Red Hat OpenStack Platform 16.2

Service Telemetry Framework 1.4

Installing and deploying Service Telemetry Framework 1.4

Last Updated: 2023-01-31

Red Hat OpenStack Platform 16.2 Service Telemetry Framework 1.4

Installing and deploying Service Telemetry Framework 1.4

OpenStack Team
rhos-docs@redhat.com

Legal Notice

Abstract

Install the core components and deploy Service Telemetry Framework 1.4.

Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE 4

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION 5

CHAPTER 1. INTRODUCTION TO SERVICE TELEMETRY FRAMEWORK 1.4 6
1.1. SUPPORT FOR SERVICE TELEMETRY FRAMEWORK 6
1.2. SERVICE TELEMETRY FRAMEWORK ARCHITECTURE 7
1.3. INSTALLATION SIZE OF RED HAT OPENSHIFT CONTAINER PLATFORM 9

CHAPTER 2. PREPARING YOUR RED HAT OPENSHIFT CONTAINER PLATFORM ENVIRONMENT FOR SERVICE TELEMETRY FRAMEWORK 10
2.1. OBSERVABILITY STRATEGY IN SERVICE TELEMETRY FRAMEWORK 10
2.2. PERSISTENT VOLUMES 10
2.2.1. Ephemeral storage 11
2.3. RESOURCE ALLOCATION 11
2.4. NETWORK CONSIDERATIONS FOR SERVICE TELEMETRY FRAMEWORK 11

CHAPTER 3. INSTALLING THE CORE COMPONENTS OF SERVICE TELEMETRY FRAMEWORK 12
3.1. DEPLOYING SERVICE TELEMETRY FRAMEWORK TO THE RED HAT OPENSHIFT CONTAINER PLATFORM ENVIRONMENT 12
3.2. CREATING A SERVICETELEMETRY OBJECT IN RED HAT OPENSHIFT CONTAINER PLATFORM 16
3.2.1. Primary parameters of the ServiceTelemetry object 19
The backends parameter 19
Enabling Prometheus as a storage back end for metrics 20
Configuring persistent storage for Prometheus 20
Enabling ElasticSearch as a storage back end for events 21
Configuring persistent storage for ElasticSearch 21
The clouds parameter 22
The alerting parameter 23
The graphing parameter 23
The highAvailability parameter 23
The transports parameter 23
3.3. ACCESSING USER INTERFACES FOR STF COMPONENTS 23
3.4. CONFIGURING AN ALTERNATE OBSERVABILITY STRATEGY 24

CHAPTER 4. CONFIGURING RED HAT OPENSTACK PLATFORM FOR SERVICE TELEMETRY FRAMEWORK . 26
4.1. DEPLOYING RED HAT OPENSTACK PLATFORM OVERCLOUD FOR SERVICE TELEMETRY FRAMEWORK 26
4.1.1. Retrieving the AMQ Interconnect route address 26
4.1.2. Creating the base configuration for STF 27
4.1.3. Configuring the STF connection for the overcloud 29
4.1.4. Deploying the overcloud 30
4.1.5. Validating client-side installation 31
4.2. SENDING METRICS TO GNOCCHI AND SERVICE TELEMETRY FRAMEWORK 34
4.3. DEPLOYING TO NON-STANDARD NETWORK TOPOLOGIES 35
4.4. CONFIGURING MULTIPLE CLOUDS 37
4.4.1. Planning AMQP address prefixes 38
4.4.2. Deploying Smart Gateways 38
4.4.3. Deleting the default Smart Gateways 40
4.4.4. Setting a unique cloud domain 41
4.4.5. Creating the Red Hat OpenStack Platform environment file for multiple clouds 42
4.4.6. Querying metrics data from multiple clouds 44

CHAPTER 5. USING OPERATIONAL FEATURES OF SERVICE TELEMETRY FRAMEWORK 45
5.1. DASHBOARDS IN SERVICE TELEMETRY FRAMEWORK 45
5.1.1. Configuring Grafana to host the dashboard 45
5.1.2. Overriding the default Grafana container image 47
5.1.3. Importing dashboards 47
5.1.4. Retrieving and setting Grafana login credentials 49
5.2. METRICS RETENTION TIME PERIOD IN SERVICE TELEMETRY FRAMEWORK 49
5.2.1. Editing the metrics retention time period in Service Telemetry Framework 50
5.3. ALERTS IN SERVICE TELEMETRY FRAMEWORK 51
5.3.1. Creating an alert rule in Prometheus 51
5.3.2. Configuring custom alerts 52
5.3.3. Creating a standard alert route in Alertmanager 53
5.3.4. Creating an alert route with templating in Alertmanager 55
5.4. CONFIGURING SNMP TRAPS 57
5.5. HIGH AVAILABILITY 58
5.5.1. Configuring high availability 58
5.6. EPHEMERAL STORAGE 59
5.6.1. Configuring ephemeral storage 59
5.7. OBSERVABILITY STRATEGY IN SERVICE TELEMETRY FRAMEWORK 60
5.7.1. Configuring an alternate observability strategy 60
5.8. RESOURCE USAGE OF RED HAT OPENSTACK PLATFORM SERVICES 61
5.8.1. Disabling resource usage monitoring of Red Hat OpenStack Platform services 61
5.9. RED HAT OPENSTACK PLATFORM API STATUS AND CONTAINERIZED SERVICES HEALTH 62
5.9.1. Disabling container health and API status monitoring 62

CHAPTER 6. UPGRADING SERVICE TELEMETRY FRAMEWORK TO VERSION 1.4 64
6.1. REMOVING STF 1.3 SMART GATEWAY OPERATOR 64
6.2. UPDATING THE SERVICE TELEMETRY OPERATOR TO 1.4 65

CHAPTER 7. REMOVING SERVICE TELEMETRY FRAMEWORK FROM THE RED HAT OPENSHIFT CONTAINER PLATFORM ENVIRONMENT 67
7.1. DELETING THE NAMESPACE 67
7.2. REMOVING THE CATALOGSOURCE 67
7.3. REMOVING THE AMQ CERTIFICATE MANAGER OPERATOR 68

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message .

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your input on our documentation. Tell us how we can make it better.

Using the Direct Documentation Feedback (DDF) function

Use the **Add Feedback** DDF function for direct comments on specific sentences, paragraphs, or code blocks.

1. View the documentation in the *Multi-page HTML* format.

2. Ensure that you see the **Feedback** button in the upper right corner of the document.

3. Highlight the part of text that you want to comment on.

4. Click **Add Feedback**.

5. Complete the **Add Feedback** field with your comments.

6. Optional: Add your email address so that the documentation team can contact you for clarification on your issue.

7. Click **Submit**.

CHAPTER 1. INTRODUCTION TO SERVICE TELEMETRY FRAMEWORK 1.4

Service Telemetry Framework (STF) collects monitoring data from Red Hat OpenStack Platform (RHOSP) or third-party nodes. You can use STF to perform the following tasks:

- Store or archive the monitoring data for historical information.
- View the monitoring data graphically on the dashboard.
- Use the monitoring data to trigger alerts or warnings.

The monitoring data can be either metric or event:

Metric

A numeric measurement of an application or system.

Event

Irregular and discrete occurrences that happen in a system.

The components of STF use a message bus for data transport. Other modular components that receive and store data are deployed as containers on Red Hat OpenShift Container Platform.

IMPORTANT

Service Telemetry Framework (STF) is compatible with Red Hat OpenShift Container Platform version 4.8.

Additional resources

- For more information about how to deploy Red Hat OpenShift Container Platform, see the Red Hat OpenShift Container Platform product documentation.
- You can install Red Hat OpenShift Container Platform on cloud platforms or on bare metal. For more information about STF performance and scaling, see https://access.redhat.com/articles/4907241.
- You can install Red Hat OpenShift Container Platform on bare metal or other supported cloud platforms. For more information about installing Red Hat OpenShift Container Platform, see OpenShift Container Platform 4.10 Documentation .

1.1. SUPPORT FOR SERVICE TELEMETRY FRAMEWORK

Red Hat supports the two most recent versions of Service Telemetry Framework (STF). Earlier versions are not supported. For more information, see the Service Telemetry Framework Supported Version Matrix.

Red Hat supports the core Operators and workloads, including AMQ Interconnect, AMQ Certificate Manager, Service Telemetry Operator, and Smart Gateway Operator. Red Hat does not support the community Operators or workload components, such as Elasticsearch, Prometheus, Alertmanager, Grafana, and their Operators.

You can only deploy STF in a fully connected network environment. You cannot deploy STF in Red Hat OpenShift Container Platform-disconnected environments or network proxy environments.

1.2. SERVICE TELEMETRY FRAMEWORK ARCHITECTURE

Service Telemetry Framework (STF) uses a client-server architecture, in which Red Hat OpenStack Platform (RHOSP) is the client and Red Hat OpenShift Container Platform is the server.

STF consists of the following components:

- Data collection
  - collectd: Collects infrastructure metrics and events.
  - Ceilometer: Collects RHOSP metrics and events.
- Transport
  - AMQ Interconnect: An AMQP 1.x compatible messaging bus that provides fast and reliable data transport to transfer the metrics to STF for storage.
  - Smart Gateway: A Golang application that takes metrics and events from the AMQP 1.x bus to deliver to ElasticSearch or Prometheus.
- Data storage
  - Prometheus: Time-series data storage that stores STF metrics received from the Smart Gateway.
  - ElasticSearch: Events data storage that stores STF events received from the Smart Gateway.
- Observation
  - Alertmanager: An alerting tool that uses Prometheus alert rules to manage alerts.
  - Grafana: A visualization and analytics application that you can use to query, visualize, and explore data.

The following table describes the application of the client and server components:

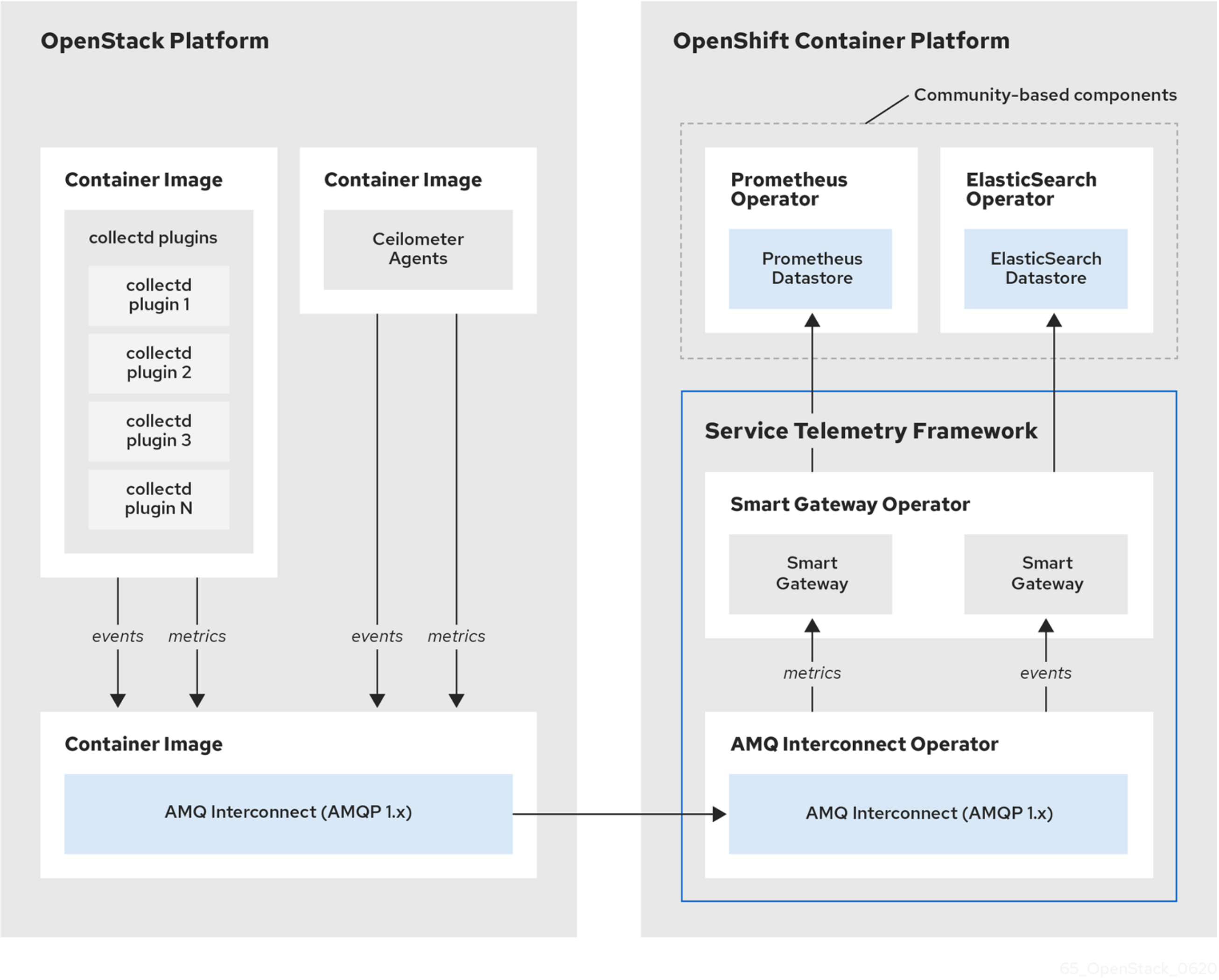
Table 1.1. Client and server components of STF

Component	Client	Server
An AMQP 1.x compatible messaging bus	yes	yes
Smart Gateway	no	yes
Prometheus	no	yes
ElasticSearch	no	yes
collectd	yes	no
Ceilometer	yes	no

IMPORTANT

To ensure that the monitoring platform can report operational problems with your cloud, do not install STF on the same infrastructure that you are monitoring.

Figure 1.1. Service Telemetry Framework architecture overview



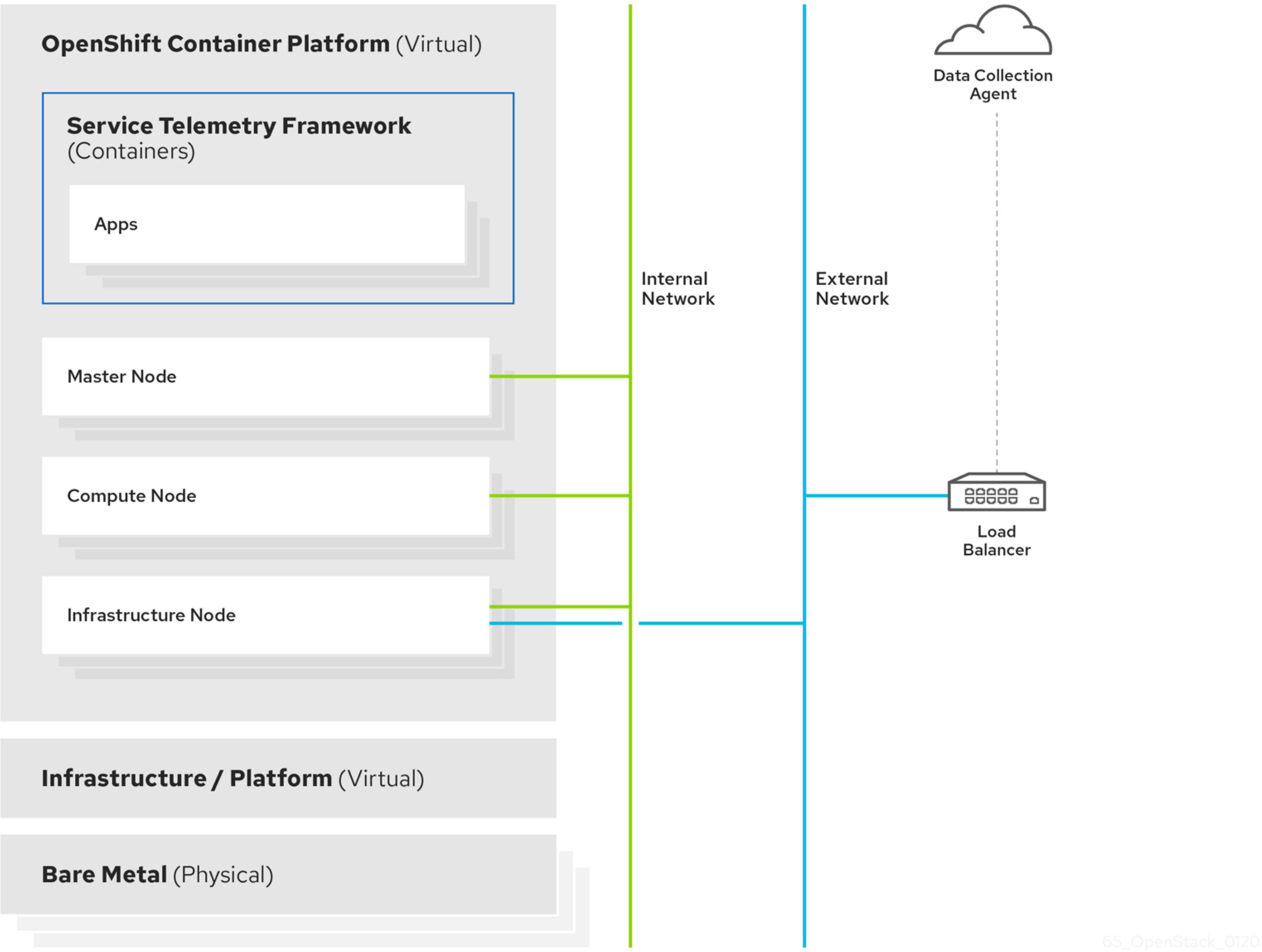
For client side metrics, collectd provides infrastructure metrics without project data, and Ceilometer provides RHOSP platform data based on projects or user workload. Both Ceilometer and collectd deliver data to Prometheus by using the AMQ Interconnect transport, delivering the data through the message bus. On the server side, a Golang application called the Smart Gateway takes the data stream from the bus and exposes it as a local scrape endpoint for Prometheus.

If you plan to collect and store events, collectd and Ceilometer deliver event data to the server side by using the AMQ Interconnect transport. Another Smart Gateway writes the data to the ElasticSearch datastore.

Server-side STF monitoring infrastructure consists of the following layers:

- Service Telemetry Framework 1.5
- Red Hat OpenShift Container Platform 4.10 through 4.10
- Infrastructure platform

Figure 1.2. Server-side STF monitoring infrastructure



1.3. INSTALLATION SIZE OF RED HAT OPENSHIFT CONTAINER PLATFORM

The size of your Red Hat OpenShift Container Platform installation depends on the following factors:

- The infrastructure that you select.
- The number of nodes that you want to monitor.
- The number of metrics that you want to collect.
- The resolution of metrics.
- The length of time that you want to store the data.

Installation of Service Telemetry Framework (STF) depends on an existing Red Hat OpenShift Container Platform environment.

For more information about minimum resources requirements when you install Red Hat OpenShift Container Platform on baremetal, see Minimum resource requirements in the *Installing a cluster on bare metal* guide. For installation requirements of the various public and private cloud platforms that you can install, see the corresponding installation documentation for your cloud platform of choice.

CHAPTER 2. PREPARING YOUR RED HAT OPENSHIFT CONTAINER PLATFORM ENVIRONMENT FOR SERVICE TELEMETRY FRAMEWORK

To prepare your Red Hat OpenShift Container Platform environment for Service Telemetry Framework (STF), you must plan for persistent storage, adequate resources, event storage, and network considerations:

- Ensure that you have persistent storage available in your Red Hat OpenShift Container Platform cluster for a production-grade deployment. For more information, see Section 2.2, “Persistent volumes”.
- Ensure that enough resources are available to run the Operators and the application containers. For more information, see Section 2.3, “Resource allocation”.
- Ensure that you have a fully connected network environment. For more information, see Section 2.4, “Network considerations for Service Telemetry Framework” .

2.1. OBSERVABILITY STRATEGY IN SERVICE TELEMETRY FRAMEWORK

Service Telemetry Framework (STF) does not include storage backends and alerting tools. STF uses community operators to deploy Prometheus, Alertmanager, Grafana, and Elasticsearch. STF makes requests to these community operators to create instances of each application configured to work with STF.

Instead of having Service Telemetry Operator create custom resource requests, you can use your own deployments of these applications or other compatible applications, and scrape the metrics Smart Gateways for delivery to your own Prometheus-compatible system for telemetry storage. If you set the observability strategy to use alternative backends instead, persistent or ephemeral storage is not required for STF.

2.2. PERSISTENT VOLUMES

Service Telemetry Framework (STF) uses persistent storage in Red Hat OpenShift Container Platform to request persistent volumes so that Prometheus and ElasticSearch can store metrics and events.

When you enable persistent storage through the Service Telemetry Operator, the Persistent Volume Claims (PVC) requested in an STF deployment results in an access mode of RWO (ReadWriteOnce). If your environment contains pre-provisioned persistent volumes, ensure that volumes of RWO are available in the Red Hat OpenShift Container Platform default configured **storageClass**.

Additional resources

- For more information about configuring persistent storage for Red Hat OpenShift Container Platform, see Understanding persistent storage.
- For more information about recommended configurable storage technology in Red Hat OpenShift Container Platform, see Recommended configurable storage technology .
- For more information about configuring persistent storage for Prometheus in STF, see the section called “Configuring persistent storage for Prometheus”.

- For more information about configuring persistent storage for ElasticSearch in STF, see the section called "Configuring persistent storage for ElasticSearch".

2.2.1. Ephemeral storage

You can use ephemeral storage to run Service Telemetry Framework (STF) without persistently storing data in your Red Hat OpenShift Container Platform cluster.

> ⚠ **WARNING**
>
> If you use ephemeral storage, you might experience data loss if a pod is restarted, updated, or rescheduled onto another node. Use ephemeral storage only for development or testing, and not production environments.

2.3. RESOURCE ALLOCATION

To enable the scheduling of pods within the Red Hat OpenShift Container Platform infrastructure, you need resources for the components that are running. If you do not allocate enough resources, pods remain in a **Pending** state because they cannot be scheduled.

The amount of resources that you require to run Service Telemetry Framework (STF) depends on your environment and the number of nodes and clouds that you want to monitor.

Additional resources

- For recommendations about sizing for metrics collection, see Service Telemetry Framework Performance and Scaling.
- For information about sizing requirements for ElasticSearch, see https://www.elastic.co/guide/en/cloud-on-k8s/current/k8s-managing-compute-resources.html.

2.4. NETWORK CONSIDERATIONS FOR SERVICE TELEMETRY FRAMEWORK

You can only deploy Service Telemetry Framework(STF) in a fully connected network environment. You cannot deploy STF in Red Hat OpenShift Container Platform-disconnected environments or network proxy environments.

CHAPTER 3. INSTALLING THE CORE COMPONENTS OF SERVICE TELEMETRY FRAMEWORK

You can use Operators to load the Service Telemetry Framework (STF) components and objects. Operators manage each of the following STF core and community components:

- AMQ Interconnect
- Smart Gateway
- Prometheus and AlertManager
- Elasticsearch
- Grafana

Prerequisites

- An Red Hat OpenShift Container Platform version inclusive of 4.10 through 4.10 is running.
- You have prepared your Red Hat OpenShift Container Platform environment and ensured that there is persistent storage and enough resources to run the STF components on top of the Red Hat OpenShift Container Platform environment. For more information, see Service Telemetry Framework Performance and Scaling.
- Your environment is fully connected. STF does not work in a Red Hat OpenShift Container Platform-disconnected environments or network proxy environments.

> **IMPORTANT**
>
> STF is compatible with Red Hat OpenShift Container Platform version 4.10 through 4.10.

Additional resources

- For more information about Operators, see the *Understanding Operators* guide.
- For more information about how to remove STF from the Red Hat OpenShift Container Platform environment, see Chapter 7, *Removing Service Telemetry Framework from the Red Hat OpenShift Container Platform environment*.

3.1. DEPLOYING SERVICE TELEMETRY FRAMEWORK TO THE RED HAT OPENSHIFT CONTAINER PLATFORM ENVIRONMENT

Deploy Service Telemetry Framework (STF) to collect, store, and monitor events:

Procedure

1. Create a namespace to contain the STF components, for example, **service-telemetry**:

```
$ oc new-project service-telemetry
```

2. Create an OperatorGroup in the namespace so that you can schedule the Operator pods:

```
$ oc create -f - <<EOF
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: service-telemetry-operator-group
  namespace: service-telemetry
spec:
  targetNamespaces:
  - service-telemetry
EOF
```

For more information, see OperatorGroups.

3. Enable the OperatorHub.io Community Catalog Source to install data storage and visualization Operators:

WARNING

Red Hat supports the core Operators and workloads, including AMQ Interconnect, AMQ Certificate Manager, Service Telemetry Operator, and Smart Gateway Operator. Red Hat does not support the community Operators or workload components, inclusive of ElasticSearch, Prometheus, Alertmanager, Grafana, and their Operators.

```
$ oc create -f - <<EOF
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: operatorhubio-operators
  namespace: openshift-marketplace
spec:
  sourceType: grpc
  image: quay.io/operatorhubio/catalog:latest
  displayName: OperatorHub.io Operators
  publisher: OperatorHub.io
EOF
```

4. Subscribe to the AMQ Certificate Manager Operator by using the redhat-operators CatalogSource:

NOTE

The AMQ Certificate Manager deploys to the **openshift-operators** namespace and is then available to all namespaces across the cluster. As a result, on clusters with a large number of namespaces, it can take several minutes for the Operator to be available in the **service-telemetry** namespace. The AMQ Certificate Manager Operator is not compatible with the dependency management of Operator Lifecycle Manager when you use it with other namespace-scoped operators.

```
$ oc create -f - <<EOF
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: amq7-cert-manager-operator
  namespace: openshift-operators
spec:
  channel: 1.x
  installPlanApproval: Automatic
  name: amq7-cert-manager-operator
  source: redhat-operators
  sourceNamespace: openshift-marketplace
EOF
```

5. Validate your ClusterServiceVersion. Ensure that amq7-cert-manager.v1.0.3 displays a phase of **Succeeded**:

```
$ oc get csv --namespace openshift-operators --selector operators.coreos.com/amq7-cert-
manager-operator.openshift-operators

NAME                    DISPLAY                                      VERSION   REPLACES
PHASE
amq7-cert-manager.v1.0.3   Red Hat Integration - AMQ Certificate Manager   1.0.3     amq7-
cert-manager.v1.0.2   Succeeded
```

6. Subscribe to the AMQ Interconnect Operator by using the redhat-operators CatalogSource:

```
$ oc create -f - <<EOF
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: amq7-interconnect-operator
  namespace: service-telemetry
spec:
  channel: 1.10.x
  installPlanApproval: Automatic
  name: amq7-interconnect-operator
  source: redhat-operators
  sourceNamespace: openshift-marketplace
EOF
```

7. Validate your ClusterServiceVersion. Ensure that amq7-interconnect-operator.v1.10.4 displays a phase of **Succeeded**:

```
$ oc get csv --selector=operators.coreos.com/amq7-interconnect-operator.service-telemetry

NAME                             DISPLAY                              VERSION   REPLACES
PHASE
amq7-interconnect-operator.v1.10.4   Red Hat Integration - AMQ Interconnect   1.10.4
amq7-interconnect-operator.v1.10.3   Succeeded
```

8. If you plan to store metrics in Prometheus, you must enable the Prometheus Operator. To enable the Prometheus Operator, create the following manifest in your Red Hat OpenShift Container Platform environment:

```
$ oc create -f - <<EOF
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: prometheus
  namespace: service-telemetry
spec:
  channel: beta
  installPlanApproval: Automatic
  name: prometheus
  source: operatorhubio-operators
  sourceNamespace: openshift-marketplace
EOF
```

9. Verify that the ClusterServiceVersion for Prometheus **Succeeded**:

```
$ oc get csv --selector=operators.coreos.com/prometheus.service-telemetry

NAME                        DISPLAY               VERSION   REPLACES                    PHASE
prometheusoperator.0.47.0   Prometheus Operator   0.47.0    prometheusoperator.0.37.0
Succeeded
```

10. If you plan to store events in ElasticSearch, you must enable the Elastic Cloud on Kubernetes (ECK) Operator. To enable the ECK Operator, create the following manifest in your Red Hat OpenShift Container Platform environment:

```
$ oc create -f - <<EOF
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: elasticsearch-eck-operator-certified
  namespace: service-telemetry
spec:
  channel: stable
  installPlanApproval: Automatic
  name: elasticsearch-eck-operator-certified
  source: certified-operators
  sourceNamespace: openshift-marketplace
EOF
```

11. Verify that the ClusterServiceVersion for Elastic Cloud on Kubernetes **Succeeded**:

```
$ oc get csv --selector=operators.coreos.com/elasticsearch-eck-operator-certified.service-telemetry

NAME                                         DISPLAY                        VERSION   REPLACES   PHASE
elasticsearch-eck-operator-certified.1.9.1   Elasticsearch (ECK) Operator   1.9.1
Succeeded
```

12. Create the Service Telemetry Operator subscription to manage the STF instances:

```
$ oc create -f - <<EOF
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
```

```
metadata:
  name: service-telemetry-operator
  namespace: service-telemetry
spec:
  channel: stable-1.4
  installPlanApproval: Automatic
  name: service-telemetry-operator
  source: redhat-operators
  sourceNamespace: openshift-marketplace
EOF
```

13. Validate the Service Telemetry Operator and the dependent operators:

```
$ oc get csv --namespace service-telemetry

NAME                                    DISPLAY                                   VERSION
REPLACES                            PHASE
amq7-cert-manager.v1.0.3                    Red Hat Integration - AMQ Certificate Manager   1.0.3
amq7-cert-manager.v1.0.2            Succeeded
amq7-interconnect-operator.v1.10.4          Red Hat Integration - AMQ Interconnect
1.10.4        amq7-interconnect-operator.v1.10.3   Succeeded
elasticsearch-eck-operator-certified.1.9.1   Elasticsearch (ECK) Operator                 1.9.1
Succeeded
prometheusoperator.0.47.0                   Prometheus Operator                          0.47.0
prometheusoperator.0.37.0           Succeeded
service-telemetry-operator.v1.4.1641489191   Service Telemetry Operator
1.4.1641489191                              Succeeded
smart-gateway-operator.v4.0.1641489202       Smart Gateway Operator
4.0.1641489202                              Succeeded
```

3.2. CREATING A SERVICETELEMETRY OBJECT IN RED HAT OPENSHIFT CONTAINER PLATFORM

Create a **ServiceTelemetry** object in Red Hat OpenShift Container Platform to result in the Service Telemetry Operator creating the supporting components for a Service Telemetry Framework (STF) deployment. For more information, see Section 3.2.1, “Primary parameters of the ServiceTelemetry object”.

Procedure

1. To create a **ServiceTelemetry** object that results in an STF deployment that uses the default values, create a **ServiceTelemetry** object with an empty **spec** parameter:

```
$ oc apply -f - <<EOF
apiVersion: infra.watch/v1beta1
kind: ServiceTelemetry
metadata:
  name: default
  namespace: service-telemetry
spec: {}
EOF
```

To override a default value, define the parameter that you want to override. In this example, enable ElasticSearch by setting **enabled** to **true**:

```
$ oc apply -f - <<EOF
apiVersion: infra.watch/v1beta1
kind: ServiceTelemetry
metadata:
  name: default
  namespace: service-telemetry
spec:
  backends:
    events:
      elasticsearch:
        enabled: true
EOF
```

Creating a **ServiceTelemetry** object with an empty **spec** parameter results in an STF deployment with the following default settings:

```
apiVersion: infra.watch/v1beta1
kind: ServiceTelemetry
metadata:
  name: default
  namespace: service-telemetry
spec:
  alerting:
    alertmanager:
      receivers:
        snmpTraps:
          enabled: false
          target: 192.168.24.254
      storage:
        persistent:
          pvcStorageRequest: 20G
        strategy: persistent
    enabled: true
  backends:
    events:
      elasticsearch:
        enabled: false
        storage:
          persistent:
            pvcStorageRequest: 20Gi
          strategy: persistent
        version: 7.16.1
    logs:
      loki:
        enabled: false
        flavor: 1x.extra-small
        replicationFactor: 1
        storage:
          objectStorageSecret: test
          storageClass: standard
    metrics:
      prometheus:
        enabled: true
        scrapeInterval: 10s
        storage:
          persistent:
            pvcStorageRequest: 20G
          retention: 24h
          strategy: persistent
  clouds:
  - events:
      collectors:
      - collectorType: collectd
        debugEnabled: false
        subscriptionAddress: collectd/cloud1-notify
      - collectorType: ceilometer
        debugEnabled: false
        subscriptionAddress: anycast/ceilometer/cloud1-event.sample
    metrics:
      collectors:
      - collectorType: collectd
        debugEnabled: false
        subscriptionAddress: collectd/cloud1-telemetry
      - collectorType: ceilometer
        debugEnabled: false
        subscriptionAddress: anycast/ceilometer/cloud1-metering.sample
      - collectorType: sensubility
        debugEnabled: false
        subscriptionAddress: sensubility/cloud1-telemetry
    name: cloud1
  graphing:
    enabled: false
    grafana:
      adminPassword: secret
      adminUser: root
      baseImage: docker.io/grafana/grafana:latest
      disableSignoutMenu: false
      ingressEnabled: false
  highAvailability:
    enabled: false
  observabilityStrategy: use_community
  transports:
    qdr:
      enabled: true
      web:
        enabled: false
```

To override these defaults, add the configuration to the **spec** parameter.

2. View the STF deployment logs in the Service Telemetry Operator:

```
$ oc logs --selector name=service-telemetry-operator

...
--------------------------- Ansible Task Status Event StdOut  -----------------

PLAY RECAP *********************************************************************
localhost                  : ok=57   changed=0    unreachable=0    failed=0    skipped=20
rescued=0    ignored=0
```

Verification

Verification

- To determine that all workloads are operating correctly, view the pods and the status of each pod.

  NOTE

  If you set the **backends.events.elasticsearch.enabled** parameter to **true**, the notification Smart Gateways report **Error** and **CrashLoopBackOff** error messages for a period of time before ElasticSearch starts.

  ```
  $ oc get pods

  NAME                                                      READY   STATUS    RESTARTS   AGE
  alertmanager-default-0                                    2/2     Running   0          17m
  default-cloud1-ceil-meter-smartgateway-6484b98b68-vd48z   2/2     Running   0          17m
  default-cloud1-coll-meter-smartgateway-799f687658-4gxpn   2/2     Running   0          17m
  default-cloud1-sens-meter-smartgateway-c7f4f7fc8-c57b4    2/2     Running   0          17m
  default-interconnect-54658f5d4-pzrpt                      1/1     Running   0          17m
  elastic-operator-66b7bc49c4-sxkc2                         1/1     Running   0          52m
  interconnect-operator-69df6b9cb6-7hhp9                    1/1     Running   0          50m
  prometheus-default-0                                      2/2     Running   1          17m
  prometheus-operator-6458b74d86-wbdqp                      1/1     Running   0          51m
  service-telemetry-operator-864646787c-hd9pm               1/1     Running   0          51m
  smart-gateway-operator-79778cf548-mz5z7                   1/1     Running   0          51m
  ```

3.2.1. Primary parameters of the ServiceTelemetry object

The **ServiceTelemetry** object comprises the following primary configuration parameters:

- **alerting**
- **backends**
- **clouds**
- **graphing**
- **highAvailability**
- **transports**

You can configure each of these configuration parameters to provide different features in an STF deployment.

IMPORTANT

Support for **servicetelemetry.infra.watch/v1alpha1** was removed from STF 1.3.

The backends parameter

Use the **backends** parameter to control which storage back ends are available for storage of metrics and events, and to control the enablement of Smart Gateways that the **clouds** parameter defines. For more information, see the section called "The clouds parameter" .

Currently, you can use Prometheus as the metrics storage back end and ElasticSearch as the events storage back end.

Enabling Prometheus as a storage back end for metrics

To enable Prometheus as a storage back end for metrics, you must configure the **ServiceTelemetry** object.

Procedure

- Configure the **ServiceTelemetry** object:

```
apiVersion: infra.watch/v1beta1
kind: ServiceTelemetry
metadata:
  name: default
  namespace: service-telemetry
spec:
  backends:
    metrics:
      prometheus:
        enabled: true
```

Configuring persistent storage for Prometheus

Use the additional parameters that are defined in **backends.metrics.prometheus.storage.persistent** to configure persistent storage options for Prometheus, such as storage class and volume size.

Use **storageClass** to define the back end storage class. If you do not set this parameter, the Service Telemetry Operator uses the default storage class for the Red Hat OpenShift Container Platform cluster.

Use the **pvcStorageRequest** parameter to define the minimum required volume size to satisfy the storage request. If volumes are statically defined, it is possible that a volume size larger than requested is used. By default, Service Telemetry Operator requests a volume size of **20G** (20 Gigabytes).

Procedure

- List the available storage classes:

```
$ oc get storageclasses
NAME                 PROVISIONER               RECLAIMPOLICY   VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION   AGE
csi-manila-ceph      manila.csi.openstack.org   Delete          Immediate              false
20h
standard (default)   kubernetes.io/cinder       Delete          WaitForFirstConsumer   true
20h
standard-csi         cinder.csi.openstack.org   Delete          WaitForFirstConsumer   true
20h
```

- Configure the **ServiceTelemetry** object:

```
apiVersion: infra.watch/v1beta1
kind: ServiceTelemetry
metadata:
  name: default
  namespace: service-telemetry
```

```
spec:
  backends:
    metrics:
      prometheus:
        enabled: true
        storage:
          strategy: persistent
          persistent:
            storageClass: standard-csi
            pvcStorageRequest: 50G
```

Enabling ElasticSearch as a storage back end for events

To enable ElasticSearch as a storage back end for events, you must configure the **ServiceTelemetry** object.

Procedure

- Configure the **ServiceTelemetry** object:

```
apiVersion: infra.watch/v1beta1
kind: ServiceTelemetry
metadata:
  name: default
  namespace: service-telemetry
spec:
  backends:
    events:
      elasticsearch:
        enabled: true
```

Configuring persistent storage for ElasticSearch

Use the additional parameters defined in **backends.events.elasticsearch.storage.persistent** to configure persistent storage options for ElasticSearch, such as storage class and volume size.

Use **storageClass** to define the back end storage class. If you do not set this parameter, the Service Telemetry Operator uses the default storage class for the Red Hat OpenShift Container Platform cluster.

Use the **pvcStorageRequest** parameter to define the minimum required volume size to satisfy the storage request. If volumes are statically defined, it is possible that a volume size larger than requested is used. By default, Service Telemetry Operator requests a volume size of **20Gi** (20 Gibibytes).

Procedure

- List the available storage classes:

```
$ oc get storageclasses
NAME               PROVISIONER            RECLAIMPOLICY   VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION   AGE
csi-manila-ceph      manila.csi.openstack.org   Delete         Immediate              false
20h
standard (default)   kubernetes.io/cinder       Delete         WaitForFirstConsumer   true
20h
standard-csi         cinder.csi.openstack.org   Delete         WaitForFirstConsumer   true
20h
```

- Configure the **ServiceTelemetry** object:

```
apiVersion: infra.watch/v1beta1
kind: ServiceTelemetry
metadata:
  name: default
  namespace: service-telemetry
spec:
  backends:
    events:
      elasticsearch:
        enabled: true
        version: 7.16.1
        storage:
          strategy: persistent
          persistent:
            storageClass: standard-csi
            pvcStorageRequest: 50G
```

The clouds parameter
Use the **clouds** parameter to define which Smart Gateway objects deploy, thereby providing the interface for multiple monitored cloud environments to connect to an instance of STF. If a supporting back end is available, then metrics and events Smart Gateways for the default cloud configuration are created. By default, the Service Telemetry Operator creates Smart Gateways for **cloud1**.

You can create a list of cloud objects to control which Smart Gateways are created for the defined clouds. Each cloud consists of data types and collectors. Data types are **metrics** or **events**. Each data type consists of a list of collectors, the message bus subscription address, and a parameter to enable debugging. Available collectors for metrics are **collectd**, **ceilometer**, and **sensubility**. Available collectors for events are **collectd** and **ceilometer**. Ensure that the subscription address for each of these collectors is unique for every cloud, data type, and collector combination.

The default **cloud1** configuration is represented by the following **ServiceTelemetry** object, which provides subscriptions and data storage of metrics and events for collectd, Ceilometer, and Sensubility data collectors for a particular cloud instance:

```
apiVersion: infra.watch/v1beta1
kind: ServiceTelemetry
metadata:
  name: stf-default
  namespace: service-telemetry
spec:
  clouds:
    - name: cloud1
      metrics:
        collectors:
          - collectorType: collectd
            subscriptionAddress: collectd/telemetry
          - collectorType: ceilometer
            subscriptionAddress: anycast/ceilometer/metering.sample
          - collectorType: sensubility
            subscriptionAddress: sensubility/telemetry
            debugEnabled: false
      events:
        collectors:
          - collectorType: collectd
      subscriptionAddress: collectd/notify
    - collectorType: ceilometer
      subscriptionAddress: anycast/ceilometer/event.sample
```

Each item of the **clouds** parameter represents a cloud instance. A cloud instance consists of three top-level parameters: **name**, **metrics**, and **events**. The **metrics** and **events** parameters represent the corresponding back end for storage of that data type. The **collectors** parameter specifies a list of objects made up of two required parameters, **collectorType** and **subscriptionAddress**, and these represent an instance of the Smart Gateway. The **collectorType** parameter specifies data collected by either collectd, Ceilometer, or Sensubility. The **subscriptionAddress** parameter provides the AMQ Interconnect address to which a Smart Gateway subscribes.

You can use the optional Boolean parameter **debugEnabled** within the **collectors** parameter to enable additional console debugging in the running Smart Gateway pod.

Additional resources

- For more information about deleting default Smart Gateways, see Section 4.4.3, "Deleting the default Smart Gateways".
- For more information about how to configure multiple clouds, see Section 4.4, "Configuring multiple clouds".

The alerting parameter

Use the **alerting** parameter to control creation of an Alertmanager instance and the configuration of the storage back end. By default, **alerting** is enabled. For more information, see Section 5.3, "Alerts in Service Telemetry Framework".

The graphing parameter

Use the **graphing** parameter to control the creation of a Grafana instance. By default, **graphing** is disabled. For more information, see Section 5.1, "Dashboards in Service Telemetry Framework" .

The highAvailability parameter

Use the **highAvailability** parameter to control the instantiation of multiple copies of STF components to reduce recovery time of components that fail or are rescheduled. By default, **highAvailability** is disabled. For more information, see Section 5.5, "High availability".

The transports parameter

Use the **transports** parameter to control the enablement of the message bus for a STF deployment. The only transport currently supported is AMQ Interconnect. By default, the **qdr** transport is enabled.

3.3. ACCESSING USER INTERFACES FOR STF COMPONENTS

In Red Hat OpenShift Container Platform, applications are exposed to the external network through a route. For more information about routes, see Configuring ingress cluster traffic.

In Service Telemetry Framework (STF), HTTPS routes are exposed for each service that has a web-based interface. These routes are protected by Red Hat OpenShift Container Platform RBAC and any user that has a **ClusterRoleBinding** that enables them to view Red Hat OpenShift Container Platform Namespaces can log in. For more information about RBAC, see Using RBAC to define and apply permissions.

Procedure

1. Log in to Red Hat OpenShift Container Platform.

2. Change to the **service-telemetry** namespace:

```
$ oc project service-telemetry
```

3. List the available web UI routes in the **service-telemetry** project:

```
$ oc get routes | grep web
default-alertmanager-proxy   default-alertmanager-proxy-service-telemetry.apps.infra.watch
default-alertmanager-proxy   web     reencrypt/Redirect   None
default-prometheus-proxy     default-prometheus-proxy-service-telemetry.apps.infra.watch
default-prometheus-proxy     web     reencrypt/Redirect   None
```

4. In a web browser, navigate to https://<route_address> to access the web interface for the corresponding service.

3.4. CONFIGURING AN ALTERNATE OBSERVABILITY STRATEGY

To configure STF to skip the deployment of storage, visualization, and alerting backends, add **observabilityStrategy: none** to the ServiceTelemetry spec. In this mode, only AMQ Interconnect routers and metrics Smart Gateways are deployed, and you must configure an external Prometheus-compatible system to collect metrics from the STF Smart Gateways.

NOTE

Currently, only metrics are supported when you set **observabilityStrategy** to **none**. Events Smart Gateways are not deployed.

Procedure

1. Create a **ServiceTelemetry** object with the property **observabilityStrategy: none** in the **spec** parameter. The manifest shows results in a default deployment of STF that is suitable for receiving telemetry from a single cloud with all metrics collector types.

```
$ oc apply -f - <<EOF
apiVersion: infra.watch/v1beta1
kind: ServiceTelemetry
metadata:
  name: default
  namespace: service-telemetry
spec:
  observabilityStrategy: none
EOF
```

2. To verify that all workloads are operating correctly, view the pods and the status of each pod:

```
$ oc get pods
NAME                                                      READY   STATUS    RESTARTS   AGE
default-cloud1-ceil-meter-smartgateway-59c845d65b-gzhcs   3/3     Running   0          132m
default-cloud1-coll-meter-smartgateway-75bbd948b9-d5phm   3/3     Running   0          132m
default-cloud1-sens-meter-smartgateway-7fdbb57b6d-dh2g9   3/3     Running   0          132m
default-interconnect-668d5bbcd6-57b2l                     1/1     Running   0          132m
interconnect-operator-b8f5bb647-tlp5t                     1/1     Running   0          47h
service-telemetry-operator-566b9dd695-wkvjq               1/1     Running   0          156m
smart-gateway-operator-58d77dcf7-6xsq7                    1/1     Running   0          47h
```

Additional resources

For more information about configuring additional clouds or to change the set of supported collectors, see Section 4.4.2, “Deploying Smart Gateways”

CHAPTER 4. CONFIGURING RED HAT OPENSTACK PLATFORM FOR SERVICE TELEMETRY FRAMEWORK

To collect metrics, events, or both, and to send them to the Service Telemetry Framework (STF) storage domain, you must configure the Red Hat OpenStack Platform (RHOSP) overcloud to enable data collection and transport.

STF can support both single and multiple clouds. The default configuration in RHOSP and STF set up for a single cloud installation.

- For a single RHOSP overcloud deployment with default configuration, see Section 4.1, "Deploying Red Hat OpenStack Platform overcloud for Service Telemetry Framework".
- To plan your RHOSP installation and configuration STF for multiple clouds, see Section 4.4, "Configuring multiple clouds".
- As part of an RHOSP overcloud deployment, you might need to configure additional features in your environment:
  - To deploy data collection and transport to STF on RHOSP cloud nodes that employ routed L3 domains, such as distributed compute node (DCN) or spine-leaf, see Section 4.3, "Deploying to non-standard network topologies".
  - To send metrics to both Gnocchi and STF, see Section 4.2, "Sending metrics to Gnocchi and Service Telemetry Framework".

4.1. DEPLOYING RED HAT OPENSTACK PLATFORM OVERCLOUD FOR SERVICE TELEMETRY FRAMEWORK

As part of the Red Hat OpenStack Platform (RHOSP) overcloud deployment, you must configure the data collectors and the data transport to Service Telemetry Framework (STF).

Procedure

1. Retrieving the AMQ Interconnect route address
2. Creating the base configuration for STF
3. Configuring the STF connection for the overcloud
4. Deploying the overcloud
5. Validating client-side installation

Additional resources

- the amqp1 plug-in.

4.1.1. Retrieving the AMQ Interconnect route address

When you configure the Red Hat OpenStack Platform (RHOSP) overcloud for Service Telemetry Framework (STF), you must provide the AMQ Interconnect route address in the STF connection file.

Procedure

1. Log in to your Red Hat OpenShift Container Platform environment.
2. In the **service-telemetry** project, retrieve the AMQ Interconnect route address:

```
$ oc get routes -ogo-template='{{ range .items }}{{printf "%s\n" .spec.host }}{{ end }}' | grep "\-5671"
default-interconnect-5671-service-telemetry.apps.infra.watch
```

4.1.2. Creating the base configuration for STF

To configure the base parameters to provide a compatible data collection and transport for Service Telemetry Framework (STF), you must create a file that defines the default data collection values.

Procedure

1. Log in to the Red Hat OpenStack Platform (RHOSP) undercloud as the **stack** user.
2. Create a configuration file called **enable-stf.yaml** in the /**home**/**stack** directory.

> **IMPORTANT**
>
> Setting **EventPipelinePublishers** and **PipelinePublishers** to empty lists results in no event or metric data passing to RHOSP telemetry components, such as Gnocchi or Panko. If you need to send data to additional pipelines, the Ceilometer polling interval of 30 seconds, as specified in **ExtraConfig**, might overwhelm the RHOSP telemetry components, and you must increase the interval to a larger value, such as **300**. Increasing the value to a longer polling interval results in less telemetry resolution in STF.
>
> To enable collection of telemetry with STF and Gnocchi, see Section 4.2, “Sending metrics to Gnocchi and Service Telemetry Framework”

enable-stf.yaml

```
parameter_defaults:
    # only send to STF, not other publishers
    EventPipelinePublishers: []
    PipelinePublishers: []

    # manage the polling and pipeline configuration files for Ceilometer agents
    ManagePolling: true
    ManagePipeline: true

    # enable Ceilometer metrics and events
    CeilometerQdrPublishMetrics: true
    CeilometerQdrPublishEvents: true

    # enable collection of API status
    CollectdEnableSensubility: true
    CollectdSensubilityTransport: amqp1
```

```
# enable collection of containerized service metrics
CollectdEnableLibpodstats: true

# set collectd overrides for higher telemetry resolution and extra plugins
# to load
CollectdConnectionType: amqp1
CollectdAmqpInterval: 5
CollectdDefaultPollingInterval: 5
CollectdExtraPlugins:
- vmem

# set standard prefixes for where metrics and events are published to QDR
MetricsQdrAddresses:
- prefix: 'collectd'
  distribution: multicast
- prefix: 'anycast/ceilometer'
  distribution: multicast

ExtraConfig:
    ceilometer::agent::polling::polling_interval: 30
    ceilometer::agent::polling::polling_meters:
    - cpu
    - disk.*
    - ip.*
    - image.*
    - memory
    - memory.*
    - network.*
    - perf.*
    - port
    - port.*
    - switch
    - switch.*
    - storage.*
    - volume.*

    # to avoid filling the memory buffers if disconnected from the message bus
    # note: this may need an adjustment if there are many metrics to be sent.
    collectd::plugin::amqp1::send_queue_limit: 5000

    # receive extra information about virtual memory
    collectd::plugin::vmem::verbose: true

    # provide name and uuid in addition to hostname for better correlation
    # to ceilometer data
    collectd::plugin::virt::hostname_format: "name uuid hostname"

    # provide the human-friendly name of the virtual instance
    collectd::plugin::virt::plugin_instance_format: metadata

    # set memcached collectd plugin to report its metrics by hostname
    # rather than host IP, ensuring metrics in the dashboard remain uniform
    collectd::plugin::memcached::instances:
      local:
        host: "%{hiera('fqdn_canonical')}"
        port: 11211
```

4.1.3. Configuring the STF connection for the overcloud

To configure the Service Telemetry Framework (STF) connection, you must create a file that contains the connection configuration of the AMQ Interconnect for the overcloud to the STF deployment. Enable the collection of events and storage of the events in STF and deploy the overcloud. The default configuration is for a single cloud instance with the default message bus topics. For configuration of multiple cloud deployments, see Section 4.4, “Configuring multiple clouds”.

Prerequisites

- Retrieve the AMQ Interconnect route address. For more information, see Section 4.1.1, “Retrieving the AMQ Interconnect route address”.

Procedure

1. Log in to the RHOSP undercloud as the **stack** user.
2. Create a configuration file called **stf-connectors.yaml** in the **/home/stack** directory.
3. In the **stf-connectors.yaml** file, configure the **MetricsQdrConnectors** address to connect the AMQ Interconnect on the overcloud to the STF deployment. You configure the topic addresses for Sensubility, Ceilometer, and collectd in this file to match the defaults in STF. For more information about customizing topics and cloud configuration, see Section 4.4, “Configuring multiple clouds”.
   - Replace the **host** parameter with the value of **HOST/PORT** that you retrieved in Section 4.1.1, “Retrieving the AMQ Interconnect route address” .

   stf-connectors.yaml

```
resource_registry:
  OS::TripleO::Services::Collectd: /usr/share/openstack-tripleo-heat-templates/deployment/metrics/collectd-container-puppet.yaml ❶

parameter_defaults:
    MetricsQdrConnectors:
        - host: stf-default-interconnect-5671-service-telemetry.apps.infra.watch ❷
          port: 443
          role: edge
          verifyHostname: false
          sslProfile: sslProfile

    MetricsQdrSSLProfiles:
        - name: sslProfile

    CeilometerQdrEventsConfig:
        driver: amqp
        topic: cloud1-event ❸

    CeilometerQdrMetricsConfig:
        driver: amqp
        topic: cloud1-metering ❹

    CollectdAmqpInstances:
        cloud1-notify: ❺
```

```
      notify: true
      format: JSON
      presettle: false
    cloud1-telemetry:   6
      format: JSON
      presettle: false

  CollectdSensubilityResultsChannel: sensubility/cloud1-telemetry 7
```

1. Directly load the collectd service because you are not including the **collectd-write-qdr.yaml** environment file for multiple cloud deployments.

2. Replace the **host** parameter with the value of **HOST/PORT** that you retrieved in Section 4.1.1, “Retrieving the AMQ Interconnect route address” .

3. Define the topic for Ceilometer events. The format of this value is **anycast/ceilometer/cloud1-event.sample**.

4. Define the topic for Ceilometer metrics. The format of this value is`anycast/ceilometer/cloud1-metering.sample`.

5. Define the topic for collectd events. The format of this value is **collectd/cloud1-notify**.

6. Define the topic for collectd metrics. The format of this value is **collectd/cloud1-telemetry**.

7. Define the topic for collectd-sensubility events. The value is the exact string **sensubility/cloud1-telemetry**.

4.1.4. Deploying the overcloud

Deploy or update the overcloud with the required environment files so that data is collected and transmitted to Service Telemetry Framework (STF).

Procedure

1. Log in to the Red Hat OpenStack Platform (RHOSP) undercloud as the **stack** user.

2. Source the authentication file:

```
[stack@undercloud-0 ~]$ source stackrc

(undercloud) [stack@undercloud-0 ~]$
```

3. Add the following files to your RHOSP director deployment to configure data collection and AMQ Interconnect:

   - The **ceilometer-write-qdr.yaml** file to ensure that Ceilometer telemetry and events are sent to STF

   - The **qdr-edge-only.yaml** file to ensure that the message bus is enabled and connected to STF message bus routers

   - The **enable-stf.yaml** environment file to ensure defaults are configured correctly

- The **stf-connectors.yaml** environment file to define the connection to STF

4. Deploy the RHOSP overcloud:

```
(undercloud) [stack@undercloud-0 ~]$ openstack overcloud deploy <other_arguments>
--templates /usr/share/openstack-tripleo-heat-templates \
  --environment-file <...other_environment_files...> \
  --environment-file /usr/share/openstack-tripleo-heat-
templates/environments/metrics/ceilometer-write-qdr.yaml \
  --environment-file /usr/share/openstack-tripleo-heat-templates/environments/metrics/qdr-
edge-only.yaml \
  --environment-file /home/stack/enable-stf.yaml \
  --environment-file /home/stack/stf-connectors.yaml
```

4.1.5. Validating client-side installation

To validate data collection from the Service Telemetry Framework (STF) storage domain, query the data sources for delivered data. To validate individual nodes in the Red Hat OpenStack Platform (RHOSP) deployment, use SSH to connect to the console.

TIP

Some telemetry data is available only when RHOSP has active workloads.

Procedure

1. Log in to an overcloud node, for example, controller-0.

2. Ensure that the **metrics_qdr** container is running on the node:

```
$ sudo podman container inspect --format '{{.State.Status}}' metrics_qdr

running
```

3. Return the internal network address on which AMQ Interconnect is running, for example, **172.17.1.44** listening on port **5666**:

```
$ sudo podman exec -it metrics_qdr cat /etc/qpid-dispatch/qdrouterd.conf

listener {
    host: 172.17.1.44
    port: 5666
    authenticatePeer: no
    saslMechanisms: ANONYMOUS
}
```

4. Return a list of connections to the local AMQ Interconnect:

```
$ sudo podman exec -it metrics_qdr qdstat --bus=172.17.1.44:5666 --connections

Connections
  id   host                                                  container
role    dir  security                        authentication  tenant
```

```
==========================================================================
==========================================================================
==========================================================================
============================================================
  1    default-interconnect-5671-service-telemetry.apps.infra.watch:443      default-
interconnect-7458fd4d69-bgzfb                                              edge    out
TLSv1.2(DHE-RSA-AES256-GCM-SHA384)  anonymous-user
  12   172.17.1.44:60290
openstack.org/om/container/controller-0/ceilometer-agent-
notification/25/5c02cee550f143ec9ea030db5cccba14  normal  in   no-security
no-auth
  16   172.17.1.44:36408                                           metrics
normal  in   no-security                    anonymous-user
  899  172.17.1.44:39500                                           10a2e99d-1b8a-4329-b48c-
4335e5f75c84                                                       normal  in   no-security
no-auth
```

There are four connections:

- Outbound connection to STF
- Inbound connection from ceilometer
- Inbound connection from collectd
- Inbound connection from our **qdstat** client
  The outbound STF connection is provided to the **MetricsQdrConnectors** host parameter and is the route for the STF storage domain. The other hosts are internal network addresses of the client connections to this AMQ Interconnect.

5. To ensure that messages are delivered, list the links, and view the **_edge** address in the **deliv** column for delivery of messages:

```
$ sudo podman exec -it metrics_qdr qdstat --bus=172.17.1.44:5666 --links
Router Links
  type      dir  conn id  id    peer  class   addr              phs  cap  pri  undel  unsett  deliv
presett  psdrop  acc  rej  rel     mod  delay  rate

==========================================================================
==========================================================================
=====
  endpoint  out  1       5          local   _edge                 250  0     0      0      2979926  0       0
0    0    2979926 0     0      0
  endpoint  in   1       6                                    250  0     0      0      0       0       0     0
0       0    0      0
  endpoint  in   1       7                                    250  0     0      0      0       0       0     0
0       0    0      0
  endpoint  out  1       8                                     250  0     0      0      0       0       0     0
0       0    0      0
  endpoint  in   1       9                                    250  0     0      0      0       0       0     0
0       0    0      0
  endpoint  out  1       10                                    250  0     0      0      911      911      0       0
0    0       0    911     0
  endpoint  in   1       11                                    250  0     0      0      0       911      0       0
0    0       0    0      0
  endpoint  out  12      32          local   temp.ISY6Mcicol4J2Kp      250  0     0      0      0       0
```

```
0      0    0    0       0    0      0
  endpoint  in   16       41                            250  0     0      0       2979924  0        0
0    0    2979924 0    0      0
  endpoint  in   912      1834        mobile  $management          0    250  0     0      0        1        0
0      1    0    0       0    0      0
  endpoint  out  912      1835        local   temp.9Ok2resI9tmt+CT       250  0     0      0        0
0        0       0    0    0       0    0       0
```

6. To list the addresses from RHOSP nodes to STF, connect to Red Hat OpenShift Container Platform to retrieve the AMQ Interconnect pod name and list the connections. List the available AMQ Interconnect pods:

```
$ oc get pods -l application=default-interconnect

NAME                                    READY   STATUS    RESTARTS   AGE
default-interconnect-7458fd4d69-bgzfb   1/1     Running   0          6d21h
```

7. Connect to the pod and list the known connections. In this example, there are three **edge** connections from the RHOSP nodes with connection **id** 22, 23, and 24:

```
$ oc exec -it default-interconnect-7458fd4d69-bgzfb -- qdstat --connections

2020-04-21 18:25:47.243852 UTC
default-interconnect-7458fd4d69-bgzfb

Connections
  id  host              container                                              role    dir  security
authentication  tenant  last dlv      uptime

=================================================================================================
=================================================================================================
==========================================
  5   10.129.0.110:48498  bridge-3f5                                              edge    in   no-security
anonymous-user          000:00:00:02  000:17:36:29
  6   10.129.0.111:43254  rcv[default-cloud1-ceil-meter-smartgateway-58f885c76d-xmxwn]
edge    in   no-security                         anonymous-user          000:00:00:02  000:17:36:20
  7   10.130.0.109:50518  rcv[default-cloud1-coll-event-smartgateway-58fbbd4485-rl9bd]
normal  in   no-security                         anonymous-user          -             000:17:36:11
  8   10.130.0.110:33802  rcv[default-cloud1-ceil-event-smartgateway-6cfb65478c-g5q82]
normal  in   no-security                         anonymous-user          000:01:26:18  000:17:36:05
  22  10.128.0.1:51948   Router.ceph-0.redhat.local                               edge    in
TLSv1/SSLv3(DHE-RSA-AES256-GCM-SHA384)  anonymous-user          000:00:00:03
000:22:08:43
  23  10.128.0.1:51950   Router.compute-0.redhat.local                               edge    in
TLSv1/SSLv3(DHE-RSA-AES256-GCM-SHA384)  anonymous-user          000:00:00:03
000:22:08:43
  24  10.128.0.1:52082   Router.controller-0.redhat.local                            edge    in
TLSv1/SSLv3(DHE-RSA-AES256-GCM-SHA384)  anonymous-user          000:00:00:00
000:22:08:34
  27  127.0.0.1:42202    c2f541c1-4c97-4b37-a189-a396c08fb079                             normal  in
no-security                      no-auth                000:00:00:00  000:00:00:00
```

8. To view the number of messages delivered by the network, use each address with the **oc exec** command:

```
$ oc exec -it default-interconnect-7458fd4d69-bgzfb -- qdstat --address

2020-04-21 18:20:10.293258 UTC
default-interconnect-7458fd4d69-bgzfb

Router Addresses
  class   addr                           phs  distrib    pri  local  remote  in         out        thru
fallback

=================================================================================
====================================================
  mobile  anycast/ceilometer/event.sample     0   balanced   -    1     0      970        970
0     0
  mobile  anycast/ceilometer/metering.sample  0   balanced   -    1     0      2,344,833
2,344,833    0     0
  mobile  collectd/notify                     0    multicast  -    1     0      70         70          0     0
  mobile  collectd/telemetry                  0    multicast  -    1     0      216,128,890  216,128,890
0     0
```

4.2. SENDING METRICS TO GNOCCHI AND SERVICE TELEMETRY FRAMEWORK

To send metrics to Service Telemetry Framework (STF) and Gnocchi simultaneously, you must include an environment file in your deployment to enable an additional publisher.

> **WARNING**
>
> If you need to send data to additional pipelines, the Ceilometer polling interval of 30 seconds, as specified in **ExtraConfig**, might overwhelm the RHOSP telemetry components, and you must increase the interval to a larger value, such as **300**. Increasing the value to a longer polling interval results in less telemetry resolution in STF.

Prerequisites

- You have created a file that contains the connection configuration of the AMQ Interconnect for the overcloud to STF. For more information, see Section 4.1.3, “Configuring the STF connection for the overcloud”.

Procedure

1. Create an environment file named **gnocchi-connectors.yaml** in the **/home/stack** directory.

```
resource_registry:
    OS::TripleO::Services::GnocchiApi: /usr/share/openstack-tripleo-heat-
templates/deployment/gnocchi/gnocchi-api-container-puppet.yaml
    OS::TripleO::Services::GnocchiMetricd: /usr/share/openstack-tripleo-heat-
templates/deployment/gnocchi/gnocchi-metricd-container-puppet.yaml
    OS::TripleO::Services::GnocchiStatsd: /usr/share/openstack-tripleo-heat-
```

```
templates/deployment/gnocchi/gnocchi-statsd-container-puppet.yaml
    OS::TripleO::Services::AodhApi: /usr/share/openstack-tripleo-heat-
templates/deployment/aodh/aodh-api-container-puppet.yaml
    OS::TripleO::Services::AodhEvaluator: /usr/share/openstack-tripleo-heat-
templates/deployment/aodh/aodh-evaluator-container-puppet.yaml
    OS::TripleO::Services::AodhNotifier: /usr/share/openstack-tripleo-heat-
templates/deployment/aodh/aodh-notifier-container-puppet.yaml
    OS::TripleO::Services::AodhListener: /usr/share/openstack-tripleo-heat-
templates/deployment/aodh/aodh-listener-container-puppet.yaml

parameter_defaults:
    CeilometerEnableGnocchi: true
    CeilometerEnablePanko: false
    GnocchiArchivePolicy: 'high'
    GnocchiBackend: 'rbd'
    GnocchiRbdPoolName: 'metrics'

    EventPipelinePublishers: ['gnocchi://?filter_project=service']
    PipelinePublishers: ['gnocchi://?filter_project=service']
```

2. Add the environment file **gnocchi-connectors.yaml** to the deployment command. Replace *<other_arguments>* with files that are applicable to your environment.

```
$ openstack overcloud deploy <other_arguments>
--templates /usr/share/openstack-tripleo-heat-templates \
  --environment-file <...other_environment_files...> \
  --environment-file /usr/share/openstack-tripleo-heat-
templates/environments/metrics/ceilometer-write-qdr.yaml \
  --environment-file /usr/share/openstack-tripleo-heat-
templates/environments/metrics/collectd-write-qdr.yaml \
  --environment-file /usr/share/openstack-tripleo-heat-templates/environments/metrics/qdr-
edge-only.yaml \
  --environment-file /home/stack/enable-stf.yaml \
  --environment-file /home/stack/stf-connectors.yaml \
  --environment-file /home/stack/gnocchi-connectors.yaml
```

3. To ensure that the configuration was successful, verify the content of the file **/var/lib/config-data/puppet-generated/ceilometer/etc/ceilometer/pipeline.yaml** on a Controller node. Ensure that the **publishers** section of the file contains information for both **notifier** and **Gnocchi**.

```
sources:
    - name: meter_source
      meters:
          - "*"
      sinks:
          - meter_sink
sinks:
    - name: meter_sink
      publishers:
          - gnocchi://?filter_project=service
          - notifier://172.17.1.35:5666/?driver=amqp&topic=metering
```

4.3. DEPLOYING TO NON-STANDARD NETWORK TOPOLOGIES

If your nodes are on a separate network from the default **InternalApi** network, you must make configuration adjustments so that AMQ Interconnect can transport data to the Service Telemetry Framework (STF) server instance. This scenario is typical in a spine-leaf or a DCN topology. For more information about DCN configuration, see the Spine Leaf Networking guide.

If you use STF with Red Hat OpenStack Platform (RHOSP) 16.2 and plan to monitor your Ceph, Block, or Object Storage nodes, you must make configuration changes that are similar to the configuration changes that you make to the spine-leaf and DCN network configuration. To monitor Ceph nodes, use the **CephStorageExtraConfig** parameter to define which network interface to load into the AMQ Interconnect and collectd configuration files.

```
CephStorageExtraConfig:
  tripleo::profile::base::metrics::collectd::amqp_host: "%{hiera('storage')}"
  tripleo::profile::base::metrics::qdr::listener_addr: "%{hiera('storage')}"
  tripleo::profile::base::ceilometer::agent::notification::notifier_host_addr: "%{hiera('storage')}"
```

Similarly, you must specify **BlockStorageExtraConfig** and **ObjectStorageExtraConfig** parameters if your environment uses Block and Object Storage roles.

To deploy a spine-leaf topology, you must create roles and networks, then assign those networks to the available roles. When you configure data collection and transport for STF for an RHOSP deployment, the default network for roles is **InternalApi**. For Ceph, Block and Object storage roles, the default network is **Storage**. Because a spine-leaf configuration can result in different networks being assigned to different Leaf groupings and those names are typically unique, additional configuration is required in the **parameter_defaults** section of the RHOSP environment files.

Procedure

1. Document which networks are available for each of the Leaf roles. For examples of network name definitions, see Creating a network data file in the *Spine Leaf Networking* guide. For more information about the creation of the Leaf groupings (roles) and assignment of the networks to those groupings, see Creating a roles data file in the *Spine Leaf Networking* guide.

2. Add the following configuration example to the **ExtraConfig** section for each of the leaf roles. In this example, **internal_api_subnet** is the value defined in the **name_lower** parameter of your network definition (with **_subnet** appended to the name for Leaf 0) , and is the network to which the **ComputeLeaf0** leaf role is connected. In this case, the network identification of 0 corresponds to the Compute role for leaf 0, and represents a value that is different from the default internal API network name.
   For the **ComputeLeaf0** leaf role, specify extra configuration to perform a hiera lookup to determine which network interface for a particular network to assign to the collectd AMQP host parameter. Perform the same configuration for the AMQ Interconnect listener address parameter.

   ```
   ComputeLeaf0ExtraConfig:
     tripleo::profile::base::metrics::collectd::amqp_host: "%{hiera('internal_api_subnet')}"
     tripleo::profile::base::metrics::qdr::listener_addr: "%{hiera('internal_api_subnet')}"
   ```

   Additional leaf roles typically replace **_subnet** with **_leafN**. **N** represents a unique identifier for the leaf.

   ```
   ComputeLeaf1ExtraConfig:
     tripleo::profile::base::metrics::collectd::amqp_host: "%{hiera('internal_api_leaf1')}"
     tripleo::profile::base::metrics::qdr::listener_addr: "%{hiera('internal_api_leaf1')}"
   ```

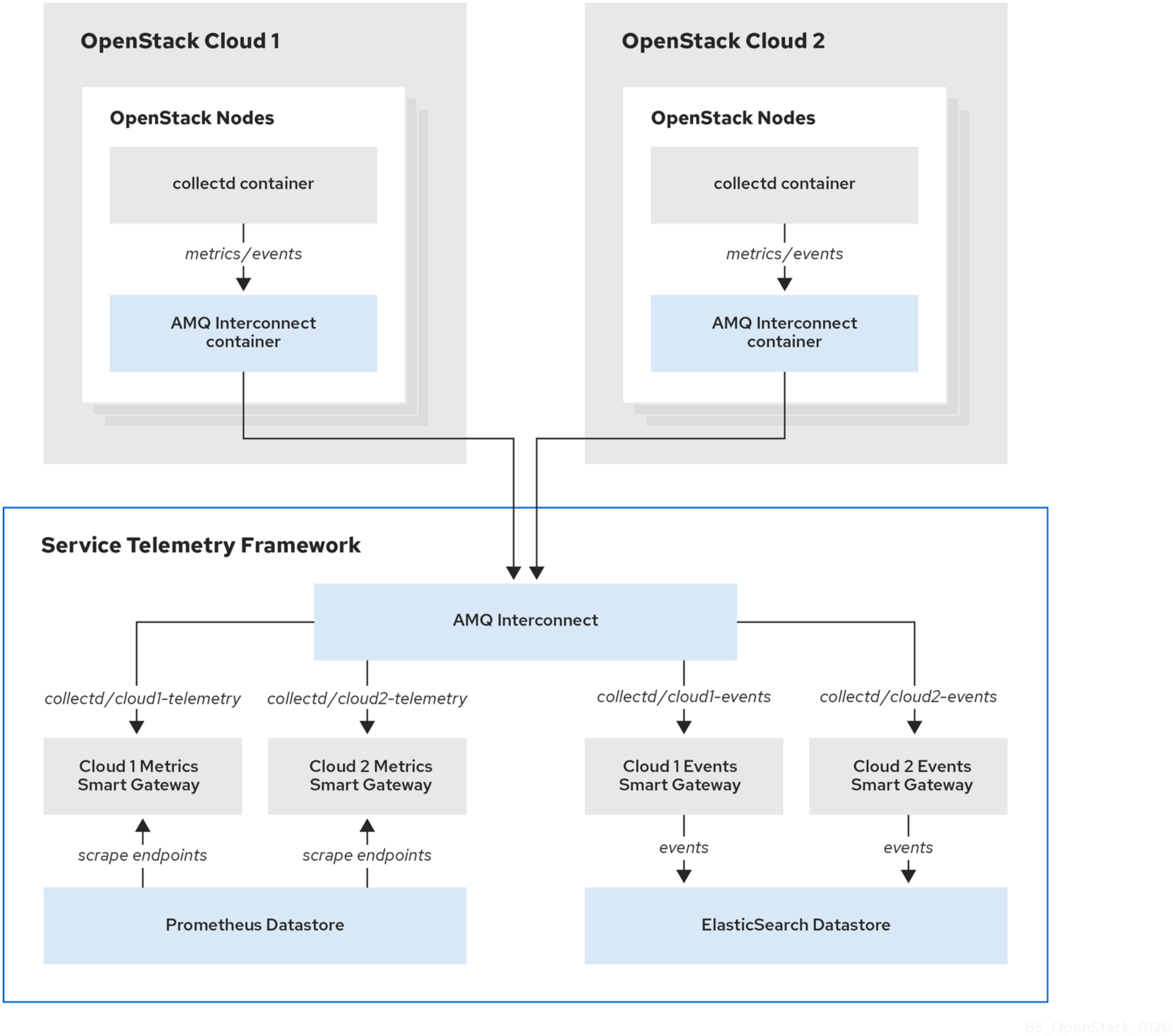
This example configuration is on a CephStorage leaf role:

```
CephStorageLeaf0ExtraConfig:
  tripleo::profile::base::metrics::collectd::amqp_host: "%{hiera('storage_subnet')}"
  tripleo::profile::base::metrics::qdr::listener_addr: "%{hiera('storage_subnet')}"
```

4.4. CONFIGURING MULTIPLE CLOUDS

You can configure multiple Red Hat OpenStack Platform (RHOSP) clouds to target a single instance of Service Telemetry Framework (STF). When you configure multiple clouds, every cloud must send metrics and events on their own unique message bus topic. In the STF deployment, Smart Gateway instances listen on these topics to save information to the common data store. Data that is stored by the Smart Gateway in the data storage domain is filtered by using the metadata that each of Smart Gateways creates.

Figure 4.1. Two RHOSP clouds connect to STF



To configure the RHOSP overcloud for a multiple cloud scenario, complete the following tasks:

1. Plan the AMQP address prefixes that you want to use for each cloud. For more information, see Section 4.4.1, “Planning AMQP address prefixes” .

2. Deploy metrics and events consumer Smart Gateways for each cloud to listen on the corresponding address prefixes. For more information, see Section 4.4.2, “Deploying Smart Gateways”.

3. Configure each cloud with a unique domain name. For more information, see Section 4.4.4, “Setting a unique cloud domain”.

4. Create the base configuration for STF. For more information, see Section 4.1.2, “Creating the base configuration for STF”.

5. Configure each cloud to send its metrics and events to STF on the correct address. For more information, see Section 4.4.5, “Creating the Red Hat OpenStack Platform environment file for multiple clouds”.

4.4.1. Planning AMQP address prefixes

By default, Red Hat OpenStack Platform (RHOSP) nodes receive data through two data collectors; collectd and Ceilometer. The collectd-sensubility plugin requires a unique address. These components send telemetry data or notifications to the respective AMQP addresses, for example, **collectd/telemetry**. STF Smart Gateways listen on those AMQP addresses for data. To support multiple clouds and to identify which cloud generated the monitoring data, configure each cloud to send data to a unique address. Add a cloud identifier prefix to the second part of the address. The following list shows some example addresses and identifiers:

- **collectd/cloud1-telemetry**
- **collectd/cloud1-notify**
- **sensubility/cloud1-telemetry**
- **anycast/ceilometer/cloud1-metering.sample**
- **anycast/ceilometer/cloud1-event.sample**
- **collectd/cloud2-telemetry**
- **collectd/cloud2-notify**
- **sensubility/cloud2-telemetry**
- **anycast/ceilometer/cloud2-metering.sample**
- **anycast/ceilometer/cloud2-event.sample**
- **collectd/us-east-1-telemetry**
- **collectd/us-west-3-telemetry**

4.4.2. Deploying Smart Gateways

You must deploy a Smart Gateway for each of the data collection types for each cloud; one for collectd metrics, one for collectd events, one for Ceilometer metrics, one for Ceilometer events, and one for collectd-sensubility metrics. Configure each of the Smart Gateways to listen on the AMQP address that you define for the corresponding cloud. To define Smart Gateways, configure the **clouds** parameter in the **ServiceTelemetry** manifest.

When you deploy STF for the first time, Smart Gateway manifests are created that define the initial Smart Gateways for a single cloud. When you deploy Smart Gateways for multiple cloud support, you deploy multiple Smart Gateways for each of the data collection types that handle the metrics and the events data for each cloud. The initial Smart Gateways are defined in **cloud1** with the following subscription addresses:

collector	type	default subscription address
collectd	metrics	collectd/telemetry
collectd	events	collectd/notify
collectd-sensubility	metrics	sensubility/telemetry
Ceilometer	metrics	anycast/ceilometer/metering.sample
Ceilometer	events	anycast/ceilometer/event.sample

Prerequisites

- You have determined your cloud naming scheme. For more information about determining your naming scheme, see Section 4.4.1, "Planning AMQP address prefixes" .
- You have created your list of clouds objects. For more information about creating the content for the **clouds** parameter, see the section called "The clouds parameter" .

Procedure

1. Log in to Red Hat OpenShift Container Platform.
2. Change to the **service-telemetry** namespace:

```
$ oc project service-telemetry
```

3. Edit the **default** ServiceTelemetry object and add a **clouds** parameter with your configuration:

> **WARNING**
>
> Long cloud names might exceed the maximum pod name of 63 characters. Ensure that the combination of the **ServiceTelemetry** name **default** and the **clouds.name** does not exceed 19 characters. Cloud names cannot contain any special characters, such as **-**. Limit cloud names to alphanumeric (a-z, 0-9).
>
> Topic addresses have no character limitation and can be different from the **clouds.name** value.

```
$ oc edit stf default
```

```
apiVersion: infra.watch/v1beta1
kind: ServiceTelemetry
metadata:
  ...
spec:
  ...
  clouds:
  - name: cloud1
    events:
      collectors:
      - collectorType: collectd
        subscriptionAddress: collectd/cloud1-notify
      - collectorType: ceilometer
        subscriptionAddress: anycast/ceilometer/cloud1-event.sample
    metrics:
      collectors:
      - collectorType: collectd
        subscriptionAddress: collectd/cloud1-telemetry
      - collectorType: sensubility
        subscriptionAddress: sensubility/cloud1-telemetry
      - collectorType: ceilometer
        subscriptionAddress: anycast/ceilometer/cloud1-metering.sample
  - name: cloud2
    events:
      ...
```

4. Save the ServiceTelemetry object.

5. Verify that each Smart Gateway is running. This can take several minutes depending on the number of Smart Gateways:

```
$ oc get po -l app=smart-gateway
NAME                                                      READY   STATUS    RESTARTS   AGE
default-cloud1-ceil-event-smartgateway-6cfb65478c-g5q82   2/2     Running   0          13h
default-cloud1-ceil-meter-smartgateway-58f885c76d-xmxwn   2/2     Running   0          13h
default-cloud1-coll-event-smartgateway-58fbbd4485-rl9bd   2/2     Running   0          13h
default-cloud1-coll-meter-smartgateway-7c6fc495c4-jn728   2/2     Running   0          13h
default-cloud1-sens-meter-smartgateway-8h4tc445a2-mm683   2/2     Running   0          13h
```

4.4.3. Deleting the default Smart Gateways

After you configure Service Telemetry Framework (STF) for multiple clouds, you can delete the default Smart Gateways if they are no longer in use. The Service Telemetry Operator can remove **SmartGateway** objects that were created but are no longer listed in the ServiceTelemetry **clouds** list of objects. To enable the removal of SmartGateway objects that are not defined by the **clouds** parameter, you must set the **cloudsRemoveOnMissing** parameter to **true** in the **ServiceTelemetry** manifest.

TIP

If you do not want to deploy any Smart Gateways, define an empty clouds list by using the **clouds: []** parameter.

WARNING

The **cloudsRemoveOnMissing** parameter is disabled by default. If you enable the **cloudsRemoveOnMissing** parameter, you remove any manually created **SmartGateway** objects in the current namespace without any possibility to restore.

Procedure

1. Define your **clouds** parameter with the list of cloud objects that you want the Service Telemetry Operator to manage. For more information, see the section called "The clouds parameter" .

2. Edit the ServiceTelemetry object and add the **cloudsRemoveOnMissing** parameter:

```
apiVersion: infra.watch/v1beta1
kind: ServiceTelemetry
metadata:
  ...
spec:
  ...
  cloudsRemoveOnMissing: true
  clouds:
    ...
```

3. Save the modifications.

4. Verify that the Operator deleted the Smart Gateways. This can take several minutes while the Operators reconcile the changes:

```
$ oc get smartgateways
```

4.4.4. Setting a unique cloud domain

To ensure that AMQ Interconnect router connections from Red Hat OpenStack Platform (RHOSP) to Service Telemetry Framework (STF) are unique and do not conflict, configure the **CloudDomain** parameter.

Procedure

1. Create a new environment file, for example, **hostnames.yaml**.

2. Set the **CloudDomain** parameter in the environment file, as shown in the following example:

hostnames.yaml

```
parameter_defaults:
    CloudDomain: newyork-west-04
    CephStorageHostnameFormat: 'ceph-%index%'
    ObjectStorageHostnameFormat: 'swift-%index%'
    ComputeHostnameFormat: 'compute-%index%'
```

3. Add the new environment file to your deployment.

Additional resources

- Section 4.4.5, “Creating the Red Hat OpenStack Platform environment file for multiple clouds”
- Core Overcloud Parameters in the *Overcloud Parameters* guide

4.4.5. Creating the Red Hat OpenStack Platform environment file for multiple clouds

To label traffic according to the cloud of origin, you must create a configuration with cloud-specific instance names. Create an **stf-connectors.yaml** file and adjust the values of **CeilometerQdrEventsConfig**, **CeilometerQdrMetricsConfig** and **CollectdAmqpInstances** to match the AMQP address prefix scheme.

NOTE

If you enabled container health and API status monitoring, you must also modify the **CollectdSensubilityResultsChannel** parameter. For more information, see Section 5.9, “Red Hat OpenStack Platform API status and containerized services health”.

Prerequisites

- You have created your list of clouds objects. For more information about creating the content for the clouds parameter, see the clouds configuration parameter.
- You have retrieved the AMQ Interconnect route address. For more information, see Section 4.1.1, “Retrieving the AMQ Interconnect route address” .
- You have created the base configuration for STF. For more information, see Section 4.1.2, “Creating the base configuration for STF”.
- You have created a unique domain name environment file. For more information, see Section 4.4.4, “Setting a unique cloud domain” .

Procedure

1. Log in to the Red Hat OpenStack Platform undercloud as the **stack** user.
2. Create a configuration file called **stf-connectors.yaml** in the **/home/stack** directory.
3. In the **stf-connectors.yaml** file, configure the **MetricsQdrConnectors** address to connect to the AMQ Interconnect on the overcloud deployment. Configure the **CeilometerQdrEventsConfig**, **CeilometerQdrMetricsConfig**, **CollectdAmqpInstances**, and **CollectdSensubilityResultsChannel** topic values to match the AMQP address that you want for this cloud deployment.

   stf-connectors.yaml

```
resource_registry:
  OS::TripleO::Services::Collectd: /usr/share/openstack-tripleo-heat-templates/deployment/metrics/collectd-container-puppet.yaml ❶

parameter_defaults:
  MetricsQdrConnectors:
    - host: stf-default-interconnect-5671-service-telemetry.apps.infra.watch ②
      port: 443
      role: edge
      verifyHostname: false
      sslProfile: sslProfile

  MetricsQdrSSLProfiles:
    - name: sslProfile

  CeilometerQdrEventsConfig:
    driver: amqp
    topic: cloud1-event ③

  CeilometerQdrMetricsConfig:
    driver: amqp
    topic: cloud1-metering ④

  CollectdAmqpInstances:
    cloud1-notify: ⑤
      notify: true
      format: JSON
      presettle: false
    cloud1-telemetry: ⑥
      format: JSON
      presettle: false

  CollectdSensubilityResultsChannel: sensubility/cloud1-telemetry ⑦
```

① Directly load the collectd service because you are not including the **collectd-write-qdr.yaml** environment file for multiple cloud deployments.

② Replace the **host** parameter with the value of **HOST/PORT** that you retrieved in Section 4.1.1, “Retrieving the AMQ Interconnect route address” .

③ Define the topic for Ceilometer events. This value is the address format of **anycast/ceilometer/cloud1-event.sample**.

④ Define the topic for Ceilometer metrics. This value is the address format of **anycast/ceilometer/cloud1-metering.sample**.

⑤ Define the topic for collectd events. This value is the format of **collectd/cloud1-notify**.

⑥ Define the topic for collectd metrics. This value is the format of **collectd/cloud1-telemetry**.

⑦ Define the topic for collectd-sensubility events. Ensure that this value is the exact string format **sensubility/cloud1-telemetry**

4. Ensure that the naming convention in the **stf-connectors.yaml** file aligns with the **spec.bridge.amqpUrl** field in the Smart Gateway configuration. For example, configure the **CeilometerQdrEventsConfig.topic** field to a value of **cloud1-event**.

5. Source the authentication file:

```
[stack@undercloud-0 ~]$ source stackrc

(undercloud) [stack@undercloud-0 ~]$
```

6. Include the **stf-connectors.yaml** file and unique domain name environment file **hostnames.yaml** in the **openstack overcloud deployment** command, with any other environment files relevant to your environment:

> **WARNING**
>
> If you use the **collectd-write-qdr.yaml** file with a custom **CollectdAmqpInstances** parameter, data publishes to the custom and default topics. In a multiple cloud environment, the configuration of the **resource_registry** parameter in the **stf-connectors.yaml** file loads the collectd service.

```
(undercloud) [stack@undercloud-0 ~]$ openstack overcloud deploy <other_arguments>
--templates /usr/share/openstack-tripleo-heat-templates \
  --environment-file <...other_environment_files...> \
  --environment-file /usr/share/openstack-tripleo-heat-
templates/environments/metrics/ceilometer-write-qdr.yaml \
  --environment-file /usr/share/openstack-tripleo-heat-templates/environments/metrics/qdr-
edge-only.yaml \
  --environment-file /home/stack/hostnames.yaml \
  --environment-file /home/stack/enable-stf.yaml \
  --environment-file /home/stack/stf-connectors.yaml
```

7. Deploy the Red Hat OpenStack Platform overcloud.

Additional resources

- For information about how to validate the deployment, see Section 4.1.5, “Validating client-side installation”.

4.4.6. Querying metrics data from multiple clouds

Data stored in Prometheus has a **service** label according to the Smart Gateway it was scraped from. You can use this label to query data from a specific cloud.

To query data from a specific cloud, use a Prometheus *promql* query that matches the associated **service** label; for example: **collectd_uptime{service="default-cloud1-coll-meter"}**.

CHAPTER 5. USING OPERATIONAL FEATURES OF SERVICE TELEMETRY FRAMEWORK

You can use the following operational features to provide additional functionality to the Service Telemetry Framework (STF):

- Configuring dashboards
- Configuring the metrics retention time period
- Configuring alerts
- Configuring SNMP traps
- Configuring high availability
- Configuring ephemeral storage
- Configuring an alternate observability strategy
- Monitoring the resource use of OpenStack services
- Monitoring container health and API status

5.1. DASHBOARDS IN SERVICE TELEMETRY FRAMEWORK

Use the third-party application, Grafana, to visualize system-level metrics that collectd and Ceilometer gathers for each individual host node.

For more information about configuring collectd, see Section 4.1, “Deploying Red Hat OpenStack Platform overcloud for Service Telemetry Framework”.

You can use two dashboards to monitor a cloud:

Infrastructure dashboard

Use the infrastructure dashboard to view metrics for a single node at a time. Select a node from the upper left corner of the dashboard.

Cloud view dashboard

Use the cloud view dashboard to view panels to monitor service resource usage, API stats, and cloud events. You must enable API health monitoring and service monitoring to provide the data for this dashboard. API health monitoring is enabled by default in the STF base configuration. For more information, see Section 4.1.2, “Creating the base configuration for STF” .

- For more information about API health monitoring, see Section 5.9, “Red Hat OpenStack Platform API status and containerized services health”.
- For more information about RHOSP service monitoring, see Section 5.8, “Resource usage of Red Hat OpenStack Platform services”.

5.1.1. Configuring Grafana to host the dashboard

Grafana is not included in the default Service Telemetry Framework (STF) deployment so you must deploy the Grafana Operator from OperatorHub.io. When you use the Service Telemetry Operator to deploy Grafana, it results in a Grafana instance and the configuration of the default data sources for the local STF deployment.

Procedure

1. Log in to Red Hat OpenShift Container Platform.
2. Change to the **service-telemetry** namespace:

   ```
   $ oc project service-telemetry
   ```

3. Deploy the Grafana operator:

   ```
   $ oc apply -f - <<EOF
   apiVersion: operators.coreos.com/v1alpha1
   kind: Subscription
   metadata:
     name: grafana-operator
     namespace: service-telemetry
   spec:
     channel: alpha
     installPlanApproval: Automatic
     name: grafana-operator
     source: operatorhubio-operators
     sourceNamespace: openshift-marketplace
   EOF
   ```

4. Verify that the Operator launched successfully. In the command output, if the value of the **PHASE** column is **Succeeded**, the Operator launched successfully:

   ```
   $ oc get csv --selector operators.coreos.com/grafana-operator.service-telemetry

   NAME                      DISPLAY            VERSION   REPLACES                  PHASE
   grafana-operator.v3.10.3  Grafana Operator   3.10.3    grafana-operator.v3.10.2  Succeeded
   ```

5. To launch a Grafana instance, create or modify the **ServiceTelemetry** object. Set **graphing.enabled** and **graphing.grafana.ingressEnabled** to **true**:

   ```
   $ oc edit stf default

   apiVersion: infra.watch/v1beta1
   kind: ServiceTelemetry
   ...
   spec:
     ...
     graphing:
       enabled: true
       grafana:
         ingressEnabled: true
   ```

6. Verify that the Grafana instance deployed:

   ```
   $ oc get pod -l app=grafana
   ```

```
NAME                              READY   STATUS    RESTARTS   AGE
grafana-deployment-7fc7848b56-sbkhv   1/1     Running   0          1m
```

7. Verify that the Grafana data sources installed correctly:

```
$ oc get grafanadatasources

NAME                  AGE
default-datasources     20h
```

8. Verify that the Grafana route exists:

```
$ oc get route grafana-route

NAME          HOST/PORT                                 PATH   SERVICES         PORT
TERMINATION   WILDCARD
grafana-route   grafana-route-service-telemetry.apps.infra.watch          grafana-service   3000
edge          None
```

5.1.2. Overriding the default Grafana container image

The dashboards in Service Telemetry Framework (STF) require features that are available only in Grafana version 8.1.0 and later. By default, the Service Telemetry Operator installs a compatible version. You can override the base Grafana image by specifying the image path to an image registry with **graphing.grafana.baseImage**.

Procedure

1. Ensure that you have the correct version of Grafana:

```
$ oc get pod -l "app=grafana" -ojsonpath='{.items[0].spec.containers[0].image}'
docker.io/grafana/grafana:7.3.10
```

2. If the running image is older than 8.1.0, patch the ServiceTelemetry object to update the image. Service Telemetry Operator updates the Grafana manifest, which restarts the Grafana deployment:

```
$ oc patch stf/default --type merge -p '{"spec":{"graphing":{"grafana":
{"baseImage":"docker.io/grafana/grafana:8.1.5"}}}}'
```

3. Verify that a new Grafana pod exists and has a **STATUS** value of **Running**:

```
$ oc get pod -l "app=grafana"
NAME                             READY     STATUS    RESTARTS   AGE
grafana-deployment-fb9799b58-j2hj2   1/1       Running   0          10s
```

4. Verify that the new instance is running the updated image:

```
$ oc get pod -l "app=grafana" -ojsonpath='{.items[0].spec.containers[0].image}'
docker.io/grafana/grafana:8.1.0
```

5.1.3. Importing dashboards

The Grafana Operator can import and manage dashboards by creating **GrafanaDashboard** objects. You can view example dashboards at https://github.com/infrawatch/dashboards.

Procedure

1. Import the infrastructure dashboard:

```
$ oc apply -f https://raw.githubusercontent.com/infrawatch/dashboards/master/deploy/stf-1.3/rhos-dashboard.yaml

grafanadashboard.integreatly.org/rhos-dashboard-1.3 created
```

2. Import the cloud dashboard:

> **WARNING**
>
> For some panels in the cloud dashboard, you must set the value of the collectd **virt** plugin parameter **hostname_format** to **name uuid hostname** in the **stf-connectors.yaml** file. If you do not configure this parameter, affected dashboards remain empty. For more information about the **virt** plugin, see collectd plugins.

```
$ oc apply -f https://raw.githubusercontent.com/infrawatch/dashboards/master/deploy/stf-1.3/rhos-cloud-dashboard.yaml

grafanadashboard.integreatly.org/rhos-cloud-dashboard-1.3 created
```

3. Import the cloud events dashboard:

```
$ oc apply -f https://raw.githubusercontent.com/infrawatch/dashboards/master/deploy/stf-1.3/rhos-cloudevents-dashboard.yaml

grafanadashboard.integreatly.org/rhos-cloudevents-dashboard created
```

4. Import the virtual machine dashboard:

```
$ oc apply -f https://raw.githubusercontent.com/infrawatch/dashboards/master/deploy/stf-1.3/virtual-machine-view.yaml

grafanadashboard.integreatly.org/virtual-machine-view-1.3 configured
```

5. Import the memcached dashboard:

```
$ oc apply -f https://raw.githubusercontent.com/infrawatch/dashboards/master/deploy/stf-1.3/memcached-dashboard.yaml

grafanadashboard.integreatly.org/memcached-dashboard-1.3 created
```

6. Verify that the dashboards are available:

```
$ oc get grafanadashboards

NAME                  AGE
memcached-dashboard-1.3      115s
rhos-cloud-dashboard-1.3     2m12s
rhos-cloudevents-dashboard   2m6s
rhos-dashboard-1.3           2m17s
virtual-machine-view-1.3     2m
```

7. Retrieve the Grafana route address:

```
$ oc get route grafana-route -ojsonpath='{.spec.host}'

grafana-route-service-telemetry.apps.infra.watch
```

8. In a web browser, navigate to https://*<grafana_route_address>*. Replace *<grafana_route_address>* with the value that you retrieved in the previous step.

9. To view the dashboard, click **Dashboards** and **Manage**.

5.1.4. Retrieving and setting Grafana login credentials

Service Telemetry Framework (STF) sets default login credentials when Grafana is enabled. You can override the credentials in the **ServiceTelemetry** object.

Procedure

1. Log in to Red Hat OpenShift Container Platform.

2. Change to the **service-telemetry** namespace:

```
$ oc project service-telemetry
```

3. Retrieve the default username and password from the STF object:

```
$ oc get stf default -o jsonpath="{.spec.graphing.grafana['adminUser','adminPassword']}"
```

4. To modify the default values of the Grafana administrator username and password through the ServiceTelemetry object, use the **graphing.grafana.adminUser** and **graphing.grafana.adminPassword** parameters.

5.2. METRICS RETENTION TIME PERIOD IN SERVICE TELEMETRY FRAMEWORK

The default retention time for metrics stored in Service Telemetry Framework (STF) is 24 hours, which provides enough data for trends to develop for the purposes of alerting.

For long-term storage, use systems designed for long-term data retention, for example, Thanos.

Additional resources

- To adjust STF for additional metrics retention time, see Section 5.2.1, “Editing the metrics retention time period in Service Telemetry Framework”.
- For recommendations about Prometheus data storage and estimating storage space, see https://prometheus.io/docs/prometheus/latest/storage/#operational-aspects
- For more information about Thanos, see https://thanos.io/

5.2.1. Editing the metrics retention time period in Service Telemetry Framework

You can adjust Service Telemetry Framework (STF) for additional metrics retention time.

Procedure

1. Log in to Red Hat OpenShift Container Platform.
2. Change to the service-telemetry namespace:

   ```
   $ oc project service-telemetry
   ```

3. Edit the ServiceTelemetry object:

   ```
   $ oc edit stf default
   ```

4. Add **retention: 7d** to the storage section of backends.metrics.prometheus.storage to increase the retention period to seven days:

   > NOTE
   >
   > If you set a long retention period, retrieving data from heavily populated Prometheus systems can result in queries returning results slowly.

   ```
   apiVersion: infra.watch/v1beta1
   kind: ServiceTelemetry
   metadata:
     name: stf-default
     namespace: service-telemetry
   spec:
     ...
     backends:
       metrics:
         prometheus:
           enabled: true
           storage:
             strategy: persistent
             retention: 7d
       ...
   ```

5. Save your changes and close the object.

Additional resources

- For more information about the metrics retention time, see Section 5.2, "Metrics retention time period in Service Telemetry Framework".

5.3. ALERTS IN SERVICE TELEMETRY FRAMEWORK

You create alert rules in Prometheus and alert routes in Alertmanager. Alert rules in Prometheus servers send alerts to an Alertmanager, which manages the alerts. Alertmanager can silence, inhibit, or aggregate alerts, and send notifications by using email, on-call notification systems, or chat platforms.

To create an alert, complete the following tasks:

1. Create an alert rule in Prometheus. For more information, see Section 5.3.1, "Creating an alert rule in Prometheus".

2. Create an alert route in Alertmanager. There are two ways in which you can create an alert route:

   - Creating a standard alert route in Alertmanager .
   - Creating an alert route with templating in Alertmanager .

Additional resources

For more information about alerts or notifications with Prometheus and Alertmanager, see https://prometheus.io/docs/alerting/overview/

To view an example set of alerts that you can use with Service Telemetry Framework (STF), see https://github.com/infrawatch/service-telemetry-operator/tree/master/deploy/alerts

5.3.1. Creating an alert rule in Prometheus

Prometheus evaluates alert rules to trigger notifications. If the rule condition returns an empty result set, the condition is false. Otherwise, the rule is true and it triggers an alert.

Procedure

1. Log in to Red Hat OpenShift Container Platform.

2. Change to the **service-telemetry** namespace:

```
$ oc project service-telemetry
```

3. Create a **PrometheusRule** object that contains the alert rule. The Prometheus Operator loads the rule into Prometheus:

```
$ oc apply -f - <<EOF
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  creationTimestamp: null
  labels:
    prometheus: default
    role: alert-rules
  name: prometheus-alarm-rules
  namespace: service-telemetry
spec:
  groups:
    - name: ./openstack.rules
      rules:
        - alert: Collectd metrics receive rate is zero
          expr: rate(sg_total_collectd_msg_received_count[1m]) == 0 1
EOF
```

1 To change the rule, edit the value of the **expr** parameter.

4. To verify that the Operator loaded the rules into Prometheus, run the **curl** command against the default-prometheus-proxy route with basic authentication:

```
$ curl -k --user "internal:$(oc get secret default-prometheus-htpasswd -ogo-template='{{
.data.password | base64decode }}')" https://$(oc get route default-prometheus-proxy -ogo-
template='{{ .spec.host }}')/api/v1/rules

{"status":"success","data":{"groups":
[{"name":"./openstack.rules","file":"/etc/prometheus/rules/prometheus-default-rulefiles-
0/service-telemetry-prometheus-alarm-rules.yaml","rules":
[{"state":"inactive","name":"Collectd metrics receive count is
zero","query":"rate(sg_total_collectd_msg_received_count[1m]) == 0","duration":0,"labels":
{},"annotations":{},"alerts":
[],"health":"ok","evaluationTime":0.00034627,"lastEvaluation":"2021-12-
07T17:23:22.160448028Z","type":"alerting"}],"interval":30,"evaluationTime":0.000353787,"last
Evaluation":"2021-12-07T17:23:22.160444017Z"}]}}
```

Additional resources

- For more information on alerting, see https://github.com/coreos/prometheus-operator/blob/master/Documentation/user-guides/alerting.md

5.3.2. Configuring custom alerts

You can add custom alerts to the **PrometheusRule** object that you created in Section 5.3.1, “Creating an alert rule in Prometheus”.

Procedure

1. Use the **oc edit** command:

```
$ oc edit prometheusrules prometheus-alarm-rules
```

2. Edit the **PrometheusRules** manifest.

3. Save and close the manifest.

Additional resources

- For more information about how to configure alerting rules, see https://prometheus.io/docs/prometheus/latest/configuration/alerting_rules/.

- For more information about PrometheusRules objects, see https://github.com/coreos/prometheus-operator/blob/master/Documentation/user-guides/alerting.md

5.3.3. Creating a standard alert route in Alertmanager

Use Alertmanager to deliver alerts to an external system, such as email, IRC, or other notification channel. The Prometheus Operator manages the Alertmanager configuration as a Red Hat OpenShift Container Platform secret. By default, Service Telemetry Framework (STF) deploys a basic configuration that results in no receivers:

```
alertmanager.yaml: |-
  global:
    resolve_timeout: 5m
  route:
    group_by: ['job']
    group_wait: 30s
    group_interval: 5m
    repeat_interval: 12h
    receiver: 'null'
  receivers:
  - name: 'null'
```

To deploy a custom Alertmanager route with STF, you must pass an **alertmanagerConfigManifest** parameter to the Service Telemetry Operator that results in an updated secret, managed by the Prometheus Operator.

> NOTE
>
> If your **alertmanagerConfigManifest** contains a custom template to construct the title and text of the sent alert, deploy the contents of the **alertmanagerConfigManifest** using a base64-encoded configuration. For more information, see Section 5.3.4, "Creating an alert route with templating in Alertmanager".

Procedure

1. Log in to Red Hat OpenShift Container Platform.

2. Change to the **service-telemetry** namespace:

   ```
   $ oc project service-telemetry
   ```

3. Edit the **ServiceTelemetry** object for your STF deployment:

   ```
   $ oc edit stf default
   ```

4. Add the new parameter **alertmanagerConfigManifest** and the **Secret** object contents to define the **alertmanager.yaml** configuration for Alertmanager:

   > NOTE
   >
   > This step loads the default template that the Service Telemetry Operator manages. To verify that the changes are populating correctly, change a value, return the **alertmanager-default** secret, and verify that the new value is loaded into memory. For example, change the value of the parameter **global.resolve_timeout** from **5m** to **10m**.

```
apiVersion: infra.watch/v1beta1
kind: ServiceTelemetry
metadata:
  name: default
  namespace: service-telemetry
spec:
  backends:
    metrics:
      prometheus:
        enabled: true
  alertmanagerConfigManifest: |
    apiVersion: v1
    kind: Secret
    metadata:
      name: 'alertmanager-default'
      namespace: 'service-telemetry'
    type: Opaque
    stringData:
      alertmanager.yaml: |-
        global:
          resolve_timeout: 10m
        route:
          group_by: ['job']
          group_wait: 30s
          group_interval: 5m
          repeat_interval: 12h
          receiver: 'null'
        receivers:
        - name: 'null'
```

5. Verify that the configuration has been applied to the secret:

```
$ oc get secret alertmanager-default -o go-template='{{index .data "alertmanager.yaml" |
base64decode }}'

global:
  resolve_timeout: 10m
route:
  group_by: ['job']
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 12h
  receiver: 'null'
receivers:
- name: 'null'
```

6. Run the **curl** command against the **alertmanager-proxy** service to retrieve the status and **configYAML** contents, and verify that the supplied configuration matches the configuration in Alertmanager:

```
$ oc run curl -it --serviceaccount=prometheus-k8s --restart='Never' --
image=radial/busyboxplus:curl -- sh -c "curl -k -H \"Content-Type: application/json\" -H
\"Authorization: Bearer \$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)\"
```

```
https://default-alertmanager-proxy:9095/api/v1/status"

{"status":"success","data":{"configYAML":"...",...}}
```

7. Verify that the **configYAML** field contains the changes you expect.

8. To clean up the environment, delete the **curl** pod:

```
$ oc delete pod curl

pod "curl" deleted
```

Additional resources

- For more information about the Red Hat OpenShift Container Platform secret and the Prometheus operator, see Prometheus user guide on alerting.

5.3.4. Creating an alert route with templating in Alertmanager

Use Alertmanager to deliver alerts to an external system, such as email, IRC, or other notification channel. The Prometheus Operator manages the Alertmanager configuration as a Red Hat OpenShift Container Platform secret. By default, Service Telemetry Framework (STF) deploys a basic configuration that results in no receivers:

```
alertmanager.yaml: |-
  global:
    resolve_timeout: 5m
  route:
    group_by: ['job']
    group_wait: 30s
    group_interval: 5m
    repeat_interval: 12h
    receiver: 'null'
  receivers:
  - name: 'null'
```

If the **alertmanagerConfigManifest** parameter contains a custom template, for example, to construct the title and text of the sent alert, deploy the contents of the **alertmanagerConfigManifest** by using a base64-encoded configuration.

Procedure

1. Log in to Red Hat OpenShift Container Platform.

2. Change to the **service-telemetry** namespace:

```
$ oc project service-telemetry
```

3. Edit the **ServiceTelemetry** object for your STF deployment:

```
$ oc edit stf default
```

4. To deploy a custom Alertmanager route with STF, you must pass an **alertmanagerConfigManifest** parameter to the Service Telemetry Operator that results in an updated secret that is managed by the Prometheus Operator:

```
apiVersion: infra.watch/v1beta1
kind: ServiceTelemetry
metadata:
  name: default
  namespace: service-telemetry
spec:
  backends:
    metrics:
      prometheus:
        enabled: true
  alertmanagerConfigManifest: |
    apiVersion: v1
    kind: Secret
    metadata:
      name: 'alertmanager-default'
      namespace: 'service-telemetry'
    type: Opaque
    data:
      alertmanager.yaml:
Z2xvYmFsOgogIHJlc29sdmVfdGltZW91dDogMTBtCiAgc2xhY2tfYXBpX3VybDogPHNsYWNrX
2FwaV91cmw+CnJlY2VpdmVyczoKICAtIG5hbWU6IHNsYWNrCiAgICBzbGFja19jb25maWdzO
gogICAgLSBjaGFubmVsOiAjc3RmLWFsZXJ0cwogICAgICB0aXRsZTogfC0KICAgICAgICAuLi
4KICAgICAgdGV4dDogPi0KICAgICAgICAuLi4Kcm91dGU6CiAgZ3JvdXBfYnk6IFsnam9iJ10KI
CBncm91cF93YWl0OiAzMHMKICBncm91cF9pbnRlcnZhbDogNW0KICByZXBlYXRfaW50ZXJ2
YWw6IDEyaAogIHJlY2VpdmVyOiAnc2xhY2snCg==
```

5. Verify that the configuration has been applied to the secret:

```
$ oc get secret alertmanager-default -o go-template='{{index .data "alertmanager.yaml" |
base64decode }}'

global:
  resolve_timeout: 10m
  slack_api_url: <slack_api_url>
receivers:
  - name: slack
    slack_configs:
    - channel: #stf-alerts
      title: |-
        ...
      text: >-
        ...
route:
  group_by: ['job']
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 12h
  receiver: 'slack'
```

6. Run the **curl** command against the **alertmanager-proxy** service to retrieve the status and **configYAML** contents, and verify that the supplied configuration matches the configuration in Alertmanager:

```
$ oc run curl -it --serviceaccount=prometheus-k8s --restart='Never' --
image=radial/busyboxplus:curl -- sh -c "curl -k -H \"Content-Type: application/json\" -H
\"Authorization: Bearer \$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)\"
https://default-alertmanager-proxy:9095/api/v1/status"

{"status":"success","data":{"configYAML":"...",...}}
```

7. Verify that the **configYAML** field contains the changes you expect.

8. To clean up the environment, delete the **curl** pod:

```
$ oc delete pod curl

pod "curl" deleted
```

Additional resources

- For more information about the Red Hat OpenShift Container Platform secret and the Prometheus operator, see Prometheus user guide on alerting.

5.4. CONFIGURING SNMP TRAPS

You can integrate Service Telemetry Framework (STF) with an existing infrastructure monitoring platform that receives notifications through SNMP traps. To enable SNMP traps, modify the **ServiceTelemetry** object and configure the **snmpTraps** parameters.

For more information about configuring alerts, see Section 5.3, "Alerts in Service Telemetry Framework" .

Prerequisites

- Know the IP address or hostname of the SNMP trap receiver where you want to send the alerts

Procedure

1. To enable SNMP traps, modify the **ServiceTelemetry** object:

```
$ oc edit stf default
```

2. Set the **alerting.alertmanager.receivers.snmpTraps** parameters:

```
apiVersion: infra.watch/v1beta1
kind: ServiceTelemetry
...
spec:
  ...
  alerting:
    alertmanager:
      receivers:
```

```
snmpTraps:
  enabled: true
  target: 10.10.10.10
```

3. Ensure that you set the value of **target** to the IP address or hostname of the SNMP trap receiver.

5.5. HIGH AVAILABILITY

With high availability, Service Telemetry Framework (STF) can rapidly recover from failures in its component services. Although Red Hat OpenShift Container Platform restarts a failed pod if nodes are available to schedule the workload, this recovery process might take more than one minute, during which time events and metrics are lost. A high availability configuration includes multiple copies of STF components, which reduces recovery time to approximately 2 seconds. To protect against failure of an Red Hat OpenShift Container Platform node, deploy STF to an Red Hat OpenShift Container Platform cluster with three or more nodes.

> **WARNING**
>
> STF is not yet a fully fault tolerant system. Delivery of metrics and events during the recovery period is not guaranteed.

Enabling high availability has the following effects:

- Three ElasticSearch pods run instead of the default one.
- The following components run two pods instead of the default one:
  - AMQ Interconnect
  - Alertmanager
  - Prometheus
  - Events Smart Gateway
  - Metrics Smart Gateway
- Recovery time from a lost pod in any of these services reduces to approximately 2 seconds.

5.5.1. Configuring high availability

To configure Service Telemetry Framework (STF) for high availability, add **highAvailability.enabled: true** to the ServiceTelemetry object in Red Hat OpenShift Container Platform. You can set this parameter at installation time or, if you already deployed STF, complete the following steps:

Procedure

1. Log in to Red Hat OpenShift Container Platform.

2. Change to the **service-telemetry** namespace:

```
$ oc project service-telemetry
```

3. Use the oc command to edit the ServiceTelemetry object:

```
$ oc edit stf default
```

4. Add **highAvailability.enabled: true** to the **spec** section:

```
apiVersion: infra.watch/v1beta1
kind: ServiceTelemetry
...
spec:
  ...
  highAvailability:
    enabled: true
```

5. Save your changes and close the object.

5.6. EPHEMERAL STORAGE

You can use ephemeral storage to run Service Telemetry Framework (STF) without persistently storing data in your Red Hat OpenShift Container Platform cluster.

> **WARNING**
>
> If you use ephemeral storage, you might experience data loss if a pod is restarted, updated, or rescheduled onto another node. Use ephemeral storage only for development or testing, and not production environments.

5.6.1. Configuring ephemeral storage

To configure STF components for ephemeral storage, add **...storage.strategy: ephemeral** to the corresponding parameter. For example, to enable ephemeral storage for the Prometheus back end, set **backends.metrics.prometheus.storage.strategy: ephemeral**. Components that support configuration of ephemeral storage include **alerting.alertmanager**, **backends.metrics.prometheus**, and **backends.events.elasticsearch**. You can add ephemeral storage configuration at installation time or, if you already deployed STF, complete the following steps:

Procedure

1. Log in to Red Hat OpenShift Container Platform.

2. Change to the **service-telemetry** namespace:

```
$ oc project service-telemetry
```

3. Edit the ServiceTelemetry object:

```
$ oc edit stf default
```

4. Add the **...storage.strategy: ephemeral** parameter to the **spec** section of the relevant component:

```
apiVersion: infra.watch/v1beta1
kind: ServiceTelemetry
metadata:
  name: stf-default
  namespace: service-telemetry
spec:
  alerting:
    enabled: true
    alertmanager:
      storage:
        strategy: ephemeral
  backends:
    metrics:
      prometheus:
        enabled: true
        storage:
          strategy: ephemeral
    events:
      elasticsearch:
        enabled: true
        storage:
          strategy: ephemeral
```

5. Save your changes and close the object.

5.7. OBSERVABILITY STRATEGY IN SERVICE TELEMETRY FRAMEWORK

Service Telemetry Framework (STF) does not include storage backends and alerting tools. STF uses community operators to deploy Prometheus, Alertmanager, Grafana, and Elasticsearch. STF makes requests to these community operators to create instances of each application configured to work with STF.

Instead of having Service Telemetry Operator create custom resource requests, you can use your own deployments of these applications or other compatible applications, and scrape the metrics Smart Gateways for delivery to your own Prometheus-compatible system for telemetry storage. If you set the observability strategy to use alternative backends instead, persistent or ephemeral storage is not required for STF.

5.7.1. Configuring an alternate observability strategy

To configure STF to skip the deployment of storage, visualization, and alerting backends, add **observabilityStrategy: none** to the ServiceTelemetry spec. In this mode, only AMQ Interconnect routers and metrics Smart Gateways are deployed, and you must configure an external Prometheus-compatible system to collect metrics from the STF Smart Gateways.

NOTE

Currently, only metrics are supported when you set **observabilityStrategy** to **none**. Events Smart Gateways are not deployed.

Procedure

1. Create a **ServiceTelemetry** object with the property **observabilityStrategy: none** in the **spec** parameter. The manifest shows results in a default deployment of STF that is suitable for receiving telemetry from a single cloud with all metrics collector types.

```
$ oc apply -f - <<EOF
apiVersion: infra.watch/v1beta1
kind: ServiceTelemetry
metadata:
  name: default
  namespace: service-telemetry
spec:
  observabilityStrategy: none
EOF
```

2. To verify that all workloads are operating correctly, view the pods and the status of each pod:

```
$ oc get pods
NAME                                                     READY   STATUS    RESTARTS   AGE
default-cloud1-ceil-meter-smartgateway-59c845d65b-gzhcs  3/3     Running   0          132m
default-cloud1-coll-meter-smartgateway-75bbd948b9-d5phm  3/3     Running   0          132m
default-cloud1-sens-meter-smartgateway-7fdbb57b6d-dh2g9  3/3     Running   0          132m
default-interconnect-668d5bbcd6-57b2l                    1/1     Running   0          132m
interconnect-operator-b8f5bb647-tlp5t                    1/1     Running   0          47h
service-telemetry-operator-566b9dd695-wkvjq              1/1     Running   0          156m
smart-gateway-operator-58d77dcf7-6xsq7                   1/1     Running   0          47h
```

Additional resources

For more information about configuring additional clouds or to change the set of supported collectors, see Section 4.4.2, "Deploying Smart Gateways"

5.8. RESOURCE USAGE OF RED HAT OPENSTACK PLATFORM SERVICES

You can monitor the resource usage of the Red Hat OpenStack Platform (RHOSP) services, such as the APIs and other infrastructure processes, to identify bottlenecks in the overcloud by showing services that run out of compute power. Resource usage monitoring is enabled by default.

Additional resources

- To disable resource usage monitoring, see Section 5.8.1, "Disabling resource usage monitoring of Red Hat OpenStack Platform services".

5.8.1. Disabling resource usage monitoring of Red Hat OpenStack Platform services

To disable the monitoring of RHOSP containerized service resource usage, you must set the **CollectdEnableLibpodstats** parameter to **false**.

Prerequisites

- You have created the **stf-connectors.yaml** file. For more information, see Section 4.1, “Deploying Red Hat OpenStack Platform overcloud for Service Telemetry Framework”.

- You are using the most current version of Red Hat OpenStack Platform (RHOSP) 16.2.

Procedure

1. Open the **stf-connectors.yaml** file and add the **CollectdEnableLibpodstats** parameter to override the setting in **enable-stf.yaml**. Ensure that **stf-connectors.yaml** is called from the **openstack overcloud deploy** command after **enable-stf.yaml**:

   ```
   CollectdEnableLibpodstats: false
   ```

2. Continue with the overcloud deployment procedure. For more information, see Section 4.1.4, “Deploying the overcloud”.

5.9. RED HAT OPENSTACK PLATFORM API STATUS AND CONTAINERIZED SERVICES HEALTH

You can use the OCI (Open Container Initiative) standard to assess the container health status of each Red Hat OpenStack Platform (RHOSP) service by periodically running a health check script. Most RHOSP services implement a health check that logs issues and returns a binary status. For the RHOSP APIs, the health checks query the root endpoint and determine the health based on the response time.

Monitoring of RHOSP container health and API status is enabled by default.

Additional resources

- To disable RHOSP container health and API status monitoring, see Section 5.9.1, “Disabling container health and API status monitoring”.

5.9.1. Disabling container health and API status monitoring

To disable RHOSP containerized service health and API status monitoring, you must set the **CollectdEnableSensubility** parameter to **false**.

Prerequisites

- You have created the **stf-connectors.yaml** file in your templates directory. For more information, see Section 4.1, “Deploying Red Hat OpenStack Platform overcloud for Service Telemetry Framework”.

- You are using the most current version of Red Hat OpenStack Platform (RHOSP) 16.2.

Procedure

1. Open the **stf-connectors.yaml** and add the **CollectdEnableSensubility** parameter to override the setting in **enable-stf.yaml**. Ensure that **stf-connectors.yaml** is called from the **openstack overcloud deploy** command after **enable-stf.yaml**:

   ```
   CollectdEnableSensubility: false
   ```

2. Continue with the overcloud deployment procedure. For more information, see Section 4.1.4, “Deploying the overcloud”.

Additional resources

- For more information about multiple cloud addresses, see Section 4.4, “Configuring multiple clouds”.

CHAPTER 6. UPGRADING SERVICE TELEMETRY FRAMEWORK TO VERSION 1.4

To migrate from Service Telemetry Framework (STF) v1.3 to STF v1.4, you must replace the **ClusterServiceVersion** and **Subscription** objects in the **service-telemetry** namespace on your Red Hat OpenShift Container Platform environment.

Prerequisites

- You have upgraded your Red Hat OpenShift Container Platform environment to v4.8. STF v1.4 does not run on Red Hat OpenShift Container Platform versions less than v4.8.
- You have backed up your data. Upgrading STF v1.3 to v1.4 results in a brief outage while the Smart Gateways and other components are updated. Additionally, changes to the **ServiceTelemetry** and **SmartGateway** objects do not have any effect while the Operators are being replaced.

To upgrade from STF v1.3 to v1.4, complete the following procedures:

Procedure

1. Remove the STF 1.3 Smart Gateway Operator .
2. Update the Service Telemetry Operator to 1.4 .

6.1. REMOVING STF 1.3 SMART GATEWAY OPERATOR

Remove the Smart Gateway Operator from STF 1.3.

Procedure

1. Log in to Red Hat OpenShift Container Platform.
2. Change to the **service-telemetry** namespace:

```
$ oc project service-telemetry
```

3. Retrieve the **Subscription** name of the Smart Gateway Operator. Replace **service-telemetry** in the selector with the namespace that hosts your STF instance if it is different from the default namespace. Verify that only one subscription is returned:

```
$ oc get sub --selector=operators.coreos.com/smart-gateway-operator.service-telemetry

NAME                                                              PACKAGE              SOURCE
CHANNEL
smart-gateway-operator-stable-1.3-redhat-operators-openshift-marketplace   smart-gateway-
operator   redhat-operators   stable-1.3
```

4. Delete the Smart Gateway Operator subscription:

```
$ oc delete sub --selector=operators.coreos.com/smart-gateway-operator.service-telemetry

subscription.operators.coreos.com "smart-gateway-operator-stable-1.3-redhat-operators-
```

```
openshift-marketplace" deleted
```

5. Retrieve the Smart Gateway Operator ClusterServiceVersion and verify that only one ClusterServiceVersion is returned:

```
$ oc get csv --selector=operators.coreos.com/smart-gateway-operator.service-telemetry

NAME                                    DISPLAY                   VERSION          REPLACES   PHASE
smart-gateway-operator.v3.0.1635451893   Smart Gateway Operator   3.0.1635451893
Succeeded
```

6. Delete the Smart Gateway Operator ClusterServiceVersion:

```
$ oc delete csv --selector=operators.coreos.com/smart-gateway-operator.service-telemetry

clusterserviceversion.operators.coreos.com "smart-gateway-operator.v3.0.1635451893"
deleted
```

7. Delete the SmartGateway Custom Resource Definition (CRD). After removal of the CRD, no data flows to STF until the upgrade is completed and the Smart Gateway instances are reinstantiated:

```
$ oc delete crd smartgateways.smartgateway.infra.watch

customresourcedefinition.apiextensions.k8s.io "smartgateways.smartgateway.infra.watch"
deleted
```

6.2. UPDATING THE SERVICE TELEMETRY OPERATOR TO 1.4

You must change the subscription channel of the Service Telemetry Operator which manages the STF instances to the **stable-1.4** channel.

Procedure

1. Log in to Red Hat OpenShift Container Platform.

2. Change to the **service-telemetry** namespace:

```
$ oc project service-telemetry
```

3. Patch the Service Telemetry Operator Subscription to use the stable-1.4 channel. Replace the **service-telemetry** in the selector with the namespace that hosts your STF instance if it is different from the default namespace:

```
$ oc patch $(oc get sub --selector=operators.coreos.com/service-telemetry-operator.service-
telemetry -oname) --patch $'spec:\n  channel: stable-1.4' --type=merge

subscription.operators.coreos.com/service-telemetry-operator patched
```

4. Monitor the output of the **oc get csv** command until the Smart Gateway Operator is installed and Service Telemetry Operator moves through the update phases. When the phase changes to **Succeeded**, the Service Telemetry Operator has completed the update:

```
$ watch -n5 oc get csv

NAME                                   DISPLAY                                 VERSION
REPLACES                                 PHASE
amq7-cert-manager.v1.0.3                   Red Hat Integration - AMQ Certificate Manager   1.0.3
amq7-cert-manager.v1.0.2                   Succeeded
amq7-interconnect-operator.v1.10.5           Red Hat Integration - AMQ Interconnect
1.10.5          amq7-interconnect-operator.v1.10.4          Succeeded
elasticsearch-eck-operator-certified.1.9.1   Elasticsearch (ECK) Operator                  1.9.1
Succeeded
prometheusoperator.0.47.0                  Prometheus Operator                           0.47.0
prometheusoperator.0.37.0                  Succeeded
service-telemetry-operator.v1.4.1641504218   Service Telemetry Operator
1.4.1641504218   service-telemetry-operator.v1.3.1635451892   Succeeded
smart-gateway-operator.v4.0.1641504216       Smart Gateway Operator
4.0.1641504216                               Succeeded
```

5. Validate that all pods are ready and running. Your environment might differ from the following example output:

```
$ oc get pods

NAME                                                      READY   STATUS    RESTARTS   AGE
alertmanager-default-0                                    3/3     Running   0          162m
default-cloud1-ceil-event-smartgateway-5599bcfc9d-wp48n   2/2     Running   1          160m
default-cloud1-ceil-meter-smartgateway-c8fdf579c-955kt    3/3     Running   0          160m
default-cloud1-coll-event-smartgateway-97b54b7dc-5zz2v    2/2     Running   0          159m
default-cloud1-coll-meter-smartgateway-774b9988b8-wb5vd   3/3     Running   0          160m
default-cloud1-sens-meter-smartgateway-b98966fbf-rnqwf    3/3     Running   0          159m
default-interconnect-675dd97bc4-dcrzk                     1/1     Running   0          171m
default-snmp-webhook-7854d4889d-wgmgg                     1/1     Running   0          171m
elastic-operator-c54ff8cc-jcg8d                           1/1     Running   6          3h55m
elasticsearch-es-default-0                                1/1     Running   0          160m
interconnect-operator-6bf74c4ffb-hkmbq                    1/1     Running   0          3h54m
prometheus-default-0                                      3/3     Running   1          160m
prometheus-operator-fc64987d-f7gx4                        1/1     Running   0          3h54m
service-telemetry-operator-68d888f767-s5kzh               1/1     Running   0          163m
smart-gateway-operator-584df7959-llxgl                    1/1     Running   0          163m
```

CHAPTER 7. REMOVING SERVICE TELEMETRY FRAMEWORK FROM THE RED HAT OPENSHIFT CONTAINER PLATFORM ENVIRONMENT

Remove Service Telemetry Framework (STF) from an Red Hat OpenShift Container Platform environment if you no longer require the STF functionality.

To remove STF from the Red Hat OpenShift Container Platform environment, you must perform the following tasks:

1. Delete the namespace.
2. Remove the catalog source.
3. Remove the AMQ Cert Manager Operator.

7.1. DELETING THE NAMESPACE

To remove the operational resources for STF from Red Hat OpenShift Container Platform, delete the namespace.

Procedure

1. Run the **oc delete** command:

   ```
   $ oc delete project service-telemetry
   ```

2. Verify that the resources have been deleted from the namespace:

   ```
   $ oc get all
   No resources found.
   ```

7.2. REMOVING THE CATALOGSOURCE

If you do not expect to install Service Telemetry Framework (STF) again, delete the CatalogSource. When you remove the CatalogSource, PackageManifests related to STF are automatically removed from the Operator Lifecycle Manager catalog.

Procedure

1. If you enabled the OperatorHub.io Community Catalog Source during the installation process and you no longer need this catalog source, delete it:

   ```
   $ oc delete --namespace=openshift-marketplace catalogsource operatorhubio-operators
   catalogsource.operators.coreos.com "operatorhubio-operators" deleted
   ```

Additional resources

- Section 3.1, “Deploying Service Telemetry Framework to the Red Hat OpenShift Container Platform environment”.

7.3. REMOVING THE AMQ CERTIFICATE MANAGER OPERATOR

If you are not using the AMQ Certificate Manager for any other applications, delete the Subscription, ClusterServiceVersion, and CustomResourceDefinitions.

Procedure

1. Retrieve the version number of your installed ClusterServiceVersion:

   ```
   $ oc get --namespace=openshift-operators subscription amq7-cert-manager-operator -oyaml
   | grep currentCSV
   ```

   Example output:

   ```
     currentCSV: amq7-cert-manager.v1.0.6
   ```

2. Delete the Subscription from the **openshift-operators** namespace:

   ```
   $ oc delete --namespace=openshift-operators subscription amq7-cert-manager-operator

   subscription.operators.coreos.com "amq7-cert-manager-operator" deleted
   ```

3. Get the current list of CustomResourceDefinitions provided by the Operator so they can be deleted after removal of the ClusterServiceVersion:

   ```
   $ oc get csv -nopenshift-operators amq7-cert-manager.v1.0.6  -oyaml | grep "kind:
   CustomResourceDefinition" -A2 | grep name | awk '{print $2}'
   ```

   Example output:

   ```
   certificates.certmanager.k8s.io
   challenges.certmanager.k8s.io
   clusterissuers.certmanager.k8s.io
   issuers.certmanager.k8s.io
   orders.certmanager.k8s.io
   ```

4. Delete the ClusterServiceVersion from the **openshift-operators** namespace:

   ```
   $ oc delete --namespace=openshift-operators csv amq7-cert-manager.v1.0.6
   ```

   Example output:

   ```
   clusterserviceversion.operators.coreos.com "amq7-cert-manager.v1.0.6" deleted
   ```

5. Delete the CustomResourceDefinitions related to the AMQ Cert Manager Operator:

   ```
   $ oc delete crd certificates.certmanager.k8s.io challenges.certmanager.k8s.io
   clusterissuers.certmanager.k8s.io issuers.certmanager.k8s.io orders.certmanager.k8s.io
   ```

   Example output:

   ```
   customresourcedefinition.apiextensions.k8s.io "certificates.certmanager.k8s.io" deleted
   ```

```
customresourcedefinition.apiextensions.k8s.io "challenges.certmanager.k8s.io" deleted
customresourcedefinition.apiextensions.k8s.io "clusterissuers.certmanager.k8s.io" deleted
customresourcedefinition.apiextensions.k8s.io "issuers.certmanager.k8s.io" deleted
customresourcedefinition.apiextensions.k8s.io "orders.certmanager.k8s.io" deleted
```

Additional information

For more information about removing Operators from Red Hat OpenShift Container Platform, see https://docs.openshift.com/container-platform/4.10/operators/admin/olm-deleting-operators-from-cluster.html