration

Whether you will be running Telegraf in various containers, or installed as a regular software within the different servers composing your Kafka infrastructure, a minimal configuration is required to teach Telegraf how to forward the metrics to your Splunk deployment.

Telegraf is able to send to data to Splunk in different ways:

- Splunk HTTP Events Collector (HEC) - Since Telegraf v1.8
- Splunk TCP inputs in Graphite format with tags support and the TA for Telegraf
- Apache Kafka topic in Graphite format with tags support and the TA for Telegraf and Splunk connect for Kafka

Who watches for the watcher?

As you are running a Kafka deployment, it would seem very logical to produce metrics in a Kafka topic. However, it presents a specific concern for Kafka itself.

If you use this same system for monitoring Kafka itself, it is very likely that you will never know when Kafka is broken because the data flow for your monitoring system will be broken as well.

The recommendation is to rely either on Splunk HEC or TCP inputs to forward Telegraf metrics data for the Kafka monitoring.

A minimal configuration for telegraf.conf, running in container or as a regular process in machine and forwarding to HEC:

```
[global_tags]
  # the env tag is used by the application for multi-environments management
  env = "my_env"
  # the label tag is an optional tag used by the application that you can use as␣
↪additional label for the services or infrastructure
  label = "my_env_label"

[agent]
  interval = "10s"
  flush_interval = "10s"
  hostname = "$HOSTNAME"

# Regular OS monitoring for Linux OS

# Read metrics about cpu usage
[[inputs.cpu]]
  ## Whether to report per-cpu stats or not
  percpu = true
  ## Whether to report total system cpu stats or not
  totalcpu = true
  ## If true, collect raw CPU time metrics.
  collect_cpu_time = false
```

(continues on next page)

(continued from previous page)

```
  ## If true, compute and report the sum of all non-idle CPU states.
  report_active = false

# Read metrics about disk usage by mount point
[[inputs.disk]]

  ## Ignore mount points by filesystem type.
  ignore_fs = ["tmpfs", "devtmpfs", "devfs"]

# Read metrics about disk IO by device
[[inputs.diskio]]

# Get kernel statistics from /proc/stat
[[inputs.kernel]]

# Read metrics about memory usage
[[inputs.mem]]

# Get the number of processes and group them by status
[[inputs.processes]]

# Read metrics about swap memory usage
[[inputs.swap]]

# Read metrics about system load & uptime
[[inputs.system]]

# # Read metrics about network interface usage
[[inputs.net]]

# # Read TCP metrics such as established, time wait and sockets counts.
[[inputs.netstat]]

# # Monitor process cpu and memory usage
[[inputs.procstat]]
   pattern = ".*"

# outputs
[[outputs.http]]
   url = "https://splunk:8088/services/collector"
   insecure_skip_verify = true
   data_format = "splunkmetric"
    ## Provides time, index, source overrides for the HEC
   splunkmetric_hec_routing = true
    ## Additional HTTP headers
    [outputs.http.headers]
   # Should be set manually to "application/json" for json data_format
      Content-Type = "application/json"
      Authorization = "Splunk 205d43f1-2a31-4e60-a8b3-327eda49944a"
      X-Splunk-Request-Channel = "205d43f1-2a31-4e60-a8b3-327eda49944a"
```

If for some reasons, you have to use either of the 2 other solutions, please consult:

- https://da-itsi-telegraf-os.readthedocs.io/en/latest/telegraf.html

Notes: The configuration above provides out of the box OS monitoring for the hosts, which can be used by the Operating System monitoring application for Splunk:

https://splunkbase.splunk.com/app/4271/

Jolokia JVM monitoring

The following Kafka components require Jolokia to be deployed and started, as the modern and efficient interface to JMX that is collected by Telegraf:

- Apache Zookeeper
- Apache Kafka Brokers
- Apache Kafka Connect
- Confluent schema-registry
- Confluent ksql-server
- Confluent kafka-rest

For the complete documentation of Jolokia, see:

- https://jolokia.org

Jolokia JVM agent can be started in 2 ways, either as using the -javaagent argument during the start of the JVM, or on the fly by attaching Jolokia to the PID ot the JVM:

- https://jolokia.org/reference/html/agents.html#agents-jvm

Starting Jolokia with the JVM

To start Jolokia agent using the -javaagent argument, use such option at the start of the JVM:

```
-javaagent:/opt/jolokia/jolokia.jar=port=8778,host=0.0.0.0
```

Note: This method is the method used in the docker example within this documentation by using the environment variables of the container.

When running on dedicated servers or virtual machines, update the relevant systemd configuration file to start Jolokia automatically:

For Zookeeper

For bare-metals and dedicated VMs:

- Edit: `/lib/systemd/system/confluent-zookeeper.service`
- Add `-javaagent` argument:

```
[Unit]
Description=Apache Kafka - ZooKeeper
Documentation=http://docs.confluent.io/
After=network.target

[Service]
Type=simple
```

(continues on next page)

(continued from previous page)

```
User=cp-kafka
Group=confluent
ExecStart=/usr/bin/zookeeper-server-start /etc/kafka/zookeeper.properties
Environment="KAFKA_OPTS=-javaagent:/opt/jolokia/jolokia.jar=port=8778,host=0.0.0.0"
Environment="LOG_DIR=/var/log/zookeeper"
TimeoutStopSec=180
Restart=no

[Install]
WantedBy=multi-user.target
```

- Reload systemd and restart:

```
sudo systemctl daemon-restart
sudo systemctl restart confluent-zookeeper
```

For container based environments:

Define the following environment variable when starting the containers:

```
KAFKA_OPTS: "-javaagent:/opt/jolokia/jolokia.jar=port=8778,host=0.0.0.0"
```

For Kafka brokers

For bare-metals and dedicated VMs:

- Edit: `/lib/systemd/system/confluent-kafka.service`
- Add `-javaagent` argument:

```
[Unit]
Description=Apache Kafka - broker
Documentation=http://docs.confluent.io/
After=network.target confluent-zookeeper.target

[Service]
Type=simple
User=cp-kafka
Group=confluent
ExecStart=/usr/bin/kafka-server-start /etc/kafka/server.properties
Environment="KAFKA_OPTS=-javaagent:/opt/jolokia/jolokia.jar=port=8778,host=0.0.0.0"
TimeoutStopSec=180
Restart=no

[Install]
WantedBy=multi-user.target
```

- Reload systemd and restart:

```
sudo systemctl daemon-restart
sudo systemctl restart confluent-kafka
```

For container based environments:

Define the following environment variable when starting the containers:

```
KAFKA_OPTS: "-javaagent:/opt/jolokia/jolokia.jar=port=8778,host=0.0.0.0"
```

For Kafka Connect

For bare-metals and dedicated VMs:

- Edit: `/lib/systemd/system/confluent-kafka-connect.service`
- Add `-javaagent` argument:

```
[Unit]
Description=Apache Kafka Connect - distributed
Documentation=http://docs.confluent.io/
After=network.target confluent-kafka.target

[Service]
Type=simple
User=cp-kafka-connect
Group=confluent
ExecStart=/usr/bin/connect-distributed /etc/kafka/connect-distributed.properties
Environment="KAFKA_OPTS=-javaagent:/opt/jolokia/jolokia.jar=port=8778,host=0.0.0.0"
Environment="LOG_DIR=/var/log/connect"
TimeoutStopSec=180
Restart=no

[Install]
WantedBy=multi-user.target
```

- Reload systemd and restart:

```
sudo systemctl daemon-restart
sudo systemctl restart confluent-kafka-connect
```

For container based environments:

Define the following environment variable when starting the containers:

```
KAFKA_OPTS: "-javaagent:/opt/jolokia/jolokia.jar=port=8778,host=0.0.0.0"
```

For Confluent schema-registry

For bare-metals and dedicated VMs:

- Edit: `/lib/systemd/system/confluent-schema-registry.service`
- Add `-javaagent` argument:

```
[Unit]
Description=RESTful Avro schema registry for Apache Kafka
Documentation=http://docs.confluent.io/
After=network.target confluent-kafka.target

[Service]
Type=simple
User=cp-schema-registry
```

(continues on next page)

(continued from previous page)

```
Group=confluent
Environment="LOG_DIR=/var/log/confluent/schema-registry"
Environment="SCHEMA_REGISTRY_OPTS=-javaagent:/opt/jolokia/jolokia.jar=port=8778,
↪host=0.0.0.0"
ExecStart=/usr/bin/schema-registry-start /etc/schema-registry/schema-registry.
↪properties
TimeoutStopSec=180
Restart=no

[Install]
WantedBy=multi-user.target
```

- Reload systemd and restart:

```
sudo systemctl daemon-restart
sudo systemctl restart confluent-schema-registry
```

For container based environments:

Define the following environment variable when starting the containers:

```
SCHEMA_REGISTRY_OPTS: "-javaagent:/opt/jolokia/jolokia.jar=port=8778,host=0.0.0.0"
```

For Confluent ksql-server

For bare-metals and dedicated VMs:

- Edit: `/lib/systemd/system/confluent-ksqldb.service`
- Add `-javaagent` argument:

```
[Unit]
Description=Streaming SQL engine for Apache Kafka
Documentation=http://docs.confluent.io/
After=network.target confluent-kafka.target confluent-schema-registry.target

[Service]
Type=simple
User=cp-ksql
Group=confluent
Environment="LOG_DIR=/var/log/confluent/ksql"
Environment="KSQL_OPTS=-javaagent:/opt/jolokia/jolokia.jar=port=8778,host=0.0.0.0"
ExecStart=/usr/bin/ksql-server-start /etc/ksqldb/ksql-server.properties
TimeoutStopSec=180
Restart=no

[Install]
WantedBy=multi-user.target
```

- Reload systemd and restart:

```
sudo systemctl daemon-restart
sudo systemctl restart confluent-ksqldb
```

For container based environments:

Define the following environment variable when starting the containers:

```
KSQL_OPTS: "-javaagent:/opt/jolokia/jolokia.jar=port=8778,host=0.0.0.0"
```

For Confluent kafka-rest

For bare-metals and dedicated VMs:

- Edit: `/lib/systemd/system/confluent-kafka-rest.service`
- Add `-javaagent` argument:

```
[Unit]
Description=A REST proxy for Apache Kafka
Documentation=http://docs.confluent.io/
After=network.target confluent-kafka.target

[Service]
Type=simple
User=cp-kafka-rest
Group=confluent
Environment="LOG_DIR=/var/log/confluent/kafka-rest"
Environment="KAFKAREST_OPTS=-javaagent:/opt/jolokia/jolokia.jar=port=8778,host=0.0.0.0
↪"


ExecStart=/usr/bin/kafka-rest-start /etc/kafka-rest/kafka-rest.properties
TimeoutStopSec=180
Restart=no

[Install]
WantedBy=multi-user.target
```

- Reload systemd and restart:

```
sudo systemctl daemon-restart
sudo systemctl restart confluent-kafka-rest
```

For container based environments:

Define the following environment variable when starting the containers:

```
KAFKAREST_OPTS: "-javaagent:/opt/jolokia/jolokia.jar=port=8778,host=0.0.0.0"
```

Notes: “KAFKAREST_OPTS” is not a typo, this is the real name of the environment variable for some reason.

Starting Jolokia on the fly

To attach Jolokia agent to an existing JVM, identify its process ID (PID), simplistic example:

```
ps -ef | grep 'kafka.properties' | grep -v grep | awk '{print $1}'
```

Then:

```
java -jar /opt/jolokia/jolokia.jar --host 0.0.0.0 --port 8778 start <PID>
```

Add this operation to any custom init scripts you use to start the Kafka components.

Zookeeper monitoring

Collecting with Telegraf

The Zookeeper monitoring is very simple and achieved by Telegraf and the Zookeeper input plugin.

The following configuration stands in telegraf.conf and configures the input plugin to monitor multiple Zookeeper servers from one source:

```
name_prefix = "zk_"
urls = ["http://zookeeper-1:8778/jolokia","http://zookeeper-2:8778/jolokia","http://
↪zookeeper-3:8778/jolokia"]
```

If each server runs an instance of Zookeeper and you deploy Telegraf, you can simply collect from the localhost:

```
# Zookeeper JVM monitoring
[[inputs.jolokia2_agent]]
  name_prefix = "zk_"
  urls = ["http://$HOSTNAME:8778/jolokia"]
```

Full telegraf.conf example

The following telegraf.conf collects a cluster of 3 Zookeeper servers:

```
[global_tags]
  # the env tag is used by the application for multi-environments management
  env = "my_env"
  # the label tag is an optional tag used by the application that you can use as␣
↪additional label for the services or infrastructure
  label = "my_env_label"

[agent]
  interval = "10s"
  flush_interval = "10s"
  hostname = "$HOSTNAME"

# outputs
[[outputs.http]]
   url = "https://splunk:8088/services/collector"
   insecure_skip_verify = true
   data_format = "splunkmetric"
    ## Provides time, index, source overrides for the HEC
   splunkmetric_hec_routing = true
    ## Additional HTTP headers
    [outputs.http.headers]
   # Should be set manually to "application/json" for json data_format
      Content-Type = "application/json"
      Authorization = "Splunk 205d43f1-2a31-4e60-a8b3-327eda49944a"
      X-Splunk-Request-Channel = "205d43f1-2a31-4e60-a8b3-327eda49944a"

 # Zookeeper JMX collection

 [[inputs.jolokia2_agent]]
   name_prefix = "zk_"
   urls = ["http://zookeeper-1:8778/jolokia","http://zookeeper-2:8778/jolokia","http:/
↪/zookeeper-3:8778/jolokia"]
```

(continues on next page)

(continued from previous page)

```
[[inputs.jolokia2_agent.metric]]
  name  = "quorum"
  mbean = "org.apache.ZooKeeperService:name0=*"
  tag_keys = ["name0"]

[[inputs.jolokia2_agent.metric]]
  name = "leader"
  mbean = "org.apache.ZooKeeperService:name0=*,name1=*,name2=Leader"
  tag_keys = ["name1"]

[[inputs.jolokia2_agent.metric]]
  name = "follower"
  mbean = "org.apache.ZooKeeperService:name0=*,name1=*,name2=Follower"
  tag_keys = ["name1"]
```

Using mcatalog search command to verify data availability:

```
| mcatalog values(metric_name) values(_dims) where index=* metric_name=zk_*
```

Kafka brokers monitoring with Jolokia

Collecting with Telegraf

Depending on how you run Kafka and your architecture preferences, you may prefer to collect all the brokers metrics from one Telegraf collector, or installed locally on the Kafka brocker machine.

Connecting to multiple remote Jolokia instances:

```
# Kafka JVM monitoring
[[inputs.jolokia2_agent]]
  name_prefix = "kafka_"
  urls = ["http://kafka-1:18778/jolokia","http://kafka-2:28778/jolokia","http://kafka-
↪3:38778/jolokia"]
```

Connecting to the local Jolokia instance:

```
# Kafka JVM monitoring
[[inputs.jolokia2_agent]]
  name_prefix = "kafka_"
  urls = ["http://$HOSTNAME:8778/jolokia"]
```

Full telegraf.conf example

The following telegraf.conf collects a cluster of 3 Kafka brokers:

```
[global_tags]
  # the env tag is used by the application for multi-environments management
  env = "my_env"
  # the label tag is an optional tag used by the application that you can use as␣
↪additional label for the services or infrastructure
  label = "my_env_label"
```

(continues on next page)

(continued from previous page)

```
[agent]
  interval = "10s"
  flush_interval = "10s"
  hostname = "$HOSTNAME"

# outputs
[[outputs.http]]
   url = "https://splunk:8088/services/collector"
   insecure_skip_verify = true
   data_format = "splunkmetric"
    ## Provides time, index, source overrides for the HEC
   splunkmetric_hec_routing = true
    ## Additional HTTP headers
    [outputs.http.headers]
   # Should be set manually to "application/json" for json data_format
      Content-Type = "application/json"
      Authorization = "Splunk 205d43f1-2a31-4e60-a8b3-327eda49944a"
      X-Splunk-Request-Channel = "205d43f1-2a31-4e60-a8b3-327eda49944a"

# Kafka JVM monitoring

[[inputs.jolokia2_agent]]
  name_prefix = "kafka_"
  urls = ["http://kafka-1:18778/jolokia","http://kafka-2:28778/jolokia","http://kafka-
↪3:38778/jolokia"]

[[inputs.jolokia2_agent.metric]]
  name         = "controller"
  mbean        = "kafka.controller:name=*,type=*"
  field_prefix = "$1."

[[inputs.jolokia2_agent.metric]]
  name         = "replica_manager"
  mbean        = "kafka.server:name=*,type=ReplicaManager"
  field_prefix = "$1."

[[inputs.jolokia2_agent.metric]]
  name         = "purgatory"
  mbean        = "kafka.server:delayedOperation=*,name=*,
↪type=DelayedOperationPurgatory"
  field_prefix = "$1."
  field_name   = "$2"

[[inputs.jolokia2_agent.metric]]
  name     = "client"
  mbean    = "kafka.server:client-id=*,type=*"
  tag_keys = ["client-id", "type"]

[[inputs.jolokia2_agent.metric]]
  name         = "network"
  mbean        = "kafka.network:name=*,request=*,type=RequestMetrics"
  field_prefix = "$1."
  tag_keys     = ["request"]

[[inputs.jolokia2_agent.metric]]
  name         = "network"
  mbean        = "kafka.network:name=ResponseQueueSize,type=RequestChannel"
```

(continues on next page)

(continued from previous page)

```
  field_prefix = "ResponseQueueSize"
  tag_keys     = ["name"]

[[inputs.jolokia2_agent.metric]]
  name         = "network"
  mbean        = "kafka.network:name=NetworkProcessorAvgIdlePercent,type=SocketServer"
  field_prefix = "NetworkProcessorAvgIdlePercent"
  tag_keys     = ["name"]

[[inputs.jolokia2_agent.metric]]
  name         = "topics"
  mbean        = "kafka.server:name=*,type=BrokerTopicMetrics"
  field_prefix = "$1."

[[inputs.jolokia2_agent.metric]]
  name         = "topic"
  mbean        = "kafka.server:name=*,topic=*,type=BrokerTopicMetrics"
  field_prefix = "$1."
  tag_keys     = ["topic"]

[[inputs.jolokia2_agent.metric]]
  name       = "partition"
  mbean      = "kafka.log:name=*,partition=*,topic=*,type=Log"
  field_name = "$1"
  tag_keys   = ["topic", "partition"]

[[inputs.jolokia2_agent.metric]]
  name       = "log"
  mbean      = "kafka.log:name=LogFlushRateAndTimeMs,type=LogFlushStats"
  field_name = "LogFlushRateAndTimeMs"
  tag_keys   = ["name"]

[[inputs.jolokia2_agent.metric]]
  name       = "partition"
  mbean      = "kafka.cluster:name=UnderReplicated,partition=*,topic=*,type=Partition"
  field_name = "UnderReplicatedPartitions"
  tag_keys   = ["topic", "partition"]

[[inputs.jolokia2_agent.metric]]
  name     = "request_handlers"
  mbean    = "kafka.server:name=RequestHandlerAvgIdlePercent,
↪type=KafkaRequestHandlerPool"
  tag_keys = ["name"]

# JVM garbage collector monitoring
[[inputs.jolokia2_agent.metric]]
  name     = "jvm_garbage_collector"
  mbean    = "java.lang:name=*,type=GarbageCollector"
  paths    = ["CollectionTime", "CollectionCount", "LastGcInfo"]
  tag_keys = ["name"]
```

Using mcatalog search command to verify data availability:

```
| mcatalog values(metric_name) values(_dims) where index=* metric_name=kafka_*.*
```

Kafka connect monitoring

Collecting with Telegraf

Connecting to multiple remote Jolokia instances:

```
# Kafka-connect JVM monitoring
[[inputs.jolokia2_agent]]
  name_prefix = "kafka_connect."
  urls = ["http://kafka-connect-1:18779/jolokia","http://kafka-connect-2:28779/jolokia
↪","http://kafka-connect-3:38779/jolokia"]
```

Connecting to local Jolokia instance:

```
# Kafka-connect JVM monitoring
 [[inputs.jolokia2_agent]]
   name_prefix = "kafka_connect."
   urls = ["http://$HOSTNAME:8778/jolokia"]
```

Full telegraf.conf example

bellow a full telegraf.conf example:

```
[global_tags]
  # the env tag is used by the application for multi-environments management
  env = "my_env"
  # the label tag is an optional tag used by the application that you can use as␣
↪additional label for the services or infrastructure
  label = "my_env_label"

[agent]
  interval = "10s"
  flush_interval = "10s"
  hostname = "$HOSTNAME"

# outputs
[[outputs.http]]
   url = "https://splunk:8088/services/collector"
   insecure_skip_verify = true
   data_format = "splunkmetric"
    ## Provides time, index, source overrides for the HEC
   splunkmetric_hec_routing = true
    ## Additional HTTP headers
    [outputs.http.headers]
   # Should be set manually to "application/json" for json data_format
      Content-Type = "application/json"
      Authorization = "Splunk 205d43f1-2a31-4e60-a8b3-327eda49944a"
      X-Splunk-Request-Channel = "205d43f1-2a31-4e60-a8b3-327eda49944a"

# Kafka-connect JVM monitoring

[[inputs.jolokia2_agent]]
  name_prefix = "kafka_connect."
  urls = ["http://kafka-connect-1:18779/jolokia","http://kafka-connect-2:28779/jolokia
↪","http://kafka-connect-3:38779/jolokia"]
```

(continues on next page)

(continued from previous page)

```
[[inputs.jolokia2_agent.metric]]
  name         = "worker"
  mbean        = "kafka.connect:type=connect-worker-metrics"

[[inputs.jolokia2_agent.metric]]
  name         = "worker"
  mbean        = "kafka.connect:type=connect-worker-rebalance-metrics"

[[inputs.jolokia2_agent.metric]]
  name         = "connector-task"
  mbean        = "kafka.connect:type=connector-task-metrics,connector=*,task=*"
  tag_keys = ["connector", "task"]

[[inputs.jolokia2_agent.metric]]
  name         = "sink-task"
  mbean        = "kafka.connect:type=sink-task-metrics,connector=*,task=*"
  tag_keys = ["connector", "task"]

[[inputs.jolokia2_agent.metric]]
  name         = "source-task"
  mbean        = "kafka.connect:type=source-task-metrics,connector=*,task=*"
  tag_keys = ["connector", "task"]

[[inputs.jolokia2_agent.metric]]
  name         = "error-task"
  mbean        = "kafka.connect:type=task-error-metrics,connector=*,task=*"
  tag_keys = ["connector", "task"]

# Kafka connect return a status value which is non numerical
# Using the enum processor with the following configuration replaces the string value␣
↪by our mapping
[[processors.enum]]
  [[processors.enum.mapping]]
    ## Name of the field to map
    field = "status"

    ## Table of mappings
    [processors.enum.mapping.value_mappings]
      paused = 0
      running = 1
      unassigned = 2
      failed = 3
      destroyed = 4
```

Using mcatalog search command to verify data availability:

```
| mcatalog values(metric_name) values(_dims) where index=* metric_name=kafka_connect.*
```

Kafka LinkedIn monitor - end to end monitoring

Installing and starting the Kafka monitor

LinkedIn provides an extremely powerful open source end to end monitoring solution for Kafka, please consult:

- https://github.com/linkedin/kafka-monitor

As a builtin configuration, the kafka-monitor implements a jolokia agent, so collecting the metrics with Telegraf cannot be more easy !

It is very straightforward to run the kafka-monitor in a docker container, first you need to create your own image:

- https://github.com/linkedin/kafka-monitor/tree/master/docker

In a nutshell, you would:

```
git clone https://github.com/linkedin/kafka-monitor.git
cd kafka-monitor
./gradlew jar
cd docker
```

Edit the Makefile to match your needs

```
make container
make push
```

Then start your container, example with docker-compose:

```
kafka-monitor:
image: guilhemmarchand/kafka-monitor:2.0.3
hostname: kafka-monitor
volumes:
  - ../kafka-monitor:/usr/local/share/kafka-monitor
command: "/opt/kafka-monitor/bin/kafka-monitor-start.sh /usr/local/share/kafka-
↪monitor/kafka-monitor.properties"
```

Once your Kafka monitor is running, you need a Telegraf instance that will be collecting the JMX beans, example:

```
[global_tags]
  # the env tag is used by the application for multi-environments management
  env = "my_env"
  # the label tag is an optional tag used by the application that you can use as␣
↪additional label for the services or infrastructure
  label = "my_env_label"

[agent]
  interval = "10s"
  flush_interval = "10s"
  hostname = "$HOSTNAME"

# outputs
[[outputs.http]]
   url = "https://splunk:8088/services/collector"
   insecure_skip_verify = true
   data_format = "splunkmetric"
    ## Provides time, index, source overrides for the HEC
   splunkmetric_hec_routing = true
    ## Additional HTTP headers
    [outputs.http.headers]
   # Should be set manually to "application/json" for json data_format
      Content-Type = "application/json"
      Authorization = "Splunk 205d43f1-2a31-4e60-a8b3-327eda49944a"
      X-Splunk-Request-Channel = "205d43f1-2a31-4e60-a8b3-327eda49944a"
```

(continues on next page)

(continued from previous page)

```
# Kafka JVM monitoring

[[inputs.jolokia2_agent]]
  name_prefix = "kafka_"
  urls = ["http://kafka-monitor:8778/jolokia"]

[[inputs.jolokia2_agent.metric]]
  name         = "kafka-monitor"
  mbean        = "kmf.services:name=*,type=*"
```

Using mcatalog search command to verify data availability:

```
| mcatalog values(metric_name) values(_dims) where index=* metric_name=kafka_kafka-
↪monitor.*
```

Confluent schema-registry

Collecting with Telegraf

Connecting to multiple remote Jolokia instances:

```
[[inputs.jolokia2_agent]]
  name_prefix = "kafka_schema-registry."
  urls = ["http://schema-registry:18783/jolokia"]
```

Connecting to local Jolokia instance:

```
# Kafka-connect JVM monitoring
 [[inputs.jolokia2_agent]]
  name_prefix = "kafka_schema-registry."
   urls = ["http://$HOSTNAME:8778/jolokia"]
```

Full telegraf.conf example

bellow a full telegraf.conf example:

```
[global_tags]
  # the env tag is used by the application for multi-environments management
  env = "my_env"
  # the label tag is an optional tag used by the application that you can use as␣
↪additional label for the services or infrastructure
  label = "my_env_label"

[agent]
  interval = "10s"
  flush_interval = "10s"
  hostname = "$HOSTNAME"

# outputs
[[outputs.http]]
   url = "https://splunk:8088/services/collector"
   insecure_skip_verify = true
   data_format = "splunkmetric"
```

(continues on next page)

(continued from previous page)

```
    ## Provides time, index, source overrides for the HEC
   splunkmetric_hec_routing = true
    ## Additional HTTP headers
    [outputs.http.headers]
   # Should be set manually to "application/json" for json data_format
      Content-Type = "application/json"
      Authorization = "Splunk 205d43f1-2a31-4e60-a8b3-327eda49944a"
      X-Splunk-Request-Channel = "205d43f1-2a31-4e60-a8b3-327eda49944a"

# schema-registry JVM monitoring

[[inputs.jolokia2_agent]]
  name_prefix = "kafka_schema-registry."
  urls = ["http://schema-registry:18783/jolokia"]

[[inputs.jolokia2_agent.metric]]
  name         = "jetty-metrics"
  mbean        = "kafka.schema.registry:type=jetty-metrics"
  paths = ["connections-active", "connections-opened-rate", "connections-closed-rate"]

[[inputs.jolokia2_agent.metric]]
  name         = "master-slave-role"
  mbean        = "kafka.schema.registry:type=master-slave-role"

[[inputs.jolokia2_agent.metric]]
  name         = "jersey-metrics"
  mbean        = "kafka.schema.registry:type=jersey-metrics"
```

Using mcatalog search command to verify data availability:

```
| mcatalog values(metric_name) values(_dims) where index=* metric_name=kafka_schema-
↪registry.*
```

Confluent ksql-server

Collecting with Telegraf

Connecting to multiple remote Jolokia instances:

```
[[inputs.jolokia2_agent]]
  name_prefix = "kafka_"
  urls = ["http://ksql-server-1:18784/jolokia"]
```

Connecting to local Jolokia instance:

```
[[inputs.jolokia2_agent]]
  name_prefix = "kafka_"
  urls = ["http://$HOSTNAME:18784/jolokia"]
```

Full telegraf.conf example

bellow a full telegraf.conf example:

```
[global_tags]
  # the env tag is used by the application for multi-environments management
  env = "my_env"
  # the label tag is an optional tag used by the application that you can use as␣
↪additional label for the services or infrastructure
  label = "my_env_label"

[agent]
  interval = "10s"
  flush_interval = "10s"
  hostname = "$HOSTNAME"

# outputs
[[outputs.http]]
   url = "https://splunk:8088/services/collector"
   insecure_skip_verify = true
   data_format = "splunkmetric"
    ## Provides time, index, source overrides for the HEC
   splunkmetric_hec_routing = true
    ## Additional HTTP headers
    [outputs.http.headers]
   # Should be set manually to "application/json" for json data_format
      Content-Type = "application/json"
      Authorization = "Splunk 205d43f1-2a31-4e60-a8b3-327eda49944a"
      X-Splunk-Request-Channel = "205d43f1-2a31-4e60-a8b3-327eda49944a"

# ksql-server JVM monitoring

 [[inputs.jolokia2_agent]]
   name_prefix = "kafka_"
   urls = ["http://ksql-server:18784/jolokia"]

 [[inputs.jolokia2_agent.metric]]
   name         = "ksql-server"
   mbean        = "io.confluent.ksql.metrics:type=*"
```

Using mcatalog search command to verify data availability:

```
| mcatalog values(metric_name) values(_dims) where index=* metric_name=kafka_ksql-
↪server.*
```

Confluent kafka-rest

Collecting with Telegraf

Connecting to multiple remote Jolokia instances:

```
[[inputs.jolokia2_agent]]
  name_prefix = "kafka_kafka-rest."
  urls = ["http://kafka-rest:8778/jolokia"]
```

Connecting to local Jolokia instance:

```
[[inputs.jolokia2_agent]]
  name_prefix = "kafka_kafka-rest."
  urls = ["http://$HOSTNAME:18785/jolokia"]
```

Full telegraf.conf example

bellow a full telegraf.conf example:

```
[global_tags]
  # the env tag is used by the application for multi-environments management
  env = "my_env"
  # the label tag is an optional tag used by the application that you can use as␣
↪additional label for the services or infrastructure
  label = "my_env_label"

[agent]
  interval = "10s"
  flush_interval = "10s"
  hostname = "$HOSTNAME"

# outputs
[[outputs.http]]
   url = "https://splunk:8088/services/collector"
   insecure_skip_verify = true
   data_format = "splunkmetric"
    ## Provides time, index, source overrides for the HEC
   splunkmetric_hec_routing = true
    ## Additional HTTP headers
    [outputs.http.headers]
   # Should be set manually to "application/json" for json data_format
      Content-Type = "application/json"
      Authorization = "Splunk 205d43f1-2a31-4e60-a8b3-327eda49944a"
      X-Splunk-Request-Channel = "205d43f1-2a31-4e60-a8b3-327eda49944a"

 # kafka-rest JVM monitoring

 [[inputs.jolokia2_agent]]
   name_prefix = "kafka_kafka-rest."
   urls = ["http://kafka-rest:18785/jolokia"]

 [[inputs.jolokia2_agent.metric]]
   name         = "jetty-metrics"
   mbean        = "kafka.rest:type=jetty-metrics"
   paths = ["connections-active", "connections-opened-rate", "connections-closed-rate
↪"]

 [[inputs.jolokia2_agent.metric]]
   name         = "jersey-metrics"
   mbean        = "kafka.rest:type=jersey-metrics"
```

Using mcatalog search command to verify data availability:

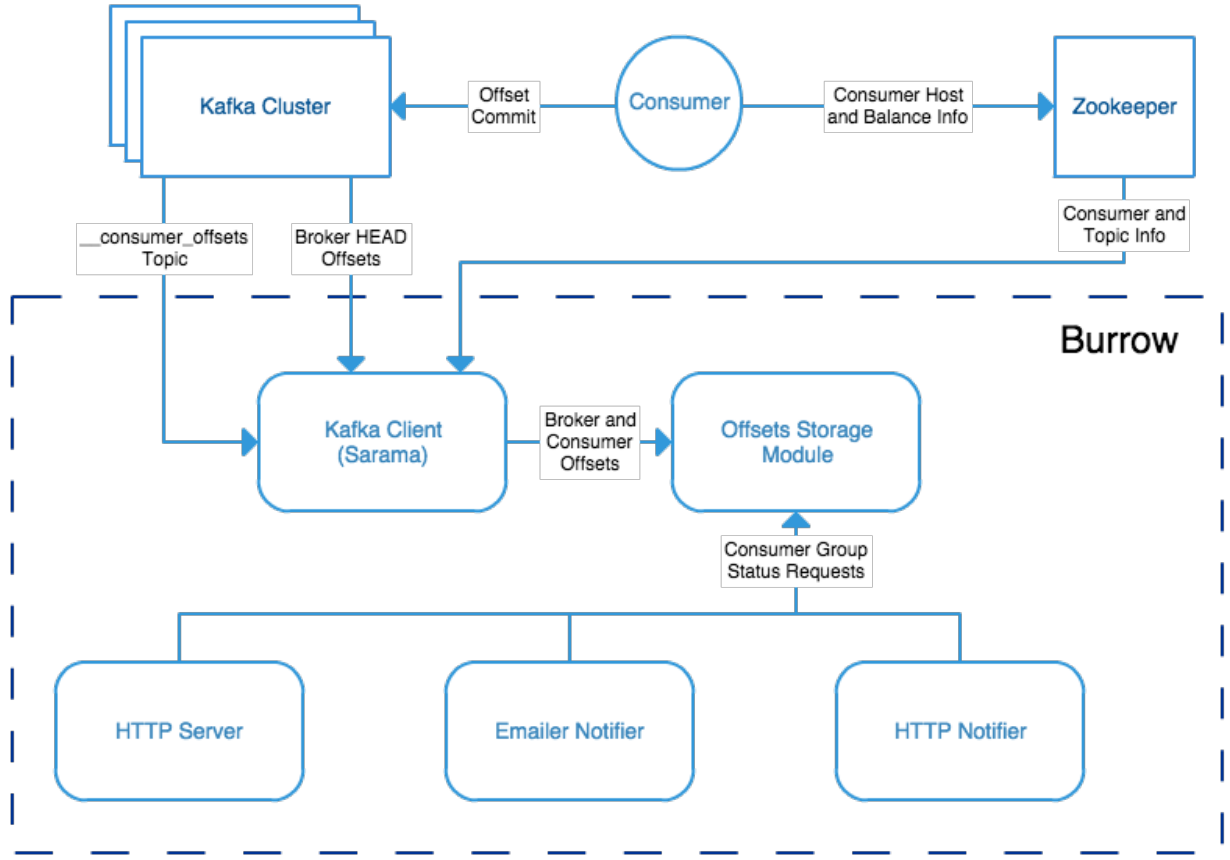
```
| mcatalog values(metric_name) values(_dims) where index=* metric_name=kafka_kafka_
↪kafka-rest.*
```

Burrow Lag Consumers

As from their authors, Burrow is a monitoring companion for Apache Kafka that provides consumer lag checking as a service without the need for specifying thresholds.

See: https://github.com/linkedin/Burrow

Burrow workflow diagram:



Burrow is a very powerful application that monitors all consumers (Kafka Connect connectors, Kafka Streams...) to report an advanced state of the service automatically, and various useful lagging metrics.

Telegraf has a native input for Burrow which polls consumers, topics and partitions lag metrics and statuses over http, use the following telegraf minimal configuration:

See: https://github.com/influxdata/telegraf/tree/master/plugins/inputs/burrow

```
[global_tags]
  # the env tag is used by the application for multi-environments management
  env = "my_env"
  # the label tag is an optional tag used by the application that you can use as␣
↪additional label for the services or infrastructure
  label = "my_env_label"

[agent]
  interval = "10s"
  flush_interval = "10s"
  hostname = "$HOSTNAME"

# outputs
[[outputs.http]]
   url = "https://splunk:8088/services/collector"
   insecure_skip_verify = true
   data_format = "splunkmetric"
    ## Provides time, index, source overrides for the HEC
```

(continues on next page)

(continued from previous page)

```
   splunkmetric_hec_routing = true
    ## Additional HTTP headers
    [outputs.http.headers]
   # Should be set manually to "application/json" for json data_format
      Content-Type = "application/json"
      Authorization = "Splunk 205d43f1-2a31-4e60-a8b3-327eda49944a"
      X-Splunk-Request-Channel = "205d43f1-2a31-4e60-a8b3-327eda49944a"

# Burrow

[[inputs.burrow]]
  ## Burrow API endpoints in format "schema://host:port".
  ## Default is "http://localhost:8000".
  servers = ["http://dockerhost:9001"]

  ## Override Burrow API prefix.
  ## Useful when Burrow is behind reverse-proxy.
  # api_prefix = "/v3/kafka"

  ## Maximum time to receive response.
  # response_timeout = "5s"

  ## Limit per-server concurrent connections.
  ## Useful in case of large number of topics or consumer groups.
  # concurrent_connections = 20

  ## Filter clusters, default is no filtering.
  ## Values can be specified as glob patterns.
  # clusters_include = []
  # clusters_exclude = []

  ## Filter consumer groups, default is no filtering.
  ## Values can be specified as glob patterns.
  # groups_include = []
  # groups_exclude = []

  ## Filter topics, default is no filtering.
  ## Values can be specified as glob patterns.
  # topics_include = []
  # topics_exclude = []

  ## Credentials for basic HTTP authentication.
  # username = ""
  # password = ""

  ## Optional SSL config
  # ssl_ca = "/etc/telegraf/ca.pem"
  # ssl_cert = "/etc/telegraf/cert.pem"
  # ssl_key = "/etc/telegraf/key.pem"
  # insecure_skip_verify = false
```

Using mcatalog search command to verify data availability:

```
| mcatalog values(metric_name) values(_dims) where index=* metric_name=burrow_*
```

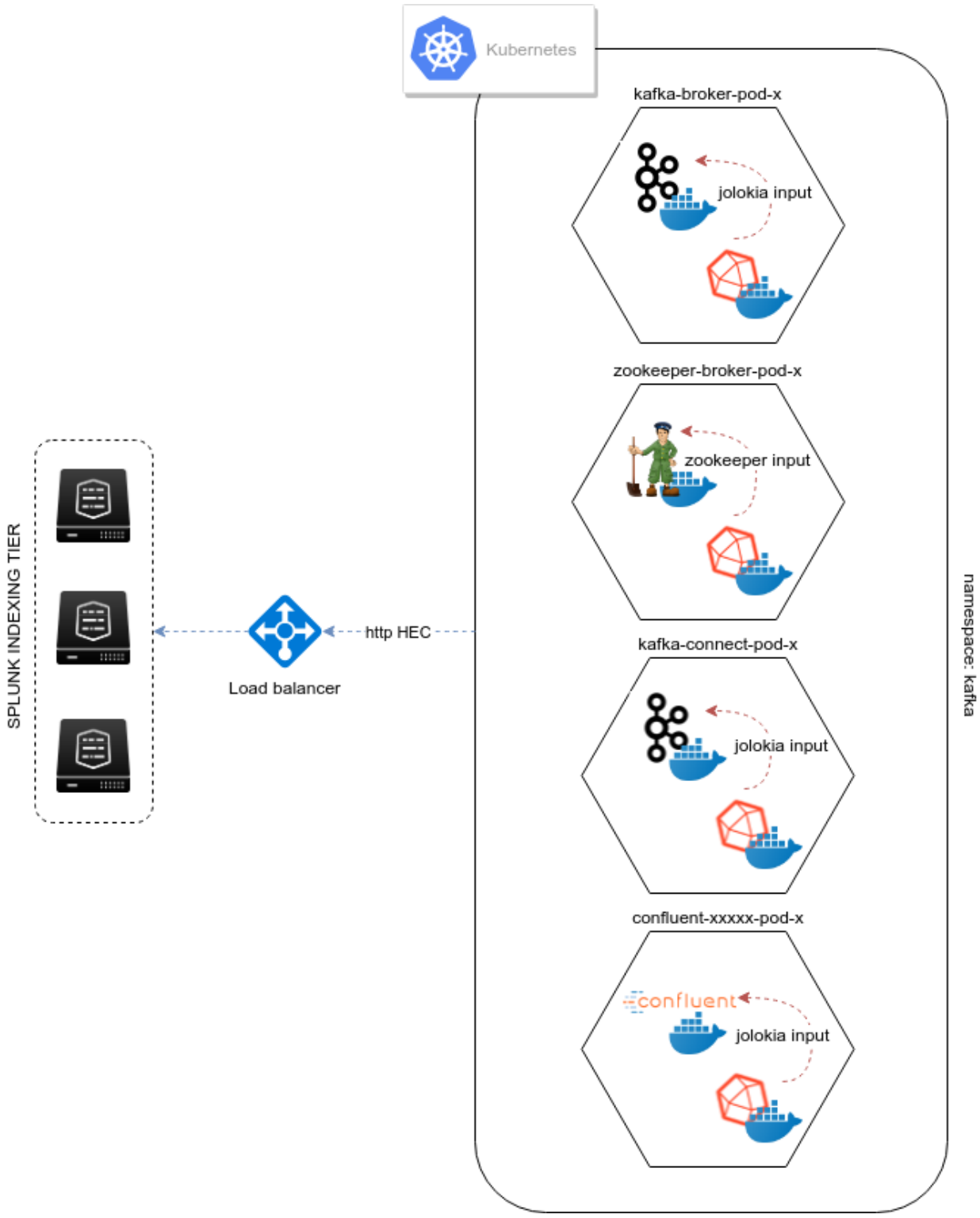
1.6.3 Monitoring Kafka in Kubernetes

For the ease of documentation, this guide assumes you are deploying containers with Kubernetes and Docker, although these instructions can transposed to other containers orchestrator solutions.

3 main steps for implementation:

1. Deploying Jolokia jar agent
2. Configuring the containers to start with Jolokia
3. Deploying the Telegraf containers

metrics collection diagram - sidecar containers:


Kubernetes
kafka-broker-pod-x
jolokia input
zookeeper-broker-pod-x
zookeeper input
kafka-connect-pod-x
jolokia input
confluent-xxxxx-pod-x
confluent
jolokia input
namespace: kafka
SPLUNK INDEXING TIER
Load balancer
http HEC

Deploying Jolokia

The Jolokia agent jar file needs to be available to the pods, you have different possibilities:

- Starting Kubernetes 1.10.0, you can store a binary file in a configMap. As such, it is very easy to load the Jolokia jar file and make it available to your pods. (**recommended approach**)
- For prior versions, you can automatically mount a persistent volume on the pods such as an NFS volume or a Cloud provider volume that will make the Jolokia jar available to your pods.
- uploading the jar file on every node and mounting a local persistent volume (requires each node to have the jolokia jar uploaded manually)

To download the latest version of Jolokia: https://jolokia.org/reference/html/agents.html#agents-jvm

Option 1: Jolokia jar in configMap

See the files in Github:

https://github.com/guilhemmarchand/splunk-guide-for-kafka-monitoring/tree/master/kubernetes-yaml-examples/Jolokia

From your management server where kubectl is configured, download the latest Jolokia jar file:

```
curl http://search.maven.org/remotecontent?filepath=org/jolokia/jolokia-jvm/1.6.0/
↪jolokia-jvm-1.6.0-agent.jar -o jolokia.jar
```

Create a configMap from the binary file:

```
kubectl create configmap jolokia-jar --from-file=jolokia.jar
```

From the configMap, optionally create the yml file:

```
kubectl get configmaps jolokia-jar -o yaml --export > 01-jolokia-jar-configmap.yml
```

If you need your configMap to be associated with a name space, simply edit the end of the file and add your name space Metadata:

```
metadata:
  name: jolokia-jar
  namespace: kafka
```

Modify your definitions to include the volume:

```
spec:
  volumes:
    - name: jolokia-jar
      configMap:
        name: jolokia-jar
  containers:
    - name: xxxxx
      image: xxxx
      volumeMounts:
        - mountPath: "/opt/jolokia"
          name: jolokia-jar
```

Finally, update the environment variable to start Jolokia (see next steps) and apply.

Option 2: NFS persistent volume configuration example

Ensure all the nodes have the nfs-common package installed:

For Ubuntu & Debian:

```
sudo apt-get -y install nfs-common
```

For RHEL, Centos and derivated:

```
sudo yum -y install nfs-common
```

Upload the jar file to your NFS server, and create a share that will be used automatically by the pods, example:

```
/export/jolokia/jolokia-jvm-1.6.0-agent.jar
```

Have your share configured in /etc/exports:

```
/export/jolokia/ *(ro,sync,no_root_squash,subtree_check)
```

Refresh exports:

```
sudo exportfs -ra
```

Create a Kubernetes PersistentVolume:

pv-jolokia.yaml

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: pv-jolokia
  labels:
    type: jolokia
spec:
  storageClassName: generic
  capacity:
    storage: 100Mi
  accessModes:
    - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  nfs:
    path: /export/jolokia
    server: <NFS server address>
    readOnly: true
```

*pvc-jolokia.yaml:**

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-jolokia
spec:
  storageClassName: generic
  accessModes:
```

(continues on next page)

(continued from previous page)

```
  - ReadOnlyMany
  resources:
    requests:
      storage: 100Mi
  selector:
    matchLabels:
      type: jolokia
```

When you will start your pods, you will specify the PersistentVolumeClaim and the mount options to get Jolokia available on the pods:

```
kind: Pod
apiVersion: v1
metadata:
  name: xxxxx
spec:
  volumes:
    - name: pv-jolokia
      persistentVolumeClaim:
       claimName: pvc-jolokia
  containers:
    - name: xxxxx
      image: xxxx
      volumeMounts:
        - mountPath: "/opt/jolokia"
          name: pv-jolokia
```

Option 3: Local persistent volume configuration example

Upload the jar file to each of Kubernetes node, this documentation assumes the agent will be available in /opt/jolokia/, example:

```
/opt/jolokia/jolokia-jvm-1.6.0-agent.jar
```

Create a Kubernetes PersistentVolume:

pv-jolokia.yaml

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: pv-jolokia
  labels:
    type: jolokia
spec:
  storageClassName: generic
  capacity:
    storage: 100Mi
  accessModes:
    - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: "/opt/jolokia"
```

Create:

```
kubectl create -f pv-jolokia.yaml
```

Create a PersistentVolumeClaim to be used by the pods definition:

*pvc-jolokia.yaml:**

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-jolokia
spec:
  storageClassName: generic
  accessModes:
  - ReadOnlyMany
  resources:
    requests:
      storage: 100Mi
  selector:
    matchLabels:
      type: jolokia
```

When you will start your pods, you will specify the PersistentVolumeClaim and the mount options to get Jolokia available on the pods:

```
kind: Pod
apiVersion: v1
metadata:
  name: xxxxx
spec:
  volumes:
    - name: pv-jolokia
      persistentVolumeClaim:
       claimName: pvc-jolokia
  containers:
    - name: xxxxx
      image: xxxx
      volumeMounts:
        - mountPath: "/opt/jolokia"
          name: pv-jolokia
```

Starting Jolokia with container startup

Kafka brokers

Modify your pod definition:

```
spec:
  containers:
  - name: xxxxxx
    image: xxxxxx:latest
    env:
    - name: KAFKA_OPTS
      value: "-javaagent:/opt/jolokia/jolokia.jar=port=8778,host=0.0.0.0"
```

You can use the patch method to update your existing pod definition:

https://github.com/guilhemmarchand/splunk-guide-for-kafka-monitoring/tree/master/kubernetes-yaml-examples/zookeeper

Kafka Connect

Modify your pod definition:

```
spec:
  containers:
  - name: xxxxxx
    image: xxxxxx:latest
    env:
    - name: KAFKA_OPTS
      value: "-javaagent:/opt/jolokia/jolokia.jar=port=8778,host=0.0.0.0"
```

You can use the patch method to update your existing pod definition:

https://github.com/guilhemmarchand/splunk-guide-for-kafka-monitoring/tree/master/kubernetes-yaml-examples/kafka-connect

Schema registry

Modify your pod definition:

```
spec:
  containers:
  - name: xxxxxx
    image: xxxxxx:latest
    env:
    - name: SCHEMA_REGISTRY_OPTS
      value: "-javaagent:/opt/jolokia/jolokia.jar=port=8778,host=0.0.0.0"
```

You can use the patch method to update your existing pod definition:

https://github.com/guilhemmarchand/splunk-guide-for-kafka-monitoring/tree/master/kubernetes-yaml-examples/confluent-schema-registry

ksql-server

Modify your pod definition:

```
spec:
  containers:
  - name: xxxxxx
    image: xxxxxx:latest
    env:
    - name: KSQL_OPTS
      value: "-javaagent:/opt/jolokia/jolokia.jar=port=8778,host=0.0.0.0"
```

You can use the patch method to update your existing pod definition:

https://github.com/guilhemmarchand/splunk-guide-for-kafka-monitoring/tree/master/kubernetes-yaml-examples/confluent-ksql-server

kafka-rest

Modify your pod definition:

```
spec:
  containers:
  - name: xxxxxx
    image: xxxxxx:latest
    env:
    - name: KAFKAREST_OPTS
      value: "-javaagent:/opt/jolokia/jolokia.jar=port=8778,host=0.0.0.0"
```

You can use the patch method to update your existing pod definition:

https://github.com/guilhemmarchand/splunk-guide-for-kafka-monitoring/tree/master/kubernetes-yaml-examples/confluent-kafka-rest

Monitoring the components metrics with Telegraf

Telegraf is a very efficient plugin driven agent collector, in the context of Kubernetes there are several design choices possible:

- Running Telegraf agent as a container in the same pod than the JVM container, called a sidecar container. (recommended approach)
- Running Telegraf agent as a deployment with 1 replica, accessing all JVMs instances via cluster exposed services (one or more deployments if you want to specialise per role, or something else)

Both designs are pertinents, however running collector agents as sidecar containers provides valuable advantages such as ensuring that the collector container will always run on the same node and it is not required to expose any endpoint.

In addition, this is an easy "build and forget" approach, each container monitors the local JVM container automatically, following the same rhythm of destruction and creation.

When running Telegraf as a sidecar container, an additional container will be running in the same pod, generally associated with a StatefulSet or Deployment.

Zookeeper monitoring

Link: Zookeeper metrics

Kafka Brokers monitoring

Link: Kafka Brokers metrics

Kafka Connect monitoring

Link: Kafka Connect metrics

Confluent schema-registry monitoring

Link: Confluent shema-registry metrics

Confluent kafka-rest monitoring

Link: Confluent kafka-rest metrics

Confluent ksql-server monitoring

Link: Confluent ksql-server metrics

1.7 Chapter 3: Alerting

1.7.1 Monitoring your Kafka infrastructure with Splunk and the application Telegraf for Kafka

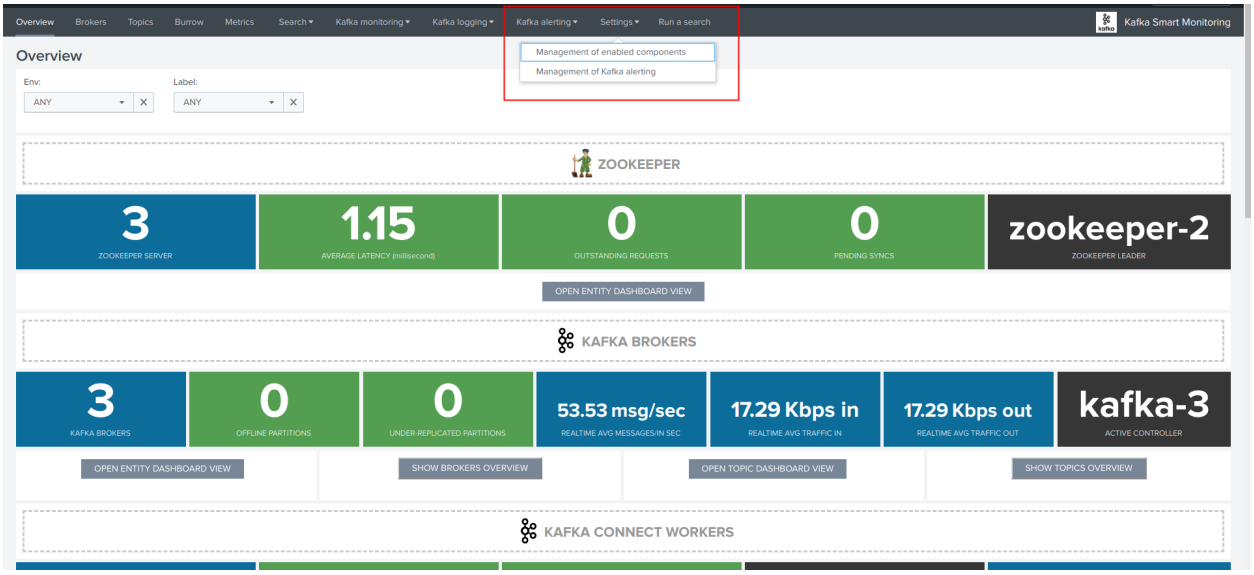
The Splunk application for Kafka monitoring with Telegraf is available in Splunk Base:

https://splunkbase.splunk.com/app/4268

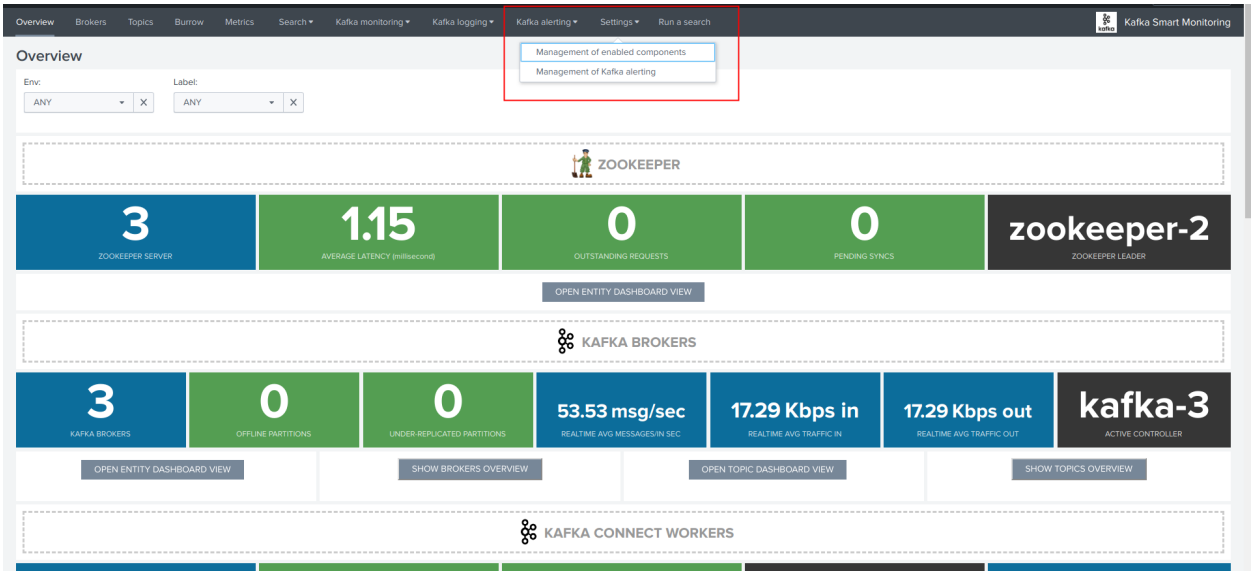
The dedicated documentation Web site is available here:

https://telegraf-kafka.readthedocs.io

Go straight to the Kafka alerting in app menu:

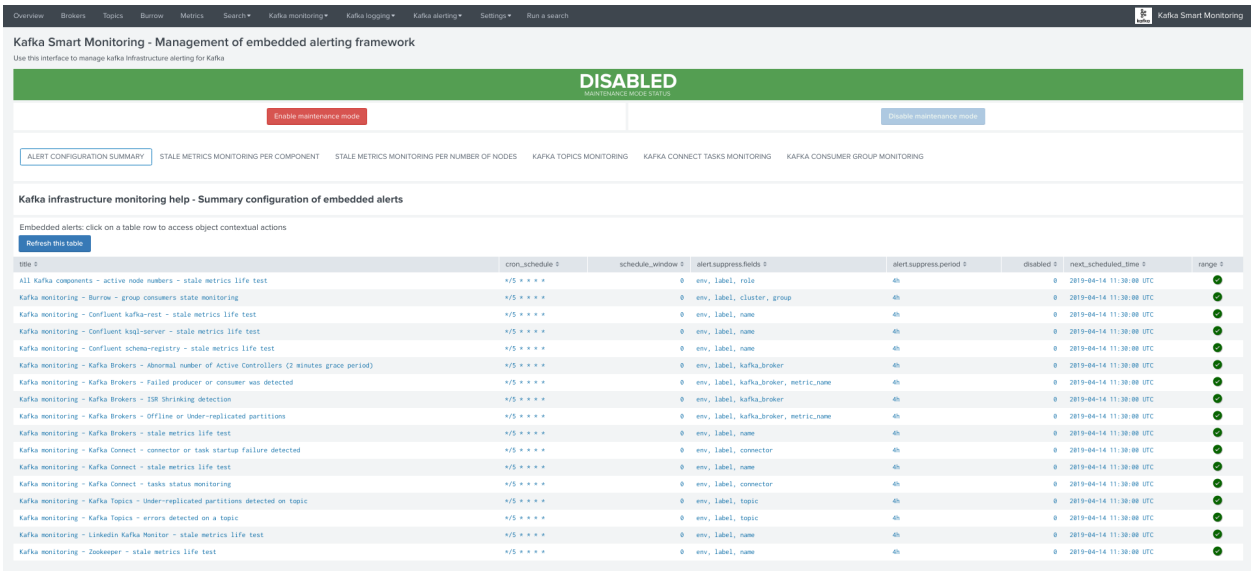


Go to app menu / Settings / Management of Kafka alerting:



Management of Kafka alerting (user interface)

The OOTB alerting model relies on several KVstore collections being automatically populated, the user interface "Management of Kafka alerting" allows you to interact easily with different aspects of the monitoring:



KVstore collections and lookup definitions

The alerting framework relies on several KVstore collections and associated lookup definitions:

Purpose	KVstore collection	Lookup definition
Monitoring per component entity	kv_telegraf_kafka_inventory	kafka_infra_inventory
Monitoring per nodes number	kv_kafka_infra_nodes_inventory	kafka_infra_nodes_inventory
Monitoring of Kafka topics	kv_telegraf_kafka_topics_monitoring	kafka_topics_monitoring
Monitoring per component entity	kv_kafka_connect_tasks_monitoring	kafka_connect_tasks_monitoring
Monitoring per Burrow consumers	kv_kafka_burrow_consumers_monitoring	kafka_burrow_consumers_monitoring
Maintenance mode management	kv_kafka_alerting_maintenance	kafka_alerting_maintenance

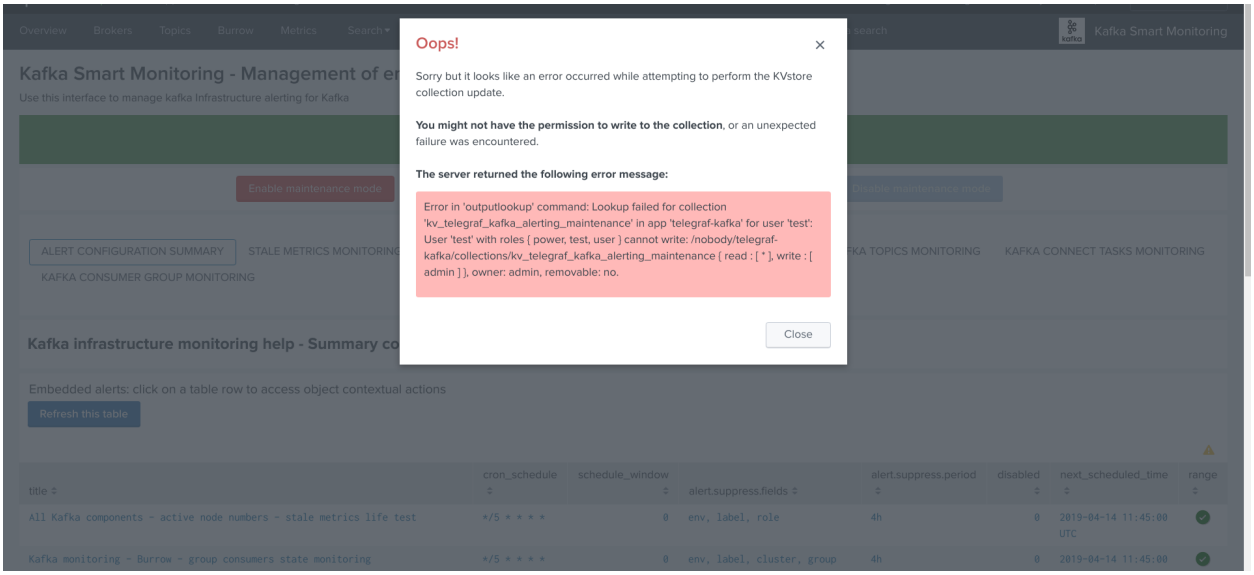
Permissions and authorizations

Managing the alerting framework and its objects require KVstore collection and lookup definition write permissions.

You can rely on the builtin role **kafka_admin** and configure your Kafka administrators to be member of the role.

The role provides the level of permissions required to administrate the KVstore collections.

Shall an unauthorized user attempt to perform an operation, or access to an object that is no readable, the following type of error window will be showed:



Maintenance mode

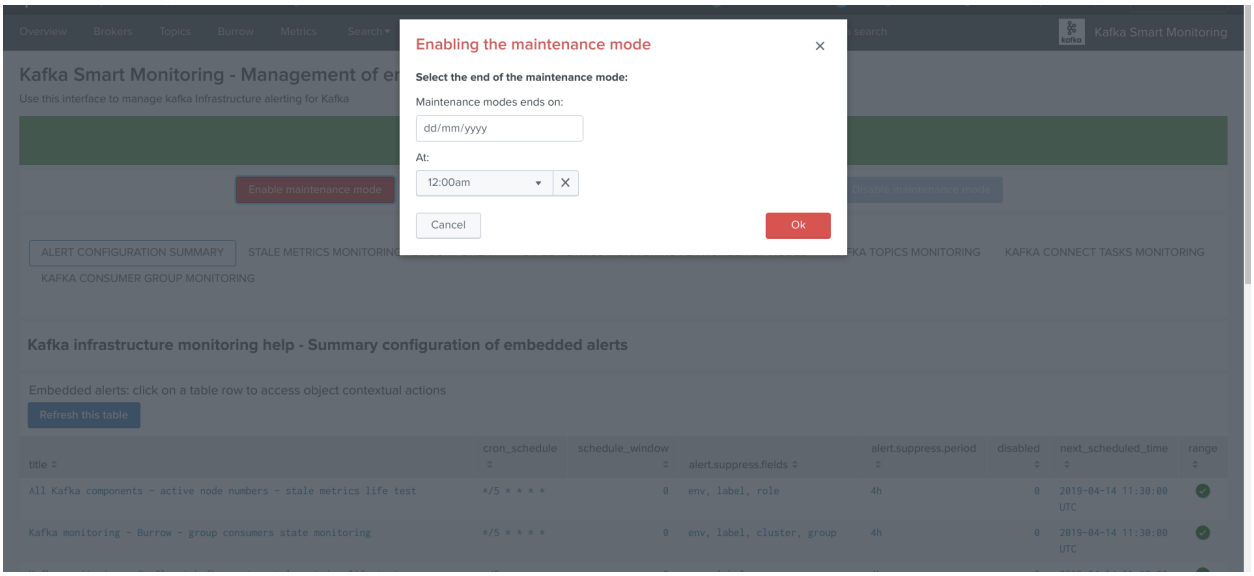
All alerts are by default driven by the status of the maintenance mode stored in a KVstore collection.

Shall the maintenance be enabled by an administrator, Splunk will continue to run the schedule alerts but none of them will be able to trigger during the maintenance time window.

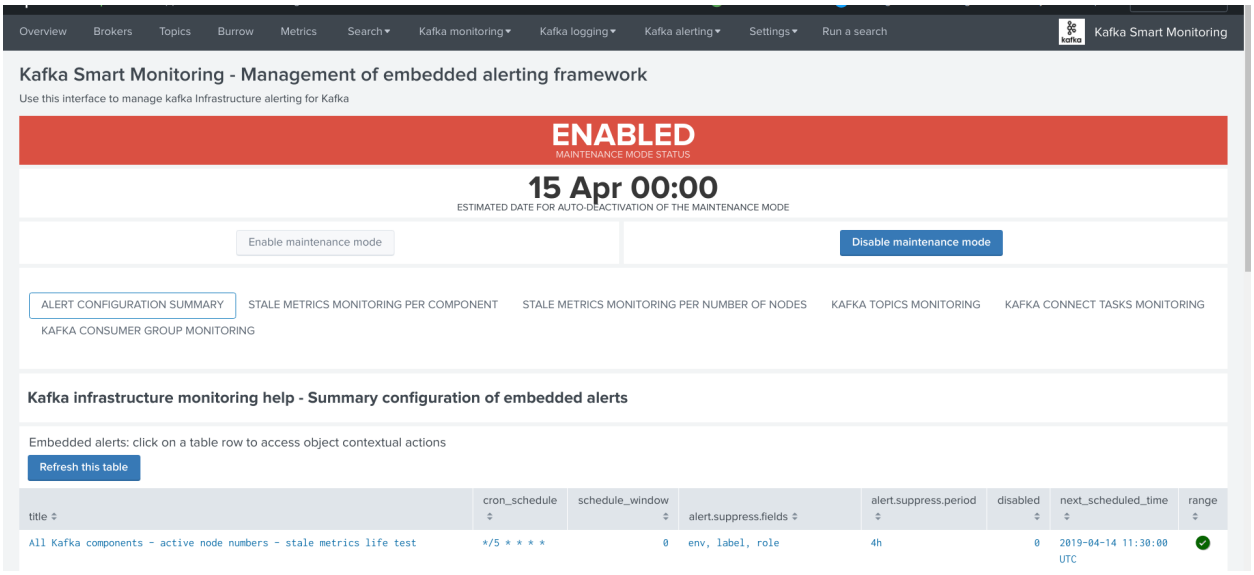
When the end of maintenance time window is reached, its state will be automatically disabled and alerts will be able to trigger again.

Enabling the maintenance mode

- Click on the enable maintenance mode button:



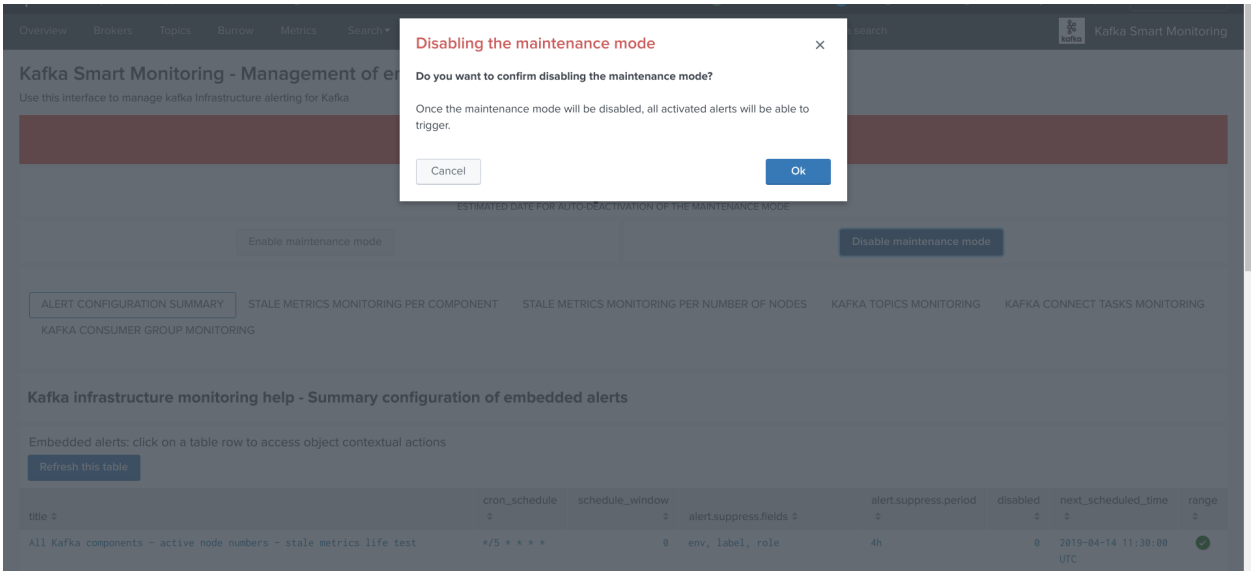
- Within the modal configuration window, enter the date and hours of the end of the maintenance time window:



- When the date and hours of the maintenance time window are reached, the scheduled report "Verify Kafka alerting maintenance status" will automatically disable the maintenance mode.

Disabling the maintenance mode

During any time of the maintenance time window, an administrator can decide to disable the maintenance mode:



The collection KVstore endpoint can be programmatically managed, as such it is easily possible to reproduce this behaviour from an external system.

(https://docs.splunk.com/Documentation/Splunk/latest/RESTREF/RESTkvstore)

Monitoring state default definition

When new objects are automatically discovered such as Kafka components or topics, these objects are added to the different KVstore collection with a default enabled maintenance mode.

The default maintenance mode that is applied on a per type of object basis can be customised via the associated macros definitions:

Purpose	Macro definition
Type of component (nodes number monitoring)	zookeeper_default_monitoring_state
Zookeeper nodes	zookeeper_default_monitoring_state
Kafka Brokers	kafka_broker_default_monitoring_state
Kafka Topics	kafka_topics_default_monitoring_state
Kafka Connect workers	kafka_connect_default_monitoring_state
Kafka Connect connectors	kafka_connect_tasks_default_monitoring_state
Kafka Burrow group consumers	kafka_burrow_consumers_default_monitoring_state
Confluent Schema registry	schema_registry_default_monitoring_state
Confluent ksql-server	ksql_server_default_monitoring_state
Confluent kafka-rest	kafka_rest_default_monitoring_state
LinkedIn kafka-monitor	kafka_monitor_default_monitoring_state

The default macro definition does the following statement:

```
eval monitoring_state="enabled"
```

A typical customisation can be to disable by default the monitoring state for non Production environments:

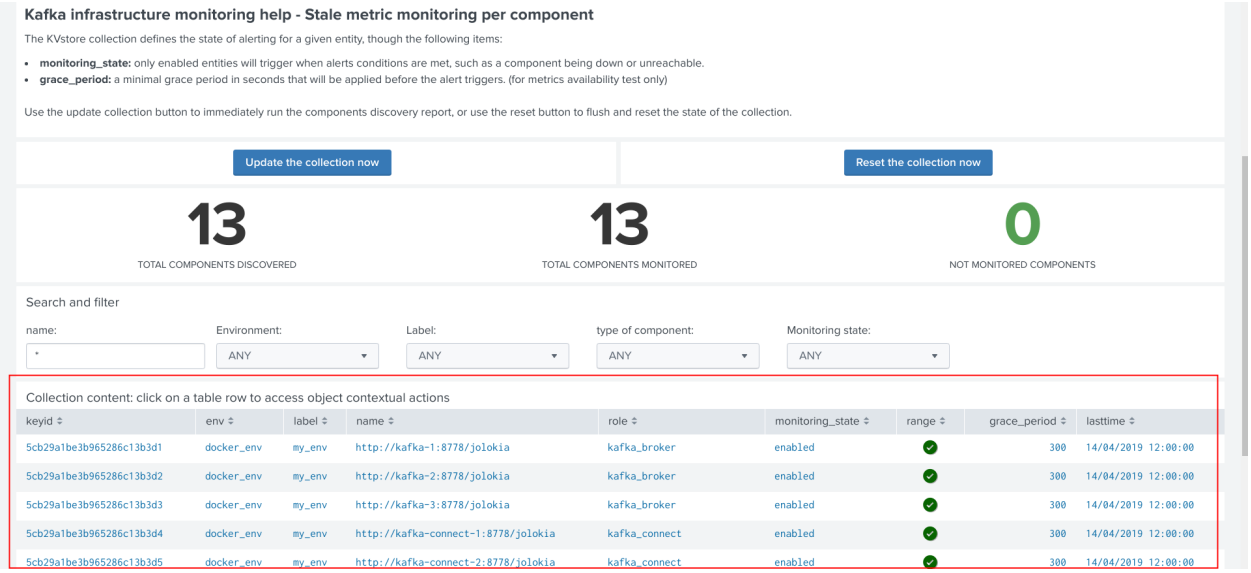
```
eval monitoring_state=if(match(env, "(?i)PROD"), "enabled", "disabled")
```

Such that if a new object is discovered for a development environment, this will not be monitored unless a manual update is performed via the user configuration interface.

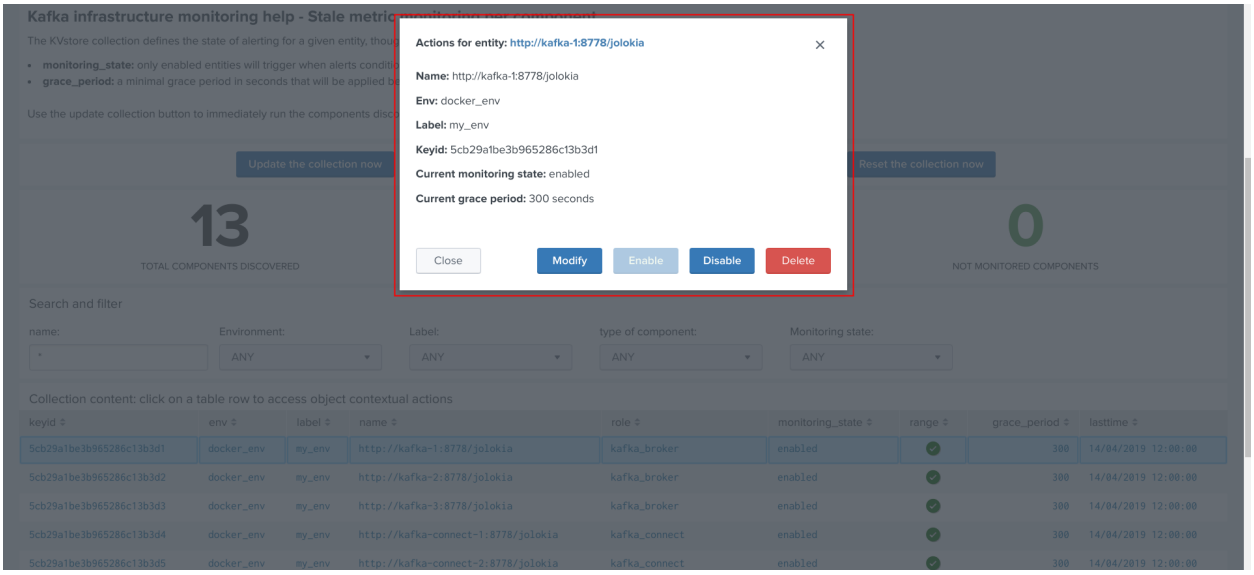
Administrating collection entries

Each type of component can be administrated in a dedicated tab within the user management interface.

When objects have been discovered, the administrator can eventually search for an object, and click on the object definition, which opens the modal interaction window:

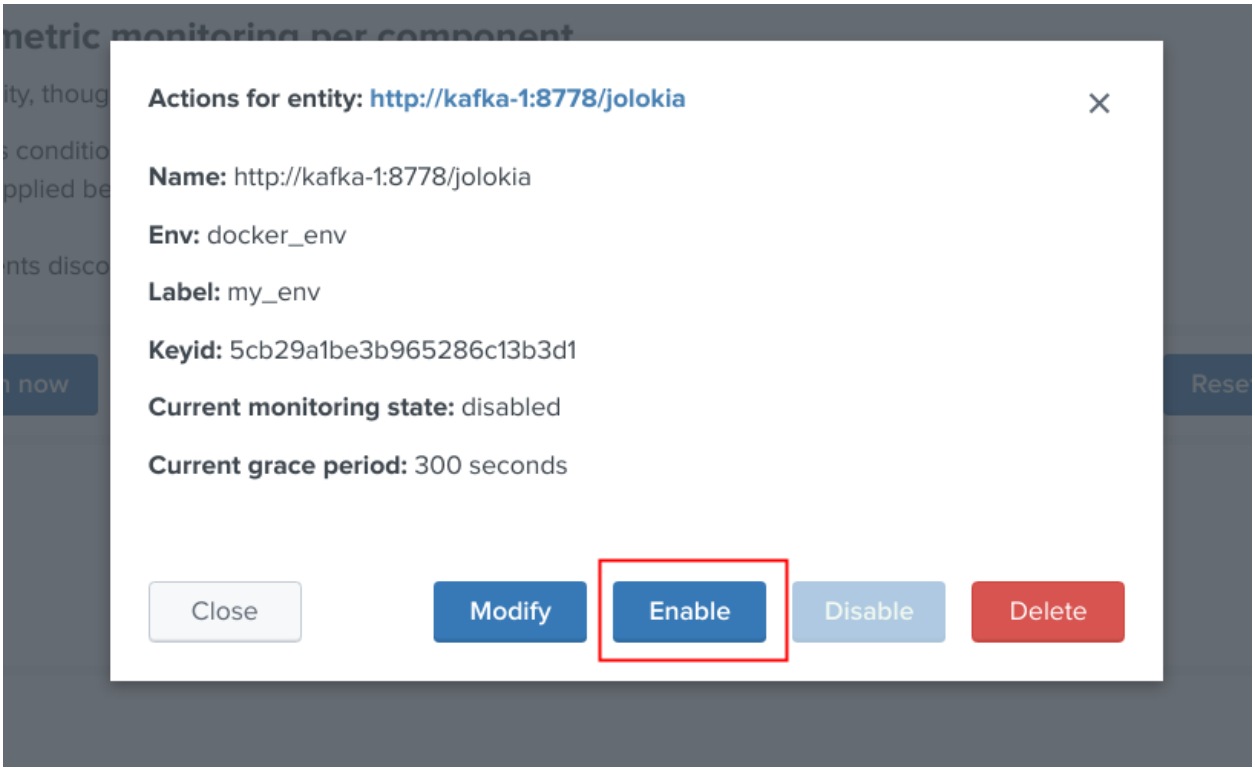


The modal interaction window provides information about this object, and different action buttons depending on this type of object:

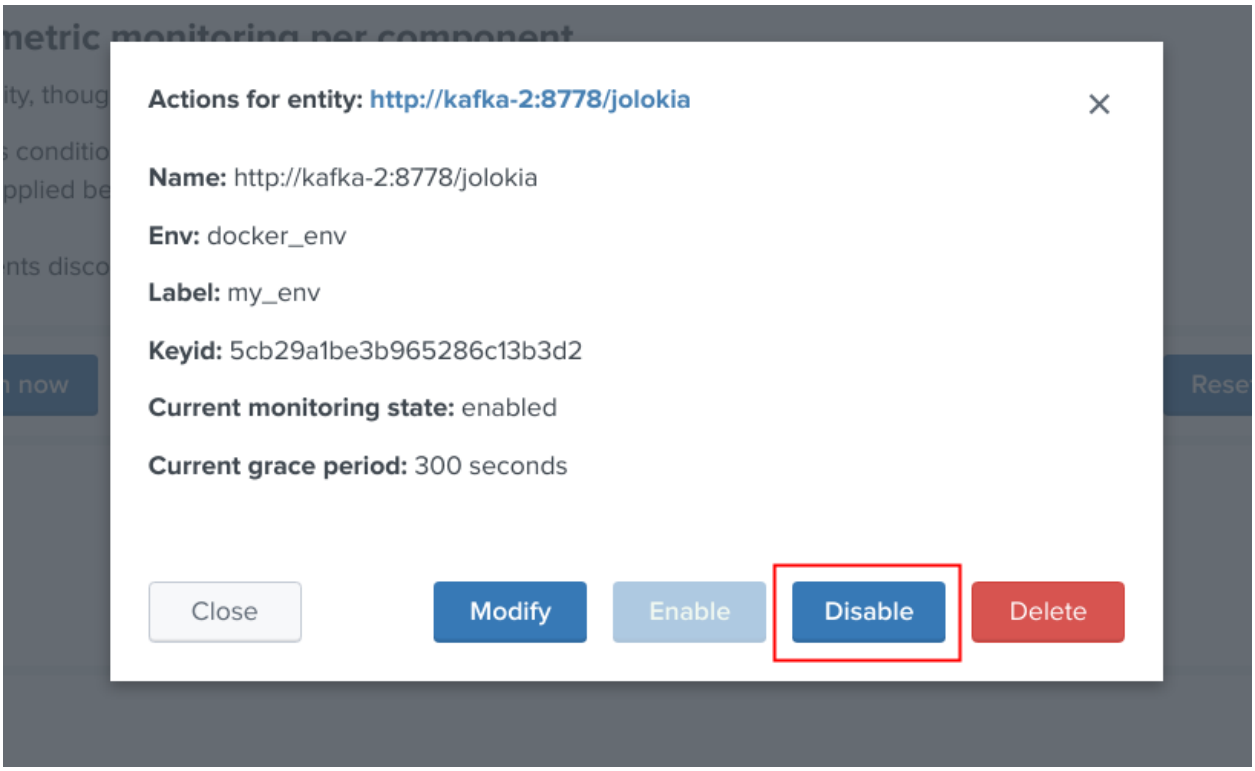


Enable/Disabling monitoring state

When an object has a disabled monitoring state, the button “enable monitoring” is automatically made available:



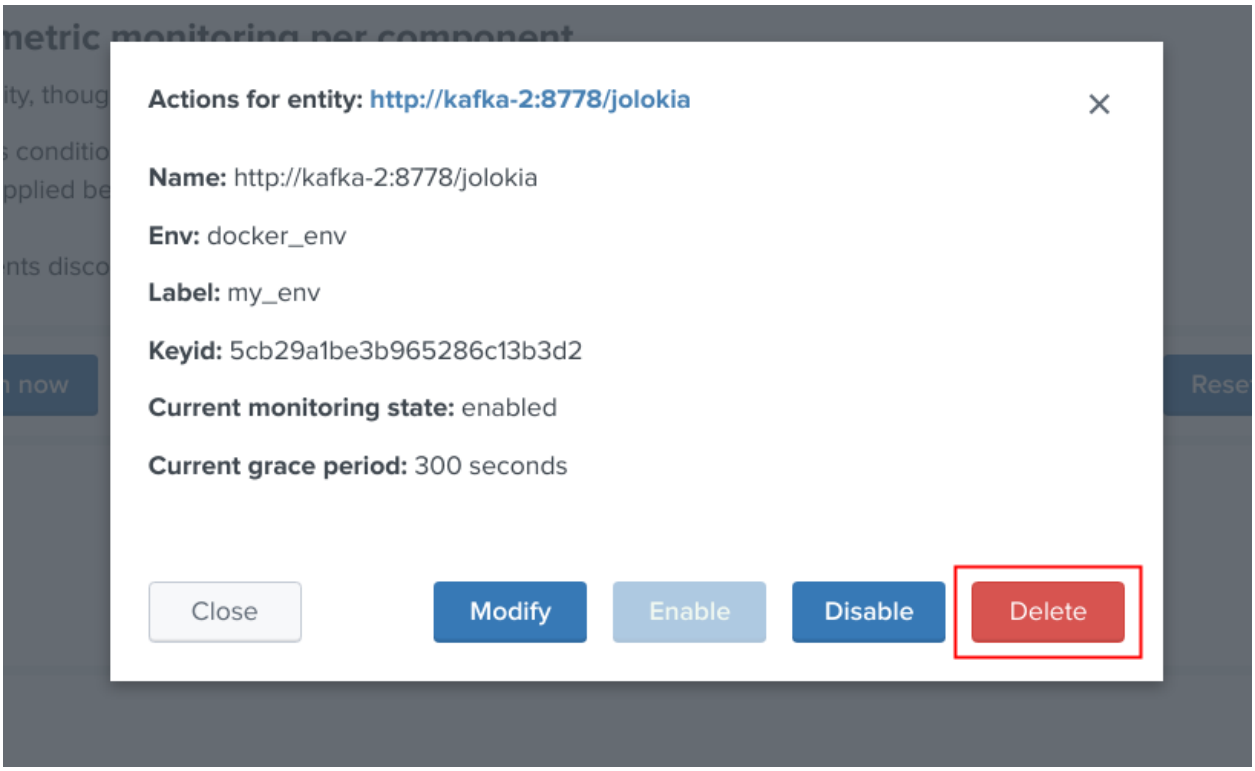
When an object has an enabled monitoring state, the button "disable monitoring" is automatically made available:



Shall the action be requested and confirmed, the object state will be updated, and the table exposing the object definition be refreshed.

Deleting objects in the collection

An object that was discovered and added to the collection automatically can be deleted via the UI:



Shall the action be requested and confirmed, the object state will be entirely removed from the collection, and the table exposing the object definition be refreshed.

Important:

By default, objects are discovered every 4 hours looking at metrics available for the last 4 hours.

This means that is the object has been still generated metrics to Splunk, it will be re-created automatically by the workflow.

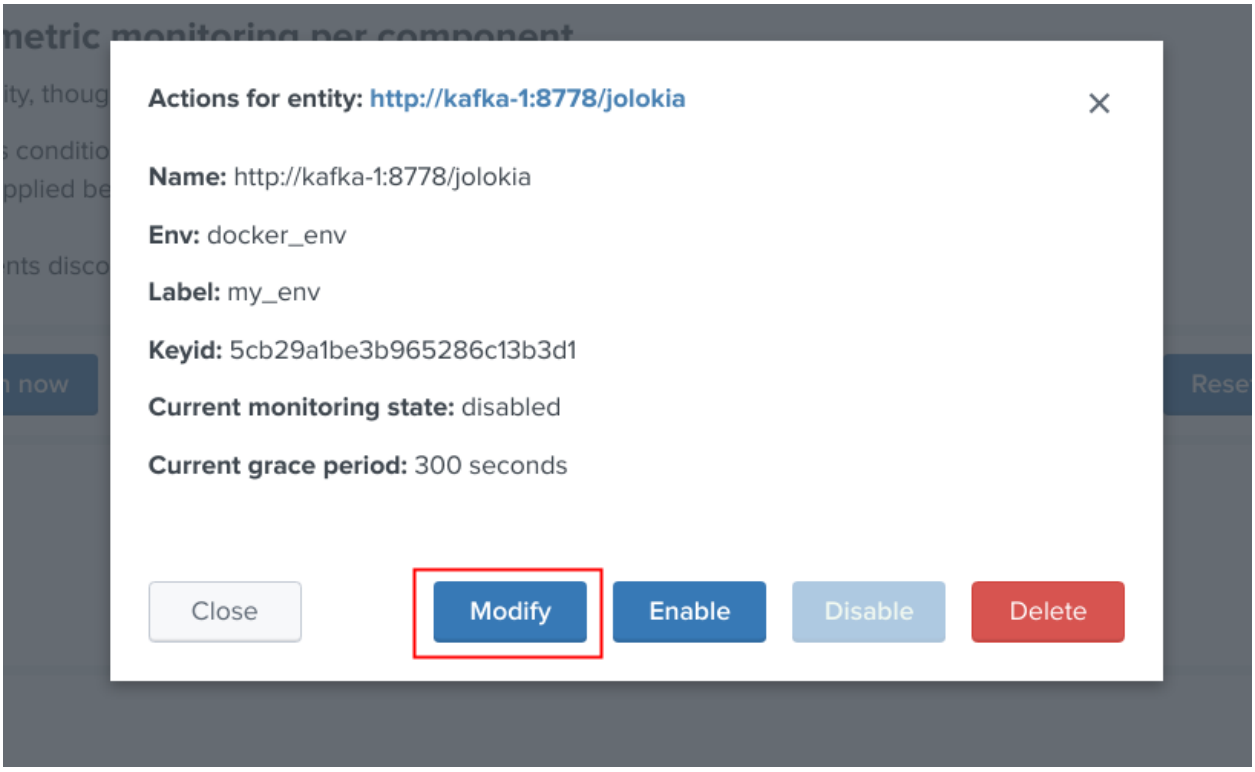
To avoid having to re-delete the same object again, you should wait 4 hours minimum before purging the object that was decommissioned.

Finally, note that if an object has not been generating metrics for a least 24 hours, its monitoring state will be disabled a special "disabled_autoforced" value.

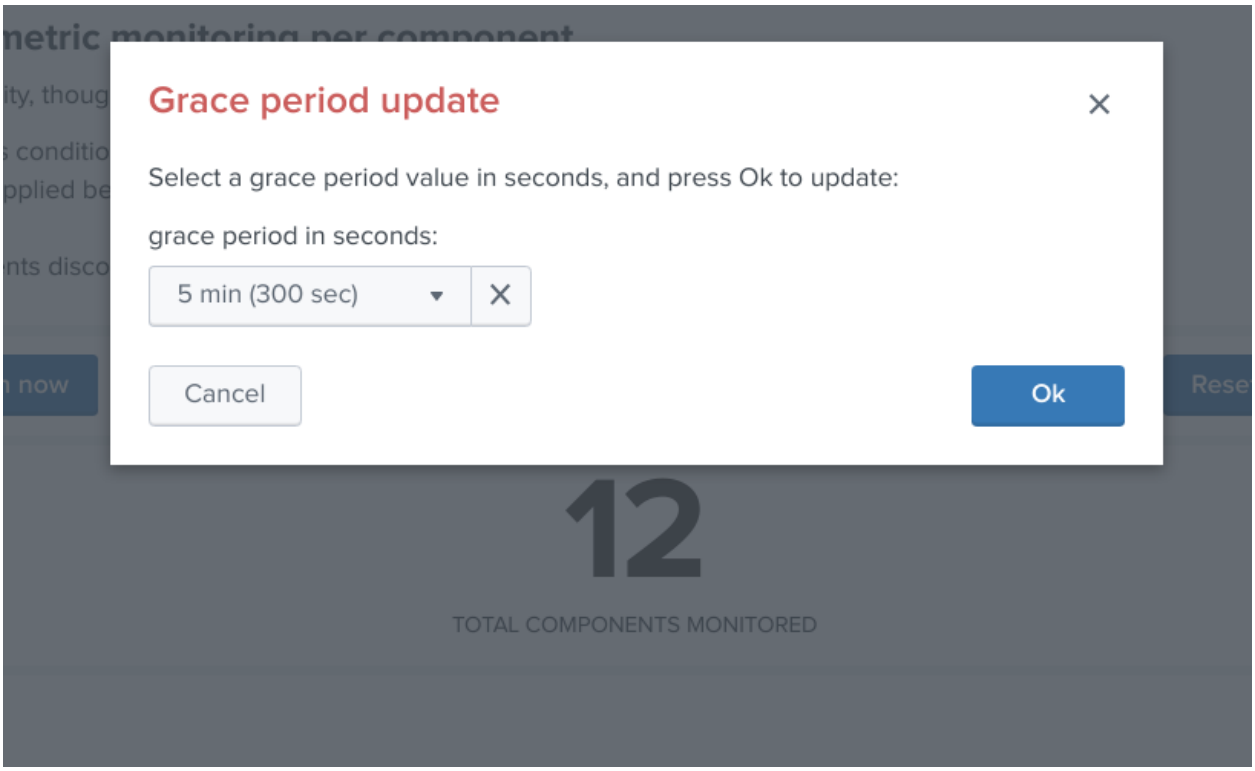
This state can still be manually updated via the UI, to permanently re-enable or disable the monitoring state if the component is still an active component.

Modifying an object in the collection

Depending on the type of object, the modal interaction window can provide a modification button:

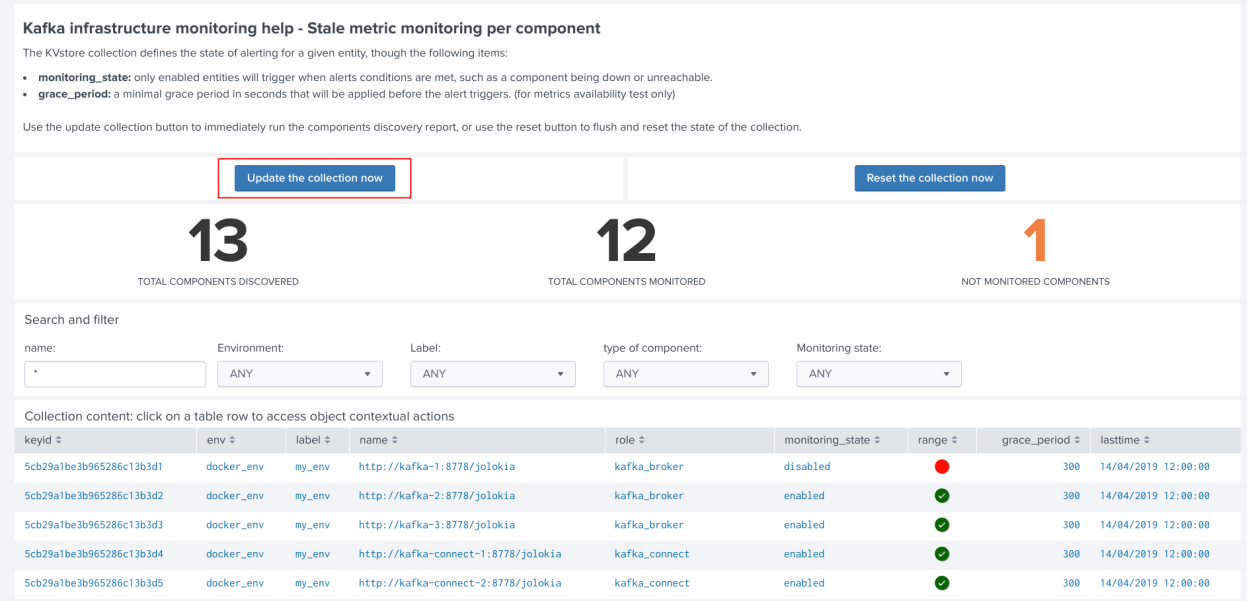


The type of modification that can be applied depends on type of component, example:

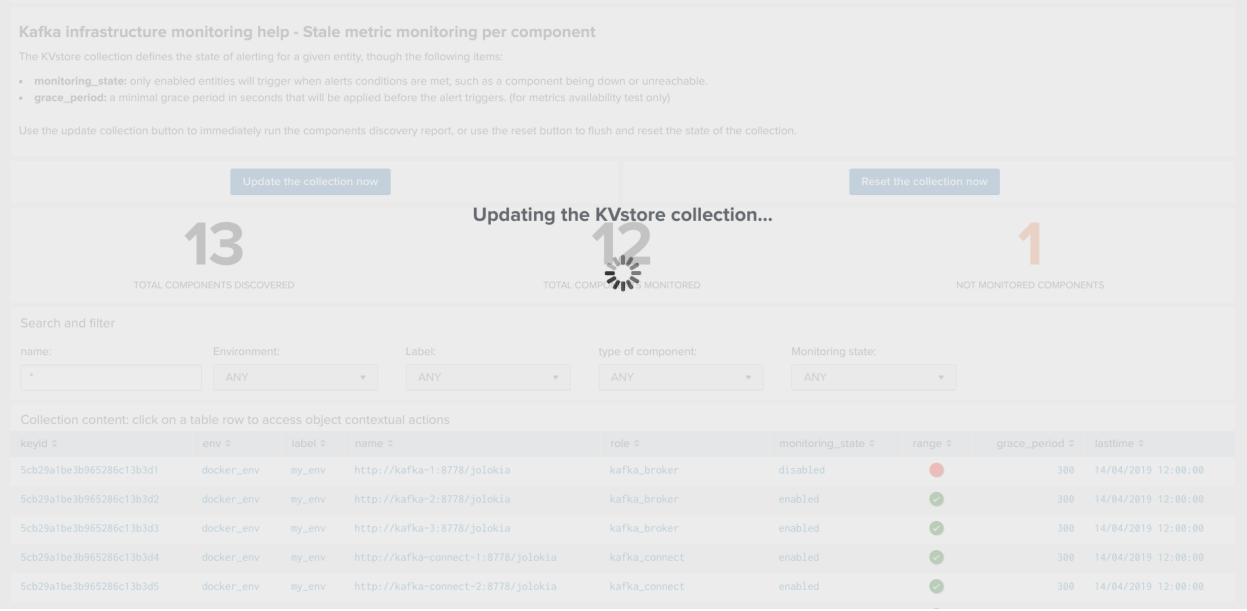


Manually request a collection update job

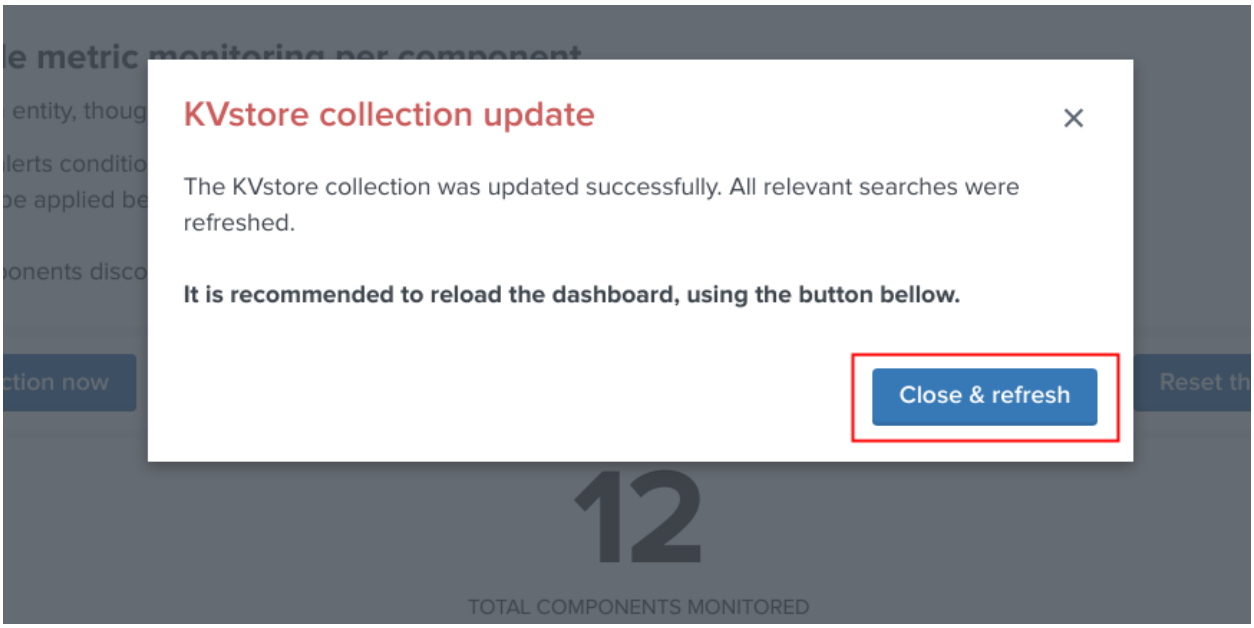
A collection update can be requested at any time within the UI:



Shall the action be requested and confirmed, the UI will automatically run the object discovery report, any new object that was not yet discovered since the last run of the report, will be added to the collection and made available within the UI.

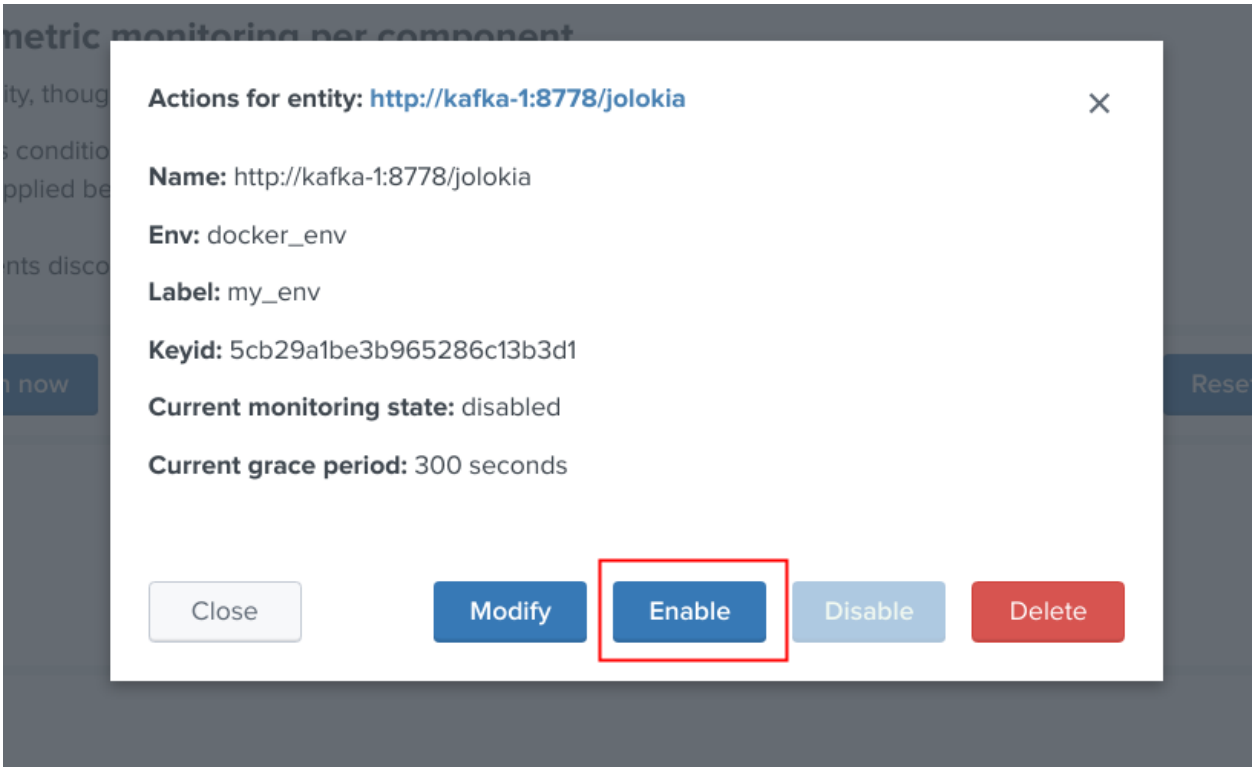


Once the job has run, click on the refresh button:

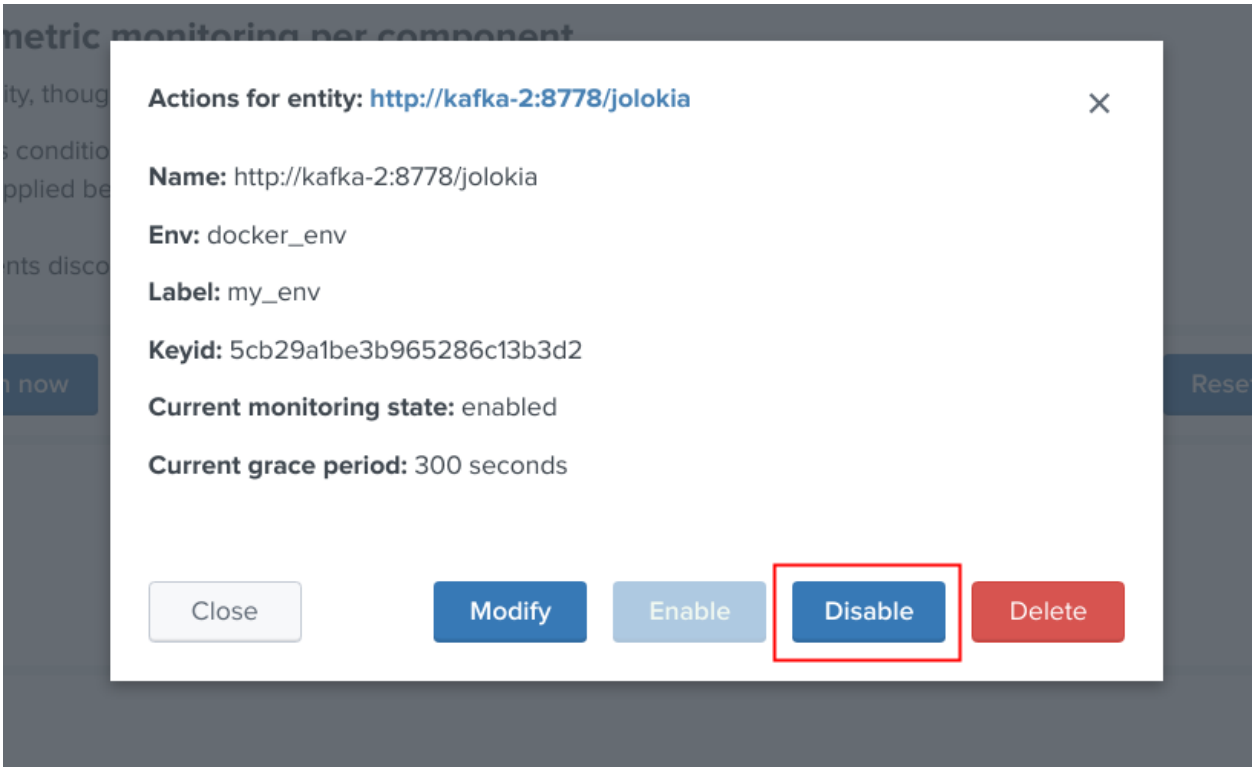


Enable/Disabling monitoring state

When an object has a disabled monitoring state, the button "enable monitoring" is automatically made available:



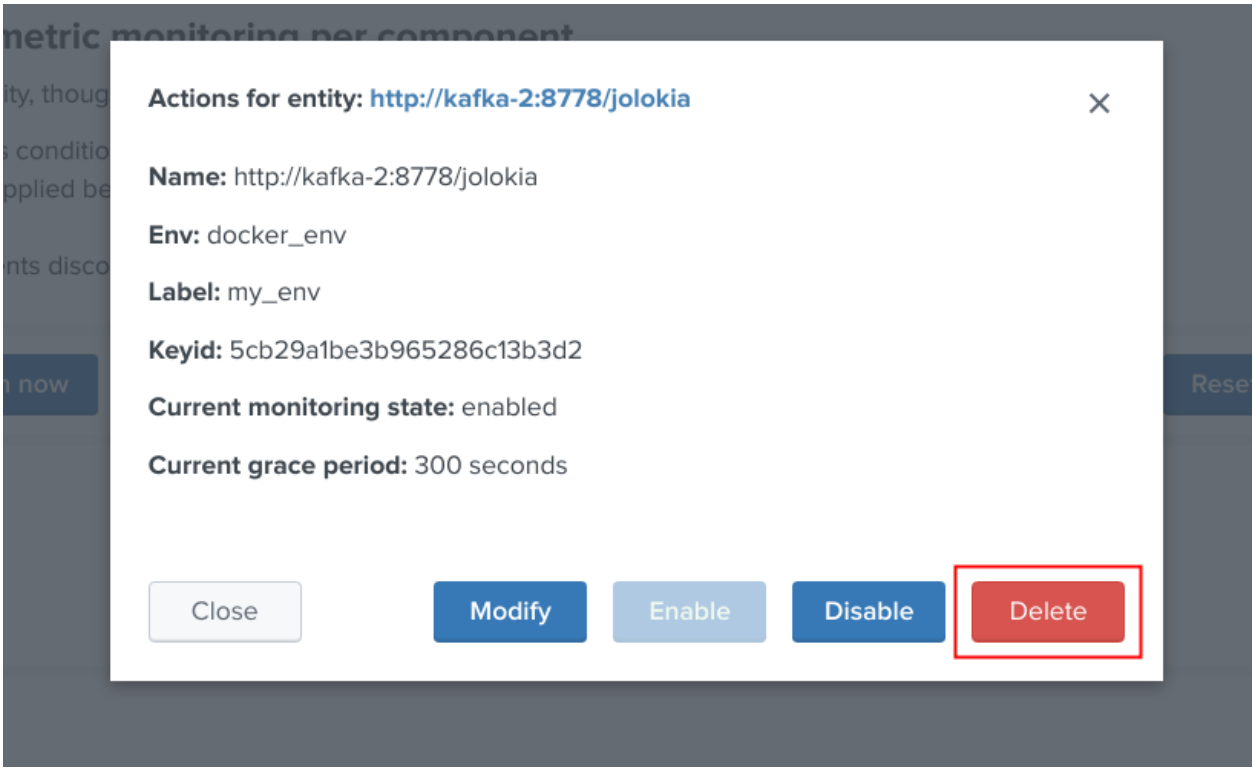
When an object has an enabled monitoring state, the button "disable monitoring" is automatically made available:



Shall the action be requested and confirmed, the object state will be updated, and the table exposing the object definition be refreshed.

Deleting objects in the collection

An object that was discovered and added to the collection automatically can be deleted via the UI:



Shall the action be requested and confirmed, the object state will be entirely removed from the collection, and the table exposing the object definition be refreshed.

Important:

By default, objects are discovered every 4 hours looking at metrics available for the last 4 hours.

This means that is the object has been still generated metrics to Splunk, it will be re-created automatically by the workflow.

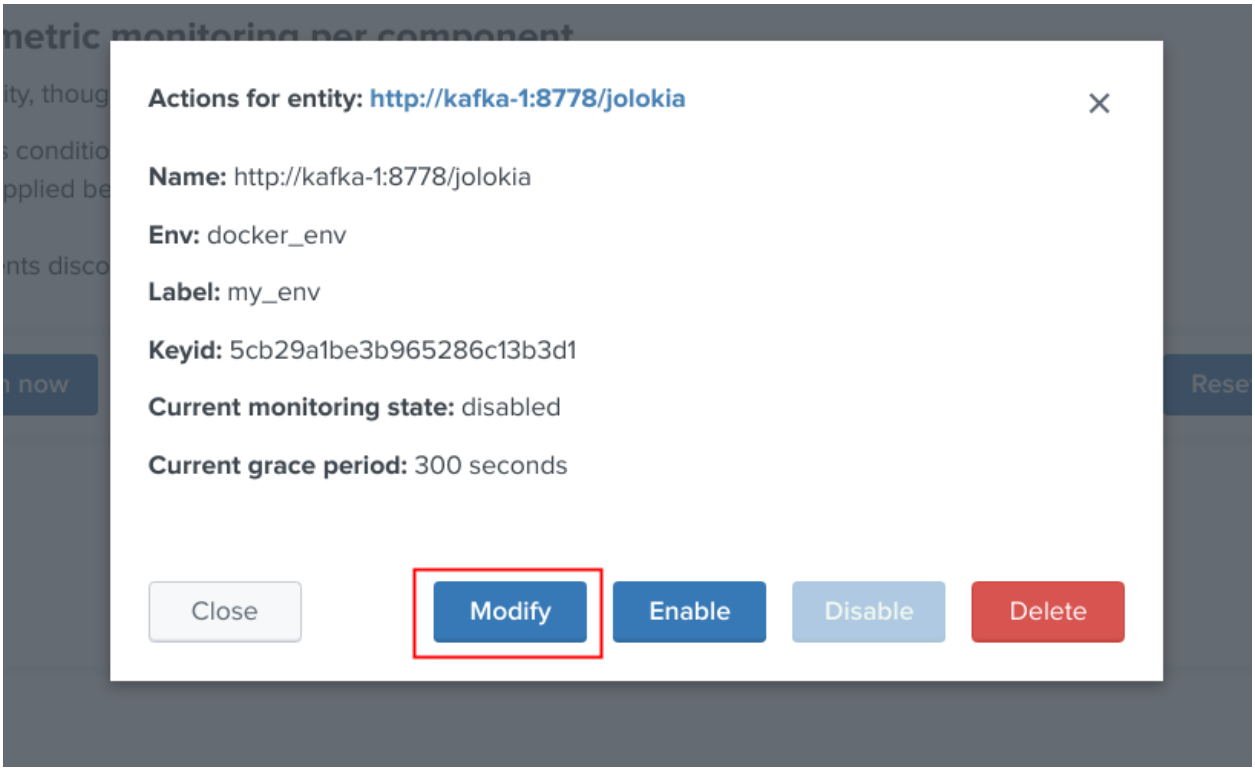
To avoid having to re-delete the same object again, you should wait 4 hours minimum before purging the object that was decommissioned.

Finally, note that if an object has not been generating metrics for a least 24 hours, its monitoring state will be disabled a special "disabled_autoforced" value.

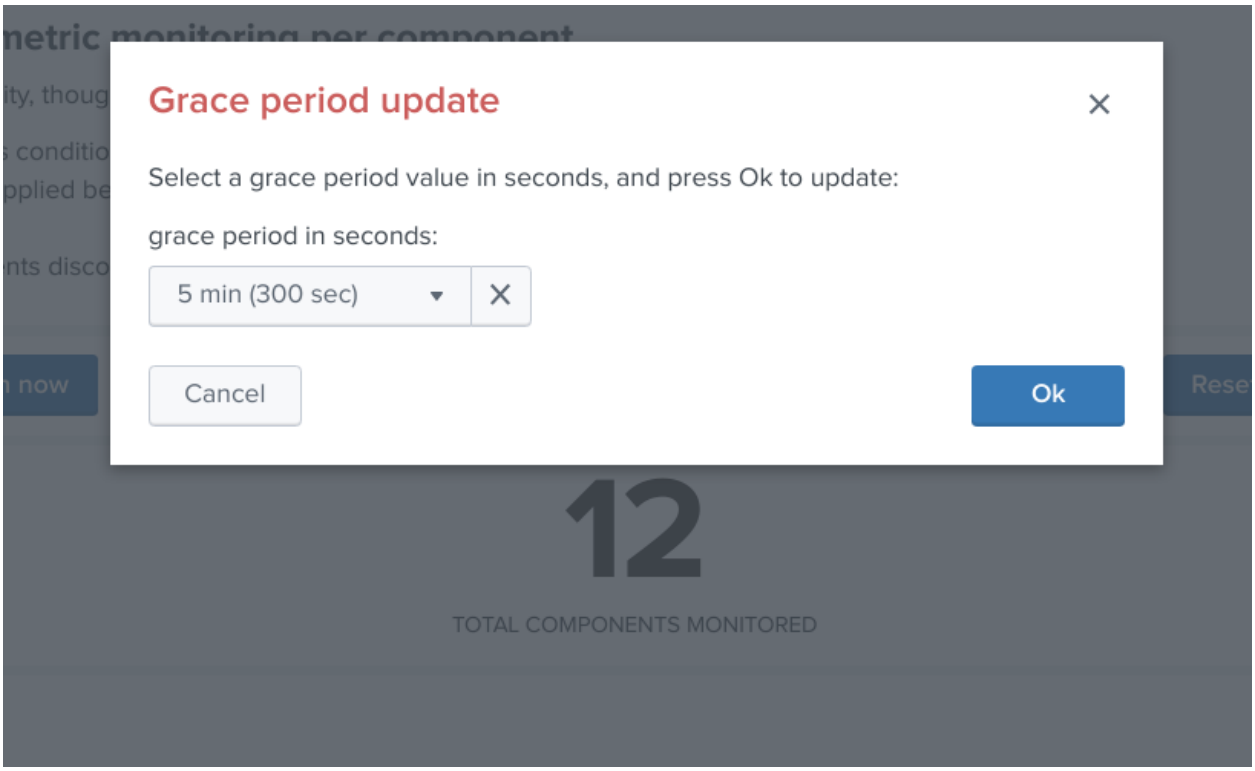
This state can still be manually updated via the UI, to permanently re-enable or disable the monitoring state if the component is still an active component.

Modifying an object in the collection

Depending on the type of object, the modal interaction window can provide a modification button:

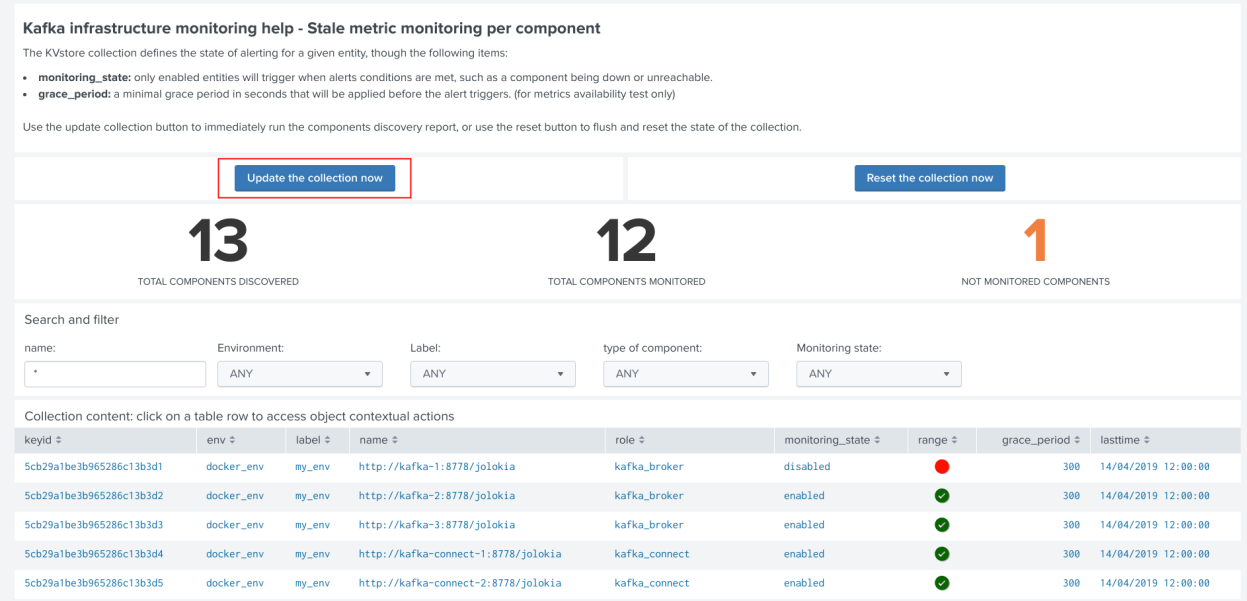


The type of modification that can be applied depends on type of component, example:

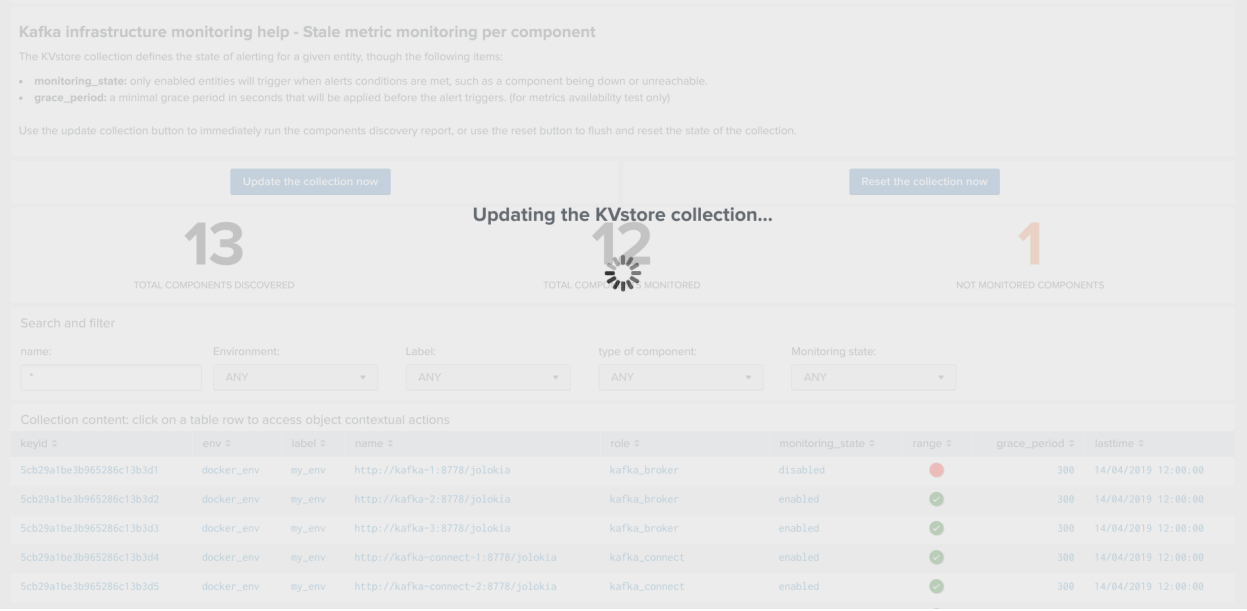


Manually request a collection update job

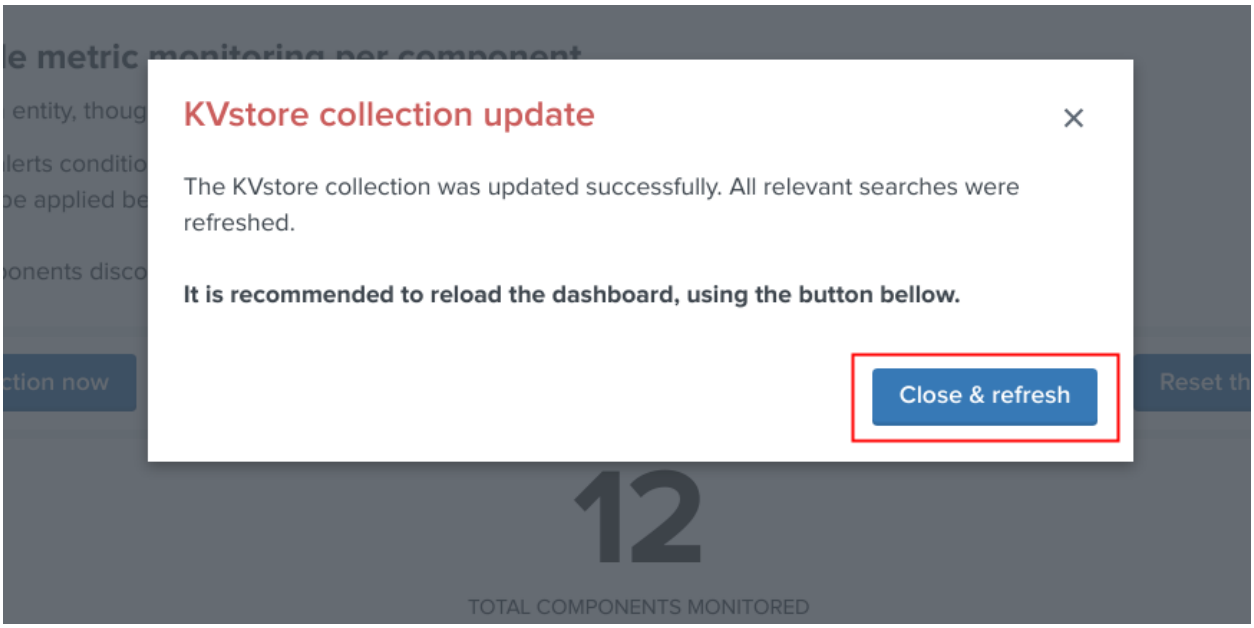
A collection update can be requested at any time within the UI:



Shall the action be requested and confirmed, the UI will automatically run the object discovery report, any new object that was not yet discovered since the last run of the report, will be added to the collection and made available within the UI.



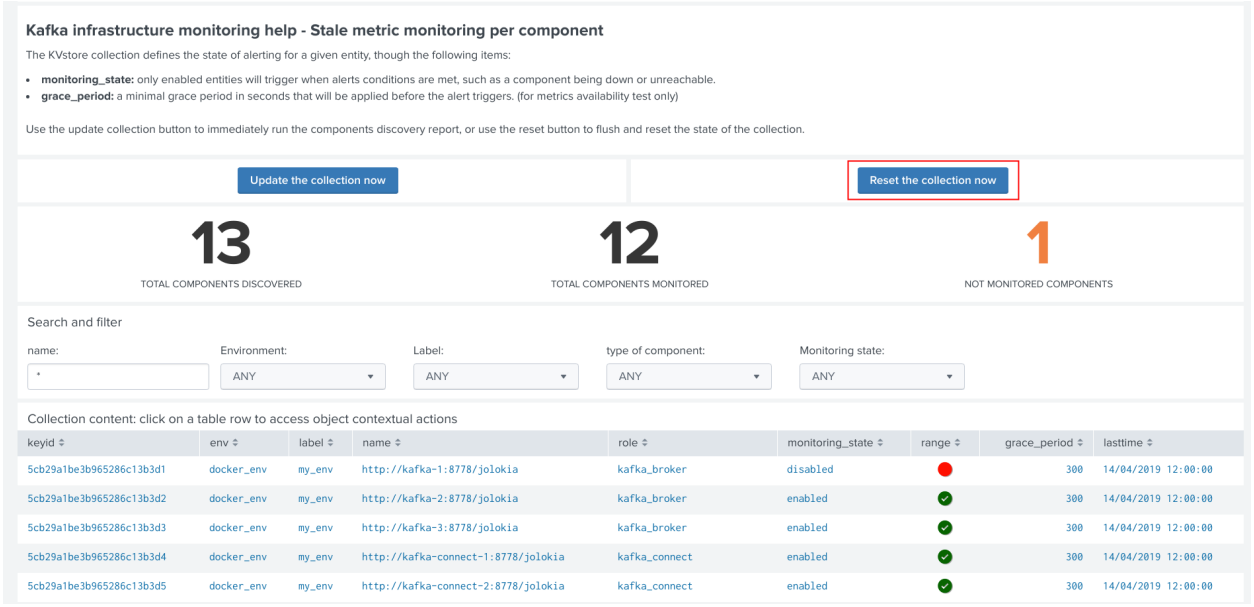
Once the job has run, click on the refresh button:



Shall the job fail for some reasons such as a lack of permissions, an error window with the Splunk error message would be exposed automatically.

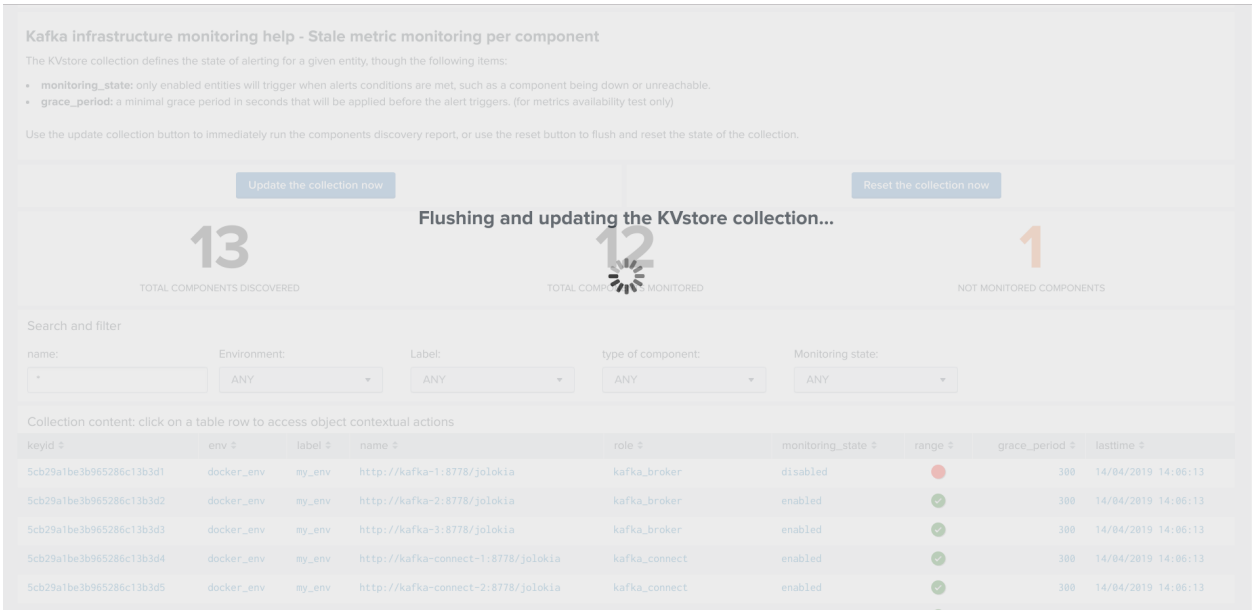
Manually request a collection rebuild job

A collection reset can be requested at any time within the UI:

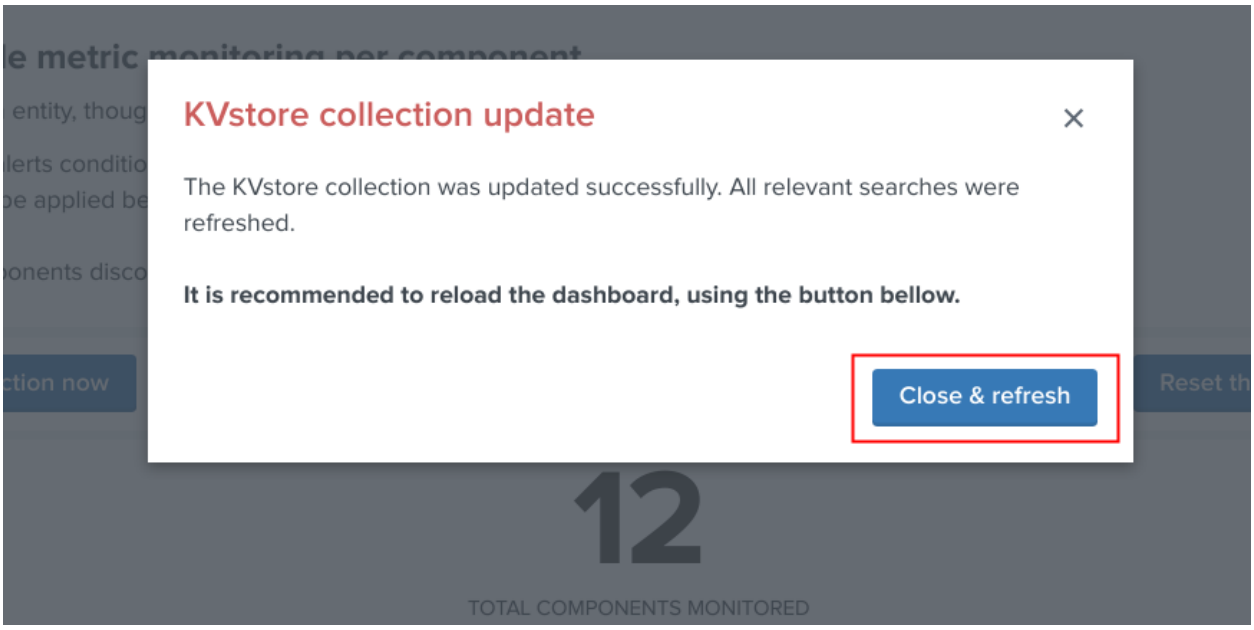


Important: When requesting a reset of the collection, all changes will be irremediably lost. All matching objects will be reset to their default discovered values.

Shall the action be requested and confirmed, the UI will automatically run the object discovery report, any new object that was not yet discovered since the last run of the report, will be added to the collection and made available within the UI.



Once the job has run, click on the refresh button:



Shall the job fail for some reasons such as a lack of permissions, an error window with the Splunk error message would be exposed automatically.

Enabling OOTB alerts

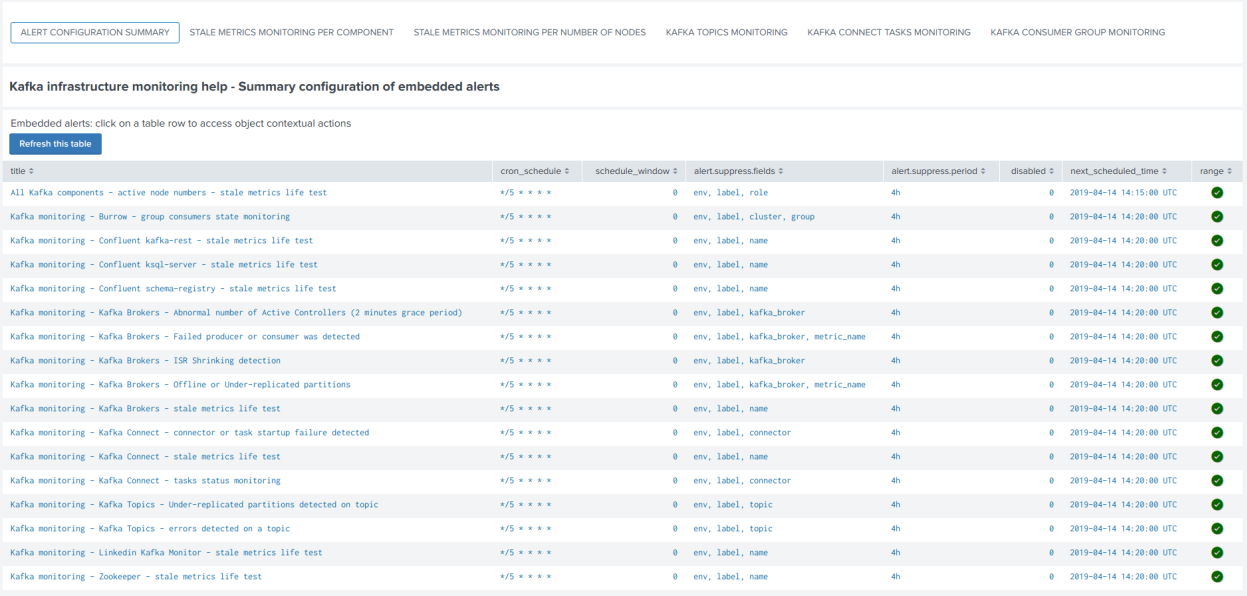
Important: By default, all alerts are disabled, you must enable the alerts within Splunk Web depending on your needs.

You need to decide which alert must be enabled depending on your needs and environments, and achieve any additional alert actions that would be required such as creating an incident in a ticketing system.

Splunk alerts can easily be extended by alert actions.

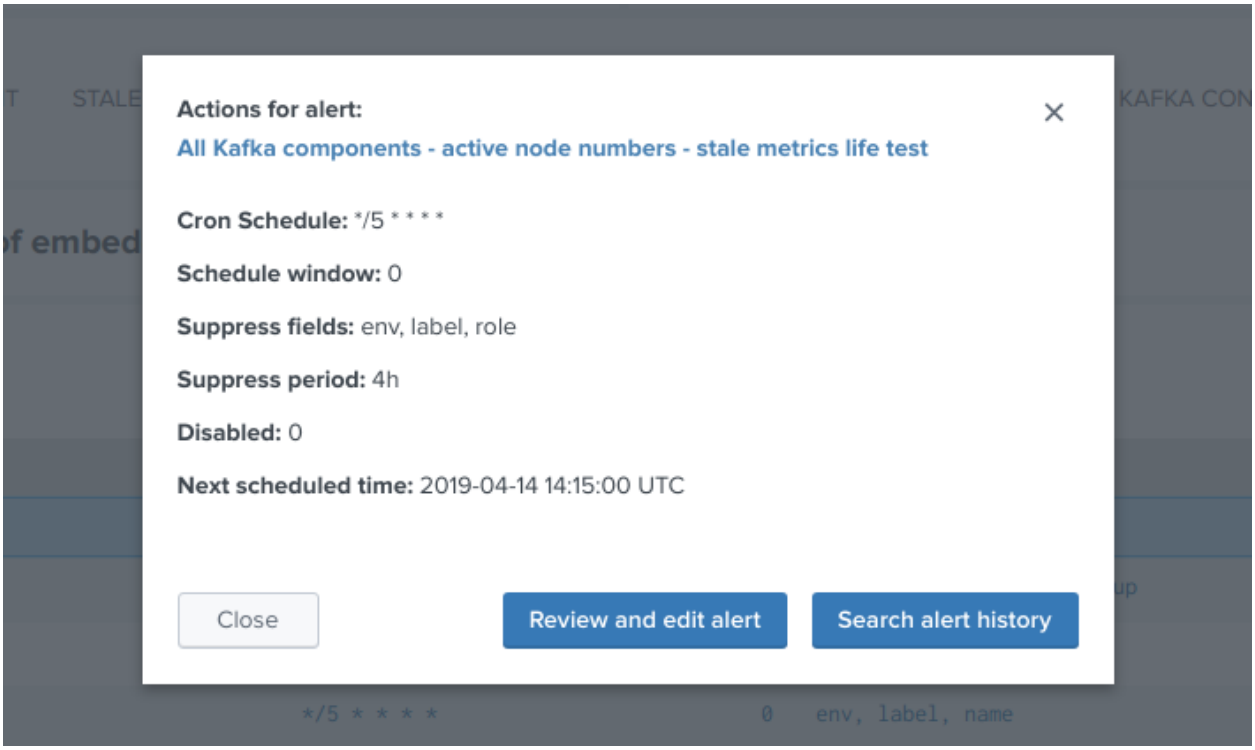
Alert configuration summary user interface

The summary alert tab exposes most valuable information about the alerts, and provides a shortcut access to the management of the alerts:

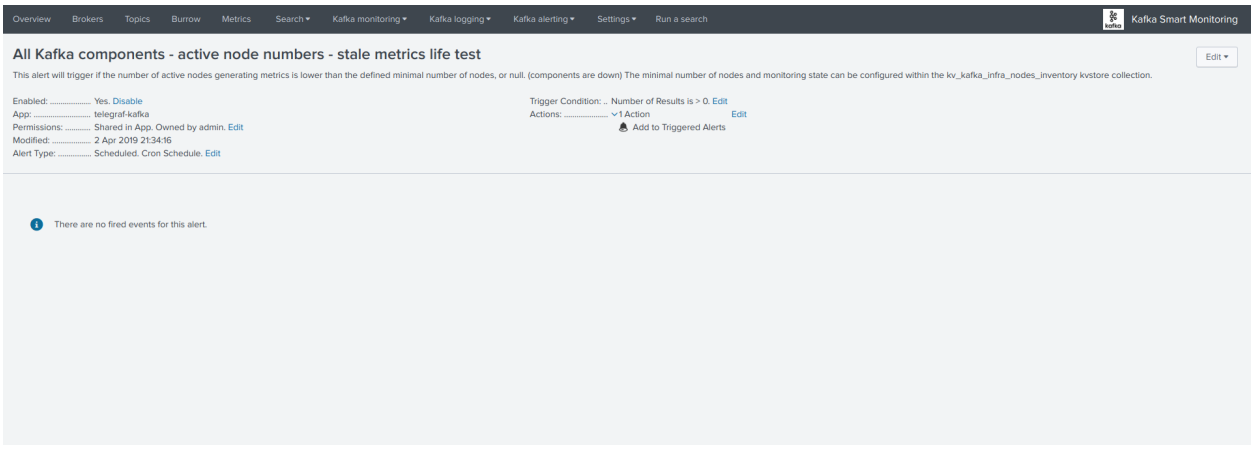


title	cron_schedule	schedule_window	alert.suppress.fields	alert.suppress.period	disabled	next_scheduled_time	range
All Kafka components - active node numbers - stale metrics life test	*/5 * * * *	0	env, label, role	4h	0	2019-04-14 14:15:00 UTC	
Kafka monitoring - Burrow - group consumers state monitoring	*/5 * * * *	0	env, label, cluster, group	4h	0	2019-04-14 14:20:00 UTC	
Kafka monitoring - Confluent kafka-rest - stale metrics life test	*/5 * * * *	0	env, label, name	4h	0	2019-04-14 14:20:00 UTC	
Kafka monitoring - Confluent ksql-server - stale metrics life test	*/5 * * * *	0	env, label, name	4h	0	2019-04-14 14:20:00 UTC	
Kafka monitoring - Confluent schema-registry - stale metrics life test	*/5 * * * *	0	env, label, name	4h	0	2019-04-14 14:20:00 UTC	
Kafka monitoring - Kafka Brokers - Abnormal number of Active Controllers (2 minutes grace period)	*/5 * * * *	0	env, label, kafka_broker	4h	0	2019-04-14 14:20:00 UTC	
Kafka monitoring - Kafka Brokers - Failed producer or consumer was detected	*/5 * * * *	0	env, label, kafka_broker, metric_name	4h	0	2019-04-14 14:20:00 UTC	
Kafka monitoring - Kafka Brokers - ISR Shrinking detection	*/5 * * * *	0	env, label, kafka_broker	4h	0	2019-04-14 14:20:00 UTC	
Kafka monitoring - Kafka Brokers - Offline or Under-replicated partitions	*/5 * * * *	0	env, label, kafka_broker, metric_name	4h	0	2019-04-14 14:20:00 UTC	
Kafka monitoring - Kafka Brokers - stale metrics life test	*/5 * * * *	0	env, label, name	4h	0	2019-04-14 14:20:00 UTC	
Kafka monitoring - Kafka Connect - connector or task startup failure detected	*/5 * * * *	0	env, label, connector	4h	0	2019-04-14 14:20:00 UTC	
Kafka monitoring - Kafka Connect - stale metrics life test	*/5 * * * *	0	env, label, name	4h	0	2019-04-14 14:20:00 UTC	
Kafka monitoring - Kafka Connect - tasks status monitoring	*/5 * * * *	0	env, label, connector	4h	0	2019-04-14 14:20:00 UTC	
Kafka monitoring - Kafka Topics - Under-replicated partitions detected on topic	*/5 * * * *	0	env, label, topic	4h	0	2019-04-14 14:20:00 UTC	
Kafka monitoring - Kafka Topics - errors detected on a topic	*/5 * * * *	0	env, label, topic	4h	0	2019-04-14 14:20:00 UTC	
Kafka monitoring - Linkedin Kafka Monitor - stale metrics life test	*/5 * * * *	0	env, label, name	4h	0	2019-04-14 14:20:00 UTC	
Kafka monitoring - Zookeeper - stale metrics life test	*/5 * * * *	0	env, label, name	4h	0	2019-04-14 14:20:00 UTC	

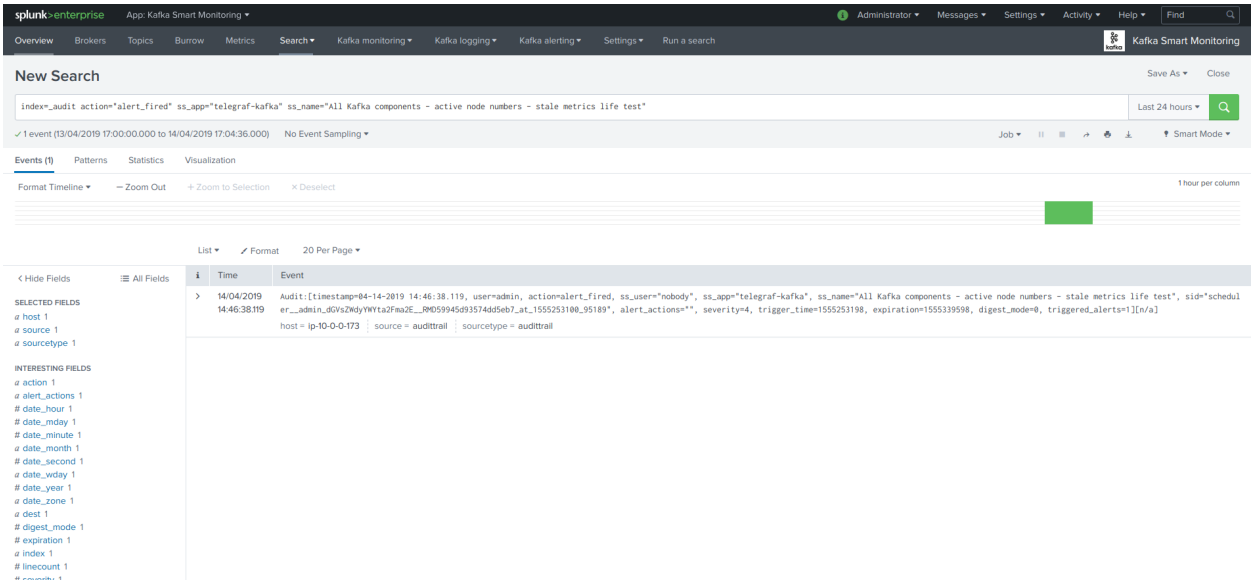
Click on any alert to open the modal interaction window:



Click on the "Review and edit alert" button to open the Splunk alert configuration UI for this alert:



Click on the "Search alert history" button to automatically open a search against the triggering history for this alert



Stale metrics life test by component

Life test monitoring alerts perform a verification of the metric availability to alert on a potential downtime or issue with a component.

- Kafka monitoring - [component] - stale metrics life test

Once activated, stale metrics alert verify the grace period to be applied, and the monitoring state of the component from the KVstore collection.

Alerts can be controlled by changing values of the fields:

- grace_period: The grace value in seconds before assuming a severe status (difference in seconds between the last communication and time of the check)
- monitoring_state: A value of "enabled" activates verification, any other value disables it

Stale metrics life test by number of nodes per type of component

If you are running the Kafka components in a container based architecture, you can monitor your infrastructure availability by monitoring the number of active nodes per type of component.

As such, you will be monitoring how many nodes are active at a time, rather than specific nodes identities which will change with the life cycle of the containers.

- All Kafka components - active node numbers - stale metrics life test

Shall an upgrade of a statefullSet or deployment in Kubernetes fail and new containers fail to start, the OOTB alerting will report this bad condition on per type of component basis.

Kafka brokers monitoring

The following alerts are available to monitor the main and most important aspects of Kafka Broker clusters:

- Abnormal number of Active Controllers
- Offline or Under-replicated partitions
- Failed producer or consumer was detected
- ISR Shrinking detection

Kafka topics monitoring

The following alerts are available to monitor Kafka topics:

- Under-replicated partitions detected on topics
- Errors reported on topics (bytes rejected, failed fetch requests, failed produce requests)

Kafka Connect task monitoring

Alerts are available to monitor the state of connectors and tasks for Kafka Connect:

- Kafka monitoring - Kafka Connect - tasks status monitoring

Alerts can be controlled by changing values of the fields:

- grace_period: The grace value in seconds before assuming a severe status (difference in seconds between the last communication and time of the check)
- monitoring_state: A value of “enabled” activates verification, any other value disables it

Kafka Consumers monitoring with Burrow

Alerts are available to monitor and report the state of Kafka Consumers via Burrow:

- Kafka monitoring - Burrow - group consumers state monitoring

Alerts can be controlled by changing values of the fields:

- monitoring_state: A value of “enabled” activates verification, any other value disables it

Notes: Kafka Connect source and sink connectors depending on their type are as well consumers, Burrow will monitor the way the connectors behave by analysing their lagging metrics and type of activity, this is a different, complimentary and advanced type of monitoring than analysing the state of the tasks.

1.7.2 ITSI: Advanced monitoring, machine learning, technical and business services monitoring

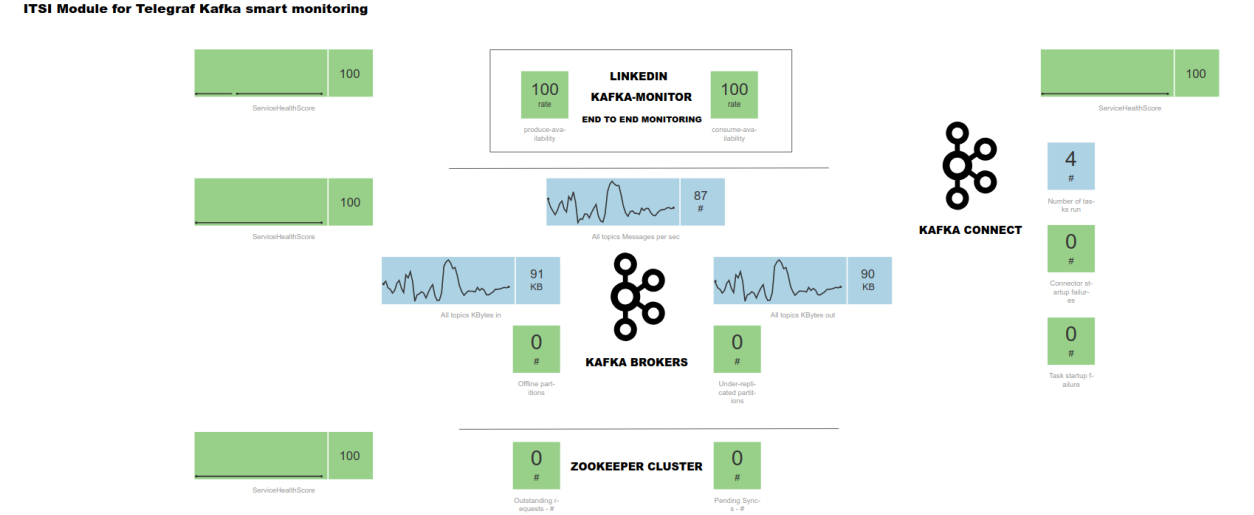
The ITSI Module for Telegraf Apache Kafka smart monitoring is available in Splunk Base:

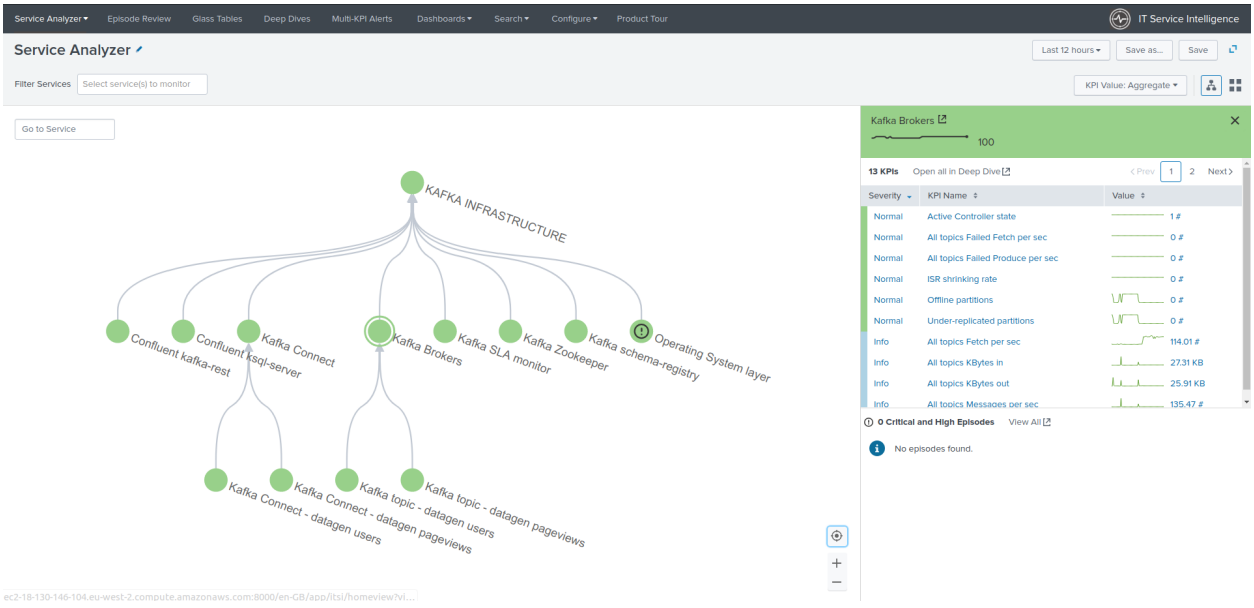
https://splunkbase.splunk.com/app/4261

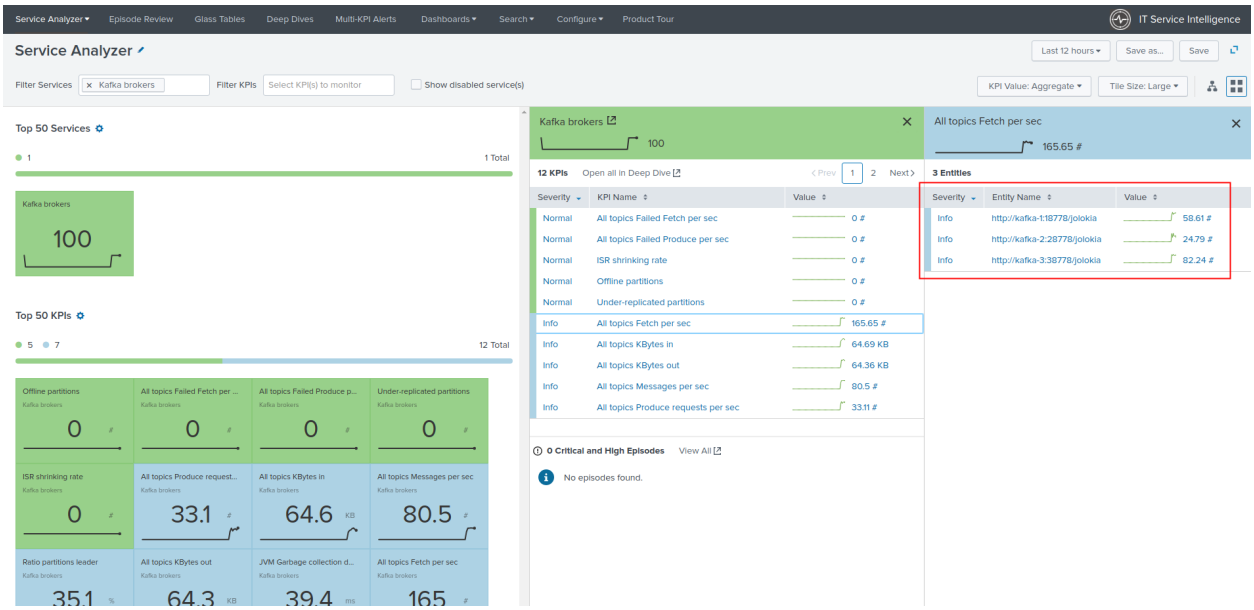
The dedicated documentation Web site is available here:

https://da-itsi-telegraf-kafka.readthedocs.io/

ITSI provides a totally different and infinitely more advanced level than a traditional monitoring solution, the ITSI module for Kafka used in combination with ITSI provides the real difference with business and technical services design and monitoring, machine learning and so much more.







Please consult the documentation of the ITSI module for more details about its configuration and use.