Red Hat OpenStack Platform 16.1

Service Telemetry Framework 1.3

Installing and deploying Service Telemetry Framework 1.3

Last Updated: 2021-07-19

Red Hat OpenStack Platform 16.1 Service Telemetry Framework 1.3

Installing and deploying Service Telemetry Framework 1.3

OpenStack Team
rhos-docs@redhat.com

Legal Notice

Abstract

This guide contains information about installing the core components and deploying Service Telemetry Framework 1.3.

Table of Contents

CHAPTER 1. INTRODUCTION TO SERVICE TELEMETRY FRAMEWORK 1.3

IMPORTANT

Service Telemetry Framework (STF) is compatible with Red Hat OpenShift Container Platform versions 4.6 and 4.7.

Service Telemetry Framework (STF) receives monitoring data from Red Hat OpenStack Platform or third-party nodes for storage, viewing on dashboards, and alerting. The monitoring data can be either of two types:

Metric

a numeric measurement of an application or system

Event

irregular and discrete occurrences that happen in a system

The collection components that are required on the clients are lightweight. The multicast message bus that is shared by all clients and the deployment provides fast and reliable data transport. Other modular components for receiving and storing data are deployed in containers on OCP.

STF provides access to monitoring functions such as alert generation, visualization through dashboards, and single source of truth telemetry analysis to support orchestration.

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.

1.2. SERVICE TELEMETRY FRAMEWORK ARCHITECTURE

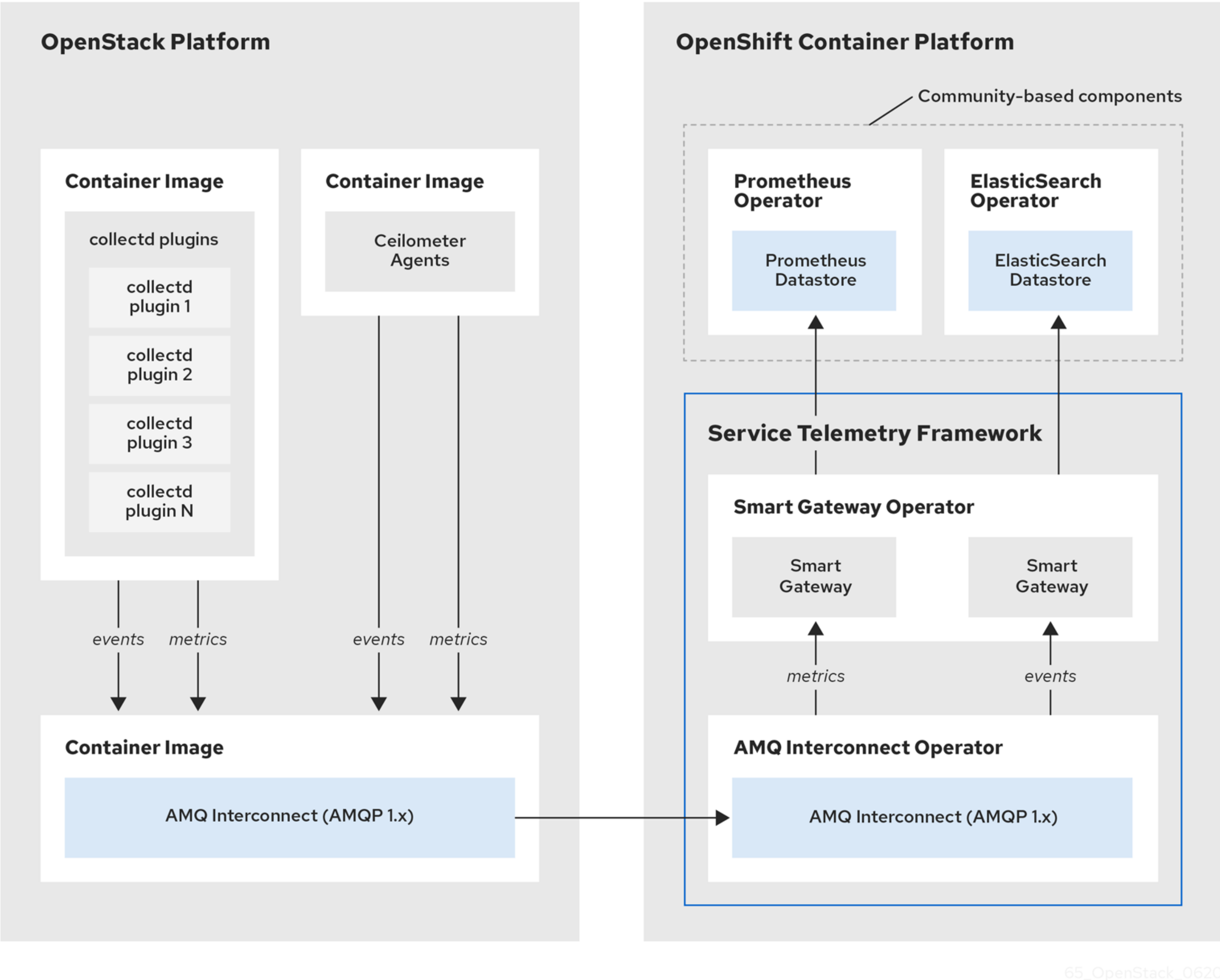
Service Telemetry Framework (STF) uses the components described in the following table:

Table 1.1. STF components

Client	Component	Server (OCP)
yes	An AMQP 1.x compatible messaging bus to shuttle the metrics to STF for storage in Prometheus	yes
no	Smart Gateway to pick metrics and events from the AMQP 1.x bus and to deliver events to ElasticSearch or to provide metrics to Prometheus	yes
no	Prometheus as time-series data storage	yes
no	ElasticSearch as events data storage	yes
yes	collectd to collect infrastructure metrics and events	no

Client	Component	Server (OCP)
yes	Ceilometer to collect Red Hat OpenStack Platform metrics and events	no

Figure 1.1. Service Telemetry Framework architecture overview



NOTE

The Service Telemetry Framework data collection components, collectd and Ceilometer, and the transport components, AMQ Interconnect and Smart Gateway, are fully supported. The data storage components, Prometheus and ElasticSearch, including the Operator artifacts, and visualization component Grafana are community-supported, and are not officially supported.

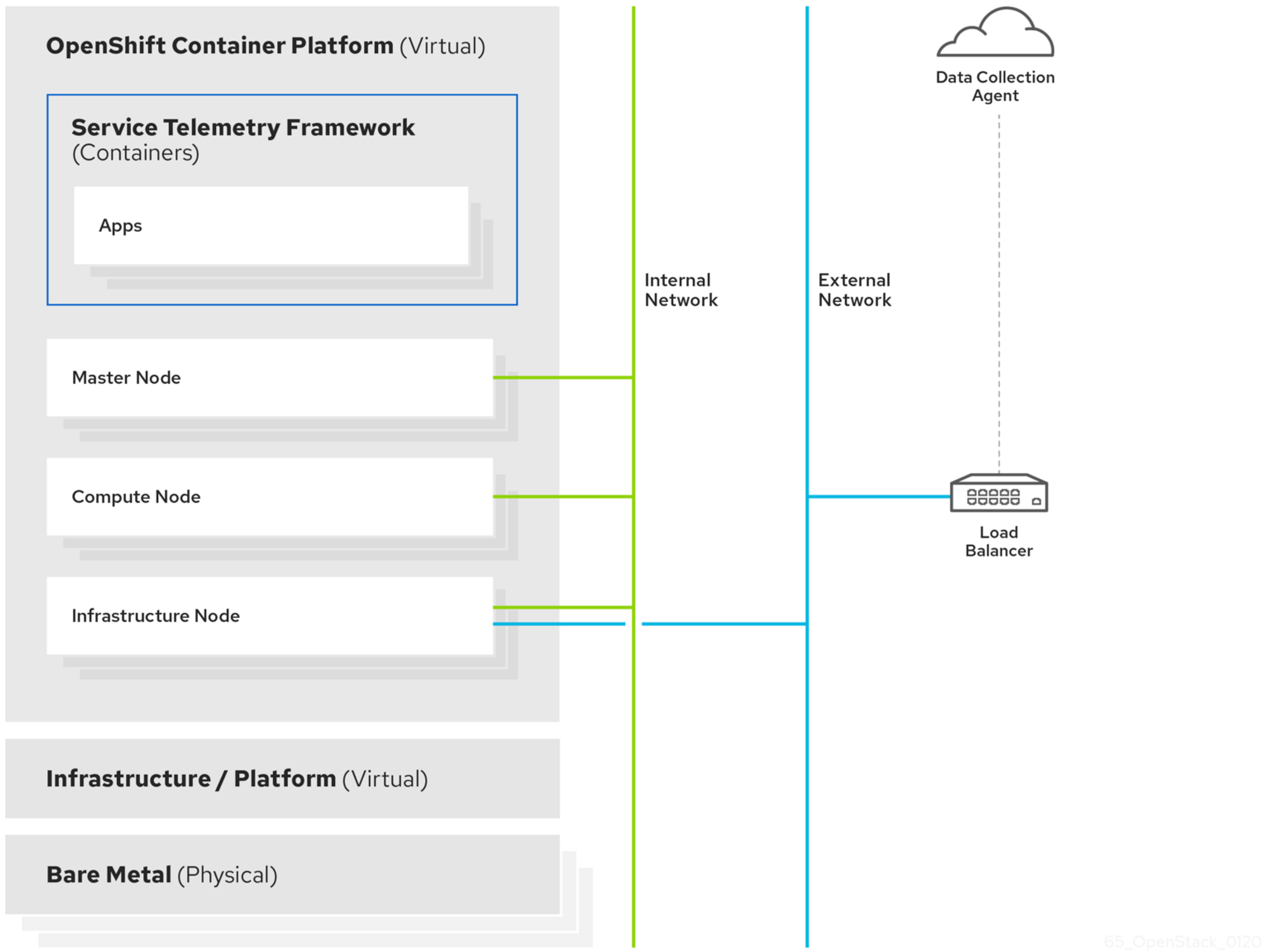
For metrics, on the client side, collectd provides infrastructure metrics (without project data), and Ceilometer provides Red Hat OpenStack Platform platform data based on projects or user workload. Both Ceilometer and collectd deliver data to Prometheus by using the AMQ Interconnect transport, delivering 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 or Ceilometer delivers event data to the server side by using the AMQ Interconnect transport, delivering the data through the message bus. Another Smart Gateway writes the data to the ElasticSearch datastore.

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

- Service Telemetry Framework 1.3 (STF)
- Red Hat OpenShift Container Platform 4.6 (OCP) or 4.7
- Infrastructure platform

Figure 1.2. Server-side STF monitoring infrastructure



> **NOTE**
>
> Do not install OCP on the same infrastructure that you want to monitor.

Additional resources

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

1.3. INSTALLATION SIZE OF RED HAT OPENSHIFT CONTAINER PLATFORM

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

- The number of nodes you want to monitor.
- The number of metrics 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 the existing Red Hat OpenShift Container Platform environment. Ensure that you install monitoring for Red Hat OpenStack Platform on a platform separate from your Red Hat OpenStack Platform environment. You can install Red Hat OpenShift Container Platform (OCP) on baremetal or other supported cloud platforms. For more information about installing OCP, see OpenShift Container Platform 4.6 Documentation.

The size of your OCP environment depends on the infrastructure you select. For more information about minimum resources requirements when installing OCP 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 which 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

As you prepare your OCP environment for STF, you must plan for persistent storage, adequate resources, and event storage:

- Ensure that persistent storage is available in your Red Hat OpenShift Container Platform cluster to permit a production grade deployment. For more information, see Section 2.1, "Persistent volumes".
- Ensure that enough resources are available to run the Operators and the application containers. For more information, see Section 2.2, "Resource allocation".
- To install ElasticSearch, you must use a community catalog source. If you do not want to use a community catalog or if you do not want to store events, see Section 3.1, "Deploying STF to the OCP environment".
- STF uses ElasticSearch to store events, which requires a larger than normal **vm.max_map_count**. The **vm.max_map_count** value is set by default in Red Hat OpenShift Container Platform. For more information about how to edit the value of **vm.max_map_count**, see Section 2.4, "Node tuning operator".

2.1. PERSISTENT VOLUMES

STF uses persistent storage in OCP to instantiate the volumes dynamically so that Prometheus and ElasticSearch can store metrics and events.

When persistent storage is enabled through the Service Telemetry Operator, the Persistent Volume Claims 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 OCP default configured **storageClass**.

Additional resources

- For more information about configuring persistent storage for OCP, see Understanding persistent storage.
- For more information about recommended configurable storage technology in Red Hat OpenShift Container Platform, see Recommended configurable storage technology .

2.1.1. Ephemeral storage

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

2.2. RESOURCE ALLOCATION

To enable the scheduling of pods within the OCP 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 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.

2.3. METRICS RETENTION TIME PERIOD

The default retention time for metrics stored in STF is 24 hours, which provides enough data to allow for trends to develop for the purposes of alerting. To adjust STF for additional metrics retention time, set a new value in **backends.metrics.prometheus.storage.retention**, for example, **7d** for seven days. If you use long retention periods, returning data from heavily populated Prometheus systems can result in queries returning slowly.

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

Additional resources

- For recommendations about Prometheus data storage and estimating storage space, see https://prometheus.io/docs/prometheus/latest/storage/#operational-aspects

2.4. NODE TUNING OPERATOR

STF uses ElasticSearch to store events, which requires a larger than normal **vm.max_map_count**. The **vm.max_map_count** value is set by default in Red Hat OpenShift Container Platform.

TIP

If your host platform is a typical Red Hat OpenShift Container Platform 4 environment, do not make any adjustments. The default node tuning operator is configured to account for ElasticSearch workloads.

If you want to edit the value of **vm.max_map_count**, you cannot apply node tuning manually using the **sysctl** command because Red Hat OpenShift Container Platform manages nodes directly. To configure values and apply them to the infrastructure, you must use the node tuning operator. For more information, see Using the Node Tuning Operator.

In an OCP deployment, the default node tuning operator specification provides the required profiles for ElasticSearch workloads or pods scheduled on nodes. To view the default cluster node tuning specification, run the following command:

```
$ oc get Tuned/default -o yaml -n openshift-cluster-node-tuning-operator
```

The output of the default specification is documented at Default profiles set on a cluster. You can manage the assignment of profiles in the **recommend** section where profiles are applied to a node when certain conditions are met. When scheduling ElasticSearch to a node in STF, one of the following profiles is applied:

- **openshift-control-plane-es**
- **openshift-node-es**

When scheduling an ElasticSearch pod, there must be a label present that matches **tuned.openshift.io/elasticsearch**. If the label is present, one of the two profiles is assigned to the pod. No action is required by the administrator if you use the recommended Operator for ElasticSearch. If you use a custom-deployed ElasticSearch with STF, ensure that you add the **tuned.openshift.io/elasticsearch** label to all scheduled pods.

Additional resources

- For more information about virtual memory usage by ElasticSearch, see https://www.elastic.co/guide/en/elasticsearch/reference/current/vm-max-map-count.html
- For more information about how the profiles are applied to nodes, see Custom tuning specification.

CHAPTER 3. INSTALLING THE CORE COMPONENTS OF SERVICE TELEMETRY FRAMEWORK

You can use Operators to load the various application components and objects. Each of the following STF core components are managed by Operators:

- Prometheus and AlertManager
- ElasticSearch
- Smart Gateway
- AMQ Interconnect

Prerequisites

- Red Hat OpenShift Container Platform (OCP) version 4.6 or 4.7 is running.
- You have prepared your Red Hat OpenShift Container Platform (OCP) environment and ensured that there is persistent storage and enough resources to run the STF components on top of the OCP environment.

IMPORTANT

Service Telemetry Framework (STF) is compatible with Red Hat OpenShift Container Platform version 4.6 and 4.7.

Additional resources

- For more information about Operators, see the *Understanding Operators* guide.

3.1. DEPLOYING STF TO THE OCP ENVIRONMENT

You can deploy STF to the OCP environment in one of two ways:

- Deploy STF and store events with ElasticSearch. For more information, see Section 3.1.1, "Deploying STF to the OpenShift environment with ElasticSearch".
- Deploy STF without ElasticSearch and disable events support. For more information, see Section 3.1.2, "Deploying STF to the Openshift environment without ElasticSearch" .

3.1.1. Deploying STF to the OpenShift environment with ElasticSearch

To deploy STF and store events with ElasticSearch, complete the following tasks:

Procedure

1. Section 3.1.3, "Creating a namespace" .
2. Section 3.1.4, "Creating an OperatorGroup" .
3. Section 3.1.5, "Enabling the OperatorHub.io Community Catalog Source" .
4. Section 3.1.6, "Subscribing to the AMQ Certificate Manager Operator" .

5. Section 3.1.7, “Subscribing to the Elastic Cloud on Kubernetes Operator” .
6. Section 3.1.8, “Subscribing to the Service Telemetry Operator” .
7. Section 3.3, “Creating a ServiceTelemetry object in OCP” .

3.1.2. Deploying STF to the Openshift environment without ElasticSearch

To deploy STF without ElasticSearch and disable events support, complete the following tasks:

Procedure

1. Section 3.1.3, “Creating a namespace”.
2. Section 3.1.4, “Creating an OperatorGroup”.
3. Section 3.1.6, “Subscribing to the AMQ Certificate Manager Operator” .
4. Section 3.1.8, “Subscribing to the Service Telemetry Operator” .
5. Section 3.3, “Creating a ServiceTelemetry object in OCP” .

3.1.3. Creating a namespace

Create a namespace to hold the STF components. The **service-telemetry** namespace is used throughout the documentation:

Procedure

- Enter the following command:

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

3.1.4. Creating an OperatorGroup

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

Procedure

- Create the STF OperatorGroup:

```
$ 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
```

Additional resources

For more information, see OperatorGroups.

3.1.5. Enabling the OperatorHub.io Community Catalog Source

Before you install data storage and visualization operators, you must have access to the resources on the OperatorHub.io Community Catalog Source.

> **NOTE**
>
> All operators installed from this CatalogSource are not supported by Red Hat.

Procedure

- Enable the OperatorHub.io CatalogSource:

```
$ 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
```

3.1.6. Subscribing to the AMQ Certificate Manager Operator

You must subscribe to the AMQ Certificate Manager Operator before you deploy the other STF components because the AMQ Certificate Manager Operator runs globally-scoped. 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.

> **NOTE**
>
> The AMQ Certificate Manager is installed globally for all namespaces, so the **namespace** value provided is **openshift-operators**. You might not see your **amq7-cert-manager.v1.0.0** ClusterServiceVersion in the **service-telemetry** namespace for a few minutes until the processing executes against the namespace.

Procedure

1. Enable the Red Hat STF Operators CatalogSource:

```
$ oc create -f - <<EOF
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: redhat-operators-stf
  namespace: openshift-marketplace
spec:
  displayName: Red Hat STF Operators
```

```
  image: quay.io/redhat-operators-stf/stf-catalog:stable
  publisher: Red Hat
  sourceType: grpc
  updateStrategy:
    registryPoll:
      interval: 30m
EOF
```

2. Subscribe to the AMQ Certificate Manager Operator via the **redhat-operators-stf** CatalogSource:

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

3. Validate your **ClusterServiceVersion**:

```
$ oc get --namespace openshift-operators csv

NAME                       DISPLAY                                         VERSION   REPLACES   PHASE
amq7-cert-manager.v1.0.0   Red Hat Integration - AMQ Certificate Manager   1.0.0
Succeeded
```

Ensure that amq7-cert-manager.v1.0.0 has a phase **Succeeded**.

3.1.7. Subscribing to the Elastic Cloud on Kubernetes Operator

Before you install the Service Telemetry Operator and if you plan to store events in ElasticSearch, you must enable the Elastic Cloud Kubernetes Operator.

Procedure

1. To enable the Elastic Cloud on Kubernetes Operator, create the following manifest in your OCP environment:

```
$ oc create -f - <<EOF
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: elastic-cloud-eck
  namespace: service-telemetry
spec:
  channel: stable
```

```
  installPlanApproval: Automatic
  name: elastic-cloud-eck
  source: operatorhubio-operators
  sourceNamespace: openshift-marketplace
EOF
```

2. Verify that the **ClusterServiceVersion** for ElasticSearch Cloud on Kubernetes **Succeeded**:

```
$ oc get csv

NAME                   DISPLAY                          VERSION   REPLACES                  PHASE
...
elastic-cloud-eck.v1.6.0   Elasticsearch (ECK) Operator   1.6.0     elastic-cloud-eck.v1.5.0
Succeeded
```

3.1.8. Subscribing to the Service Telemetry Operator

You must subscribe to the Service Telemetry Operator, which manages the STF instances.

Procedure

1. To create the Service Telemetry Operator subscription, enter the **oc create -f** command:

```
$ oc create -f - <<EOF
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: service-telemetry-operator
  namespace: service-telemetry
spec:
  channel: stable-1.3
  installPlanApproval: Automatic
  name: service-telemetry-operator
  source: redhat-operators
  sourceNamespace: openshift-marketplace
EOF
```

2. To validate the Service Telemetry Operator and the dependent operators, enter the following command:

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

NAME                                  DISPLAY                                      VERSION
REPLACES                          PHASE
amq7-cert-manager.v1.0.0                  Red Hat Integration - AMQ Certificate Manager   1.0.0
Succeeded
amq7-interconnect-operator.v1.2.3           Red Hat Integration - AMQ Interconnect          1.2.3
amq7-interconnect-operator.v1.2.2   Succeeded
elastic-cloud-eck.v1.6.0                  Elasticsearch (ECK) Operator                  1.6.0
elastic-cloud-eck.v1.5.0           Succeeded
prometheusoperator.0.47.0                  Prometheus Operator                           0.47.0
prometheusoperator.0.37.0           Succeeded
service-telemetry-operator.v1.3.1622734200   Service Telemetry Operator
```

```
1.3.1622734200                          Succeeded
smart-gateway-operator.v3.0.1622734308    Smart Gateway Operator
3.0.1622734308                          Succeeded
```

3.2. OVERVIEW OF THE SERVICETELEMETRY OBJECT

To deploy the Service Telemetry Framework, you must create an instance of **ServiceTelemetry** in OCP. The **ServiceTelemetry** object is made up of the following major configuration parameters:

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

Each of these top-level configuration parameters provides various controls for a Service Telemetry Framework deployment.

IMPORTANT

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

3.2.1. backends

Use the **backends** parameter to control which storage backends are available for storage of metrics and events, and to control the enablement of Smart Gateways, as defined by the **clouds** parameter. For more information, see Section 3.2.2, "clouds".

Currently, you can use Prometheus as the metrics backend, and ElasticSearch as the events backend.

3.2.1.1. Enabling Prometheus as a storage backend for metrics

Procedure

- To enable Prometheus as a storage backend for metrics, configure the **ServiceTelemetry** object:

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

3.2.1.2. Enabling ElasticSearch as a storage backend for events

To enable events support in STF, you must enable the Elastic Cloud for Kubernetes Operator. For more information, see Section 3.1.7, "Subscribing to the Elastic Cloud on Kubernetes Operator" .

By default, ElasticSearch storage of events is disabled. For more information, see Section 3.1.1, "Deploying STF to the OpenShift environment with ElasticSearch".

Procedure

- To enable ElasticSearch as a storage backend for events, configure the **ServiceTelemetry** object:

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

3.2.2. clouds

Use the **clouds** parameter to control which Smart Gateway objects are deployed, thereby providing the interface for multiple monitored cloud environments to connect to an instance of STF. If a supporting backend 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 each cloud defined. Each cloud is made up of data types and collectors. Data types are **metrics** or **events**. Each data type is made up of a list of collectors and the message bus subscription address. Available collectors 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, providing subscriptions and data storage of metrics and events for both collectd and Ceilometer 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
```

```
events:
  collectors:
    - collectorType: collectd
      subscriptionAddress: collectd/notify
    - collectorType: ceilometer
      subscriptionAddress: anycast/ceilometer/event.sample
```

Each item of the **clouds** parameter represents a cloud instance. The cloud instances are made up of 3 top-level parameters: **name**, **metrics**, and **events**. The **metrics** and **events** parameters represent the corresponding backend 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 or Ceilometer. The **subscriptionAddress** parameter provides the AMQ Interconnect address to which a Smart Gateway should subscribe.

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

3.2.3. alerting

Use the **alerting** parameter to control creation of an Alertmanager instance and the configuration of the storage backend. By default, **alerting** is enabled. For more information, see Section 5.2, “Alerts”.

3.2.4. graphing

Use the **graphing** parameter to control the creation of a Grafana instance. By default, **graphing** is disabled. For more information, see Section 5.5, “Dashboards”.

3.2.5. highAvailability

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 5.4, “High availability”.

3.2.6. transports

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

3.3. CREATING A SERVICETELEMETRY OBJECT IN OCP

Create a **ServiceTelemetry** object in OCP to result in the creation of supporting components for a Service Telemetry Framework deployment. For more information, see Section 3.2, “Overview 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, you need to define only the parameter that you want to override. In this example, you 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 defaults. To override these defaults, add the configuration to the spec parameter:

```
apiVersion: infra.watch/v1beta1
kind: ServiceTelemetry
metadata:
  name: default
spec:
  alerting:
    enabled: true
    alertmanager:
      storage:
        strategy: persistent
        persistent:
          pvcStorageRequest: 20G
          storageSelector: {}
        receivers:
          snmpTraps:
            enabled: false
            target: 192.168.24.254
  backends:
    events:
      elasticsearch:
        enabled: false
        storage:
          strategy: persistent
          persistent:
            pvcStorageRequest: 20Gi
            storageSelector: {}
    metrics:
      prometheus:
        enabled: true
```

```
        scrapeInterval: 10s
        storage:
          strategy: persistent
          persistent:
            pvcStorageRequest: 20G
            storageSelector: {}
          retention: 24h
  graphing:
    enabled: false
    grafana:
      adminPassword: secret
      adminUser: root
      disableSignoutMenu: false
      ingressEnabled: false
  highAvailability:
    enabled: false
  transports:
    qdr:
      enabled: true
      web:
        enabled: false
  clouds:
    - name: cloud1
      metrics:
        collectors:
          - collectorType: collectd
            subscriptionAddress: collectd/telemetry
            debugEnabled: false
          - collectorType: ceilometer
            subscriptionAddress: anycast/ceilometer/metering.sample
            debugEnabled: false
      events:
        collectors:
          - collectorType: collectd
            subscriptionAddress: collectd/notify
            debugEnabled: false
          - collectorType: ceilometer
            subscriptionAddress: anycast/ceilometer/event.sample
            debugEnabled: false
```

2. To view the STF deployment logs in the Service Telemetry Operator, use the **oc logs** command:

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

```
--------------------------- Ansible Task Status Event StdOut -----------------

PLAY RECAP *********************************************************************
localhost                  : ok=54   changed=0    unreachable=0    failed=0    skipped=19
rescued=0    ignored=0
```

3. View the pods and the status of each pod to determine that all workloads are operating nominally:

NOTE

If you set **backends.events.elasticsearch.enabled: true**, the notification Smart Gateways reports **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-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
```

3.4. REMOVING STF FROM THE OCP ENVIRONMENT

Remove STF from an OCP environment if you no longer require the STF functionality.

Complete the following tasks:

1. Section 3.4.1, “Deleting the namespace”.
2. Section 3.4.2, “Removing the CatalogSource”.

3.4.1. Deleting the namespace

To remove the operational resources for STF from OCP, 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.
```

3.4.2. Removing the CatalogSource

If you do not expect to install Service Telemetry Framework again, delete the CatalogSource. When you remove the CatalogSource, PackageManifests related to STF are 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

For more information about the OperatorHub.io Community Catalog Source, see Section 3.1, “Deploying STF to the OCP environment”.

CHAPTER 4. COMPLETING THE SERVICE TELEMETRY FRAMEWORK CONFIGURATION

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

STF can support both single and multiple clouds, with the default configuration in Red Hat OpenStack Platform and STF set up for a single cloud installation. For a single Red Hat OpenStack Platform overcloud deployment with default configuration, see Section 4.1, "Configuring Red Hat OpenStack Platform overcloud for Service Telemetry Framework".

To plan your Red Hat OpenStack Platform installation and configuration STF for multiple clouds, see Section 4.4, "Configuring multiple clouds".

As part of an Red Hat OpenStack Platform overcloud deployment, you might need to configure additional features in your environment:

- To deploy data collection and transport to STF on Red Hat OpenStack Platform 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 to STF".

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

To configure the Red Hat OpenStack Platform overcloud, you must configure the data collection applications and the data transport to STF, and deploy the overcloud.

To configure the Red Hat OpenStack Platform overcloud, complete the following tasks:

1. Section 4.1.1, "Retrieving the AMQ Interconnect route address"
2. Section 4.1.2, "Creating the base configuration for STF"
3. Section 4.1.3, "Configuring the STF connection for the overcloud"
4. Section 4.1.4, "Deploying the overcloud"
5. Section 4.1.5, "Validating client-side installation"

Additional resources

- To collect data through AMQ Interconnect, see The amqp1 plug-in in the *Monitoring Tools Configuration* guide.

4.1.1. Retrieving the AMQ Interconnect route address

When you configure the Red Hat OpenStack Platform overcloud for 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 (OCP) 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
```

NOTE

If your STF installation differs from the documentation, ensure that you retrieve the correct AMQ Interconnect route address.

4.1.2. Creating the base configuration for STF

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

Procedure

1. Log in to the Red Hat OpenStack Platform 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 Red Hat OpenStack Platform legacy 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 legacy components, and should be increased to a larger value such as **300**. Increasing the value to a longer polling interval will result in less telemetry resolution in STF.

To enable collection of telemetry with STF and Gnocchi, see Section 4.2, “Sending metrics to Gnocchi and to STF”

```
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
  CollectdSensubilityResultsChannel: sensubility/telemetry
```

```
# 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
    collectd::plugin::amqp1::send_queue_limit: 50

    # 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 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 configuration is set up 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 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 the AMQ Interconnect on the overcloud to the STF deployment.
   - Replace the **host** parameter with the value of **HOST/PORT** that you retrieved in Section 4.1.1, "Retrieving the AMQ Interconnect route address".

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

    MetricsQdrSSLProfiles:
    -   name: sslProfile
```

4.1.4. Deploying the overcloud

Deploy or update the overcloud with the required environment files to result in data being collected and transmitted to STF.

Procedure

1. Log in to the Red Hat OpenStack Platform 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 Red Hat OpenStack Platform director deployment to setup data collection and AMQ Interconnect:

- the **collectd-write-qdr.yaml** file to ensure that collectd telemetry and events are sent to STF
- 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 set up correctly
- the **stf-connectors.yaml** environment file to define the connection to STF

```
(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/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
```

4. Deploy the Red Hat OpenStack Platform overcloud.

4.1.5. Validating client-side installation

To validate data collection from the STF storage domain, query the data sources for delivered data. To validate individual nodes in the Red Hat OpenStack Platform deployment, connect to the console using SSH.

TIP

Some telemetry data is only available when Red Hat OpenStack Platform has active workloads.

Procedure

1. Log in to an overcloud node, for example, controller-0.
2. Ensure that **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 being 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      0
  endpoint  out  1       8                                 250  0    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      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.lSY6Mcicol4J2Kp      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 Red Hat OpenStack Platform nodes to STF, connect to OCP to get 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 run the **qdstat --connections** command to list the known connections:

```
$ 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
```

In this example, there are three **edge** connections from the Red Hat OpenStack Platform nodes with connection **id** 22, 23, and 24.

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 TO STF

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.

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 verify 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 16.1 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.

The deployment of a spine-leaf topology involves creating roles and networks, then assigning those networks to the available roles. When you configure data collection and transport for STF for an Red Hat OpenStack Platform 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 Red Hat OpenStack Platform 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** where **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')}"
   ```

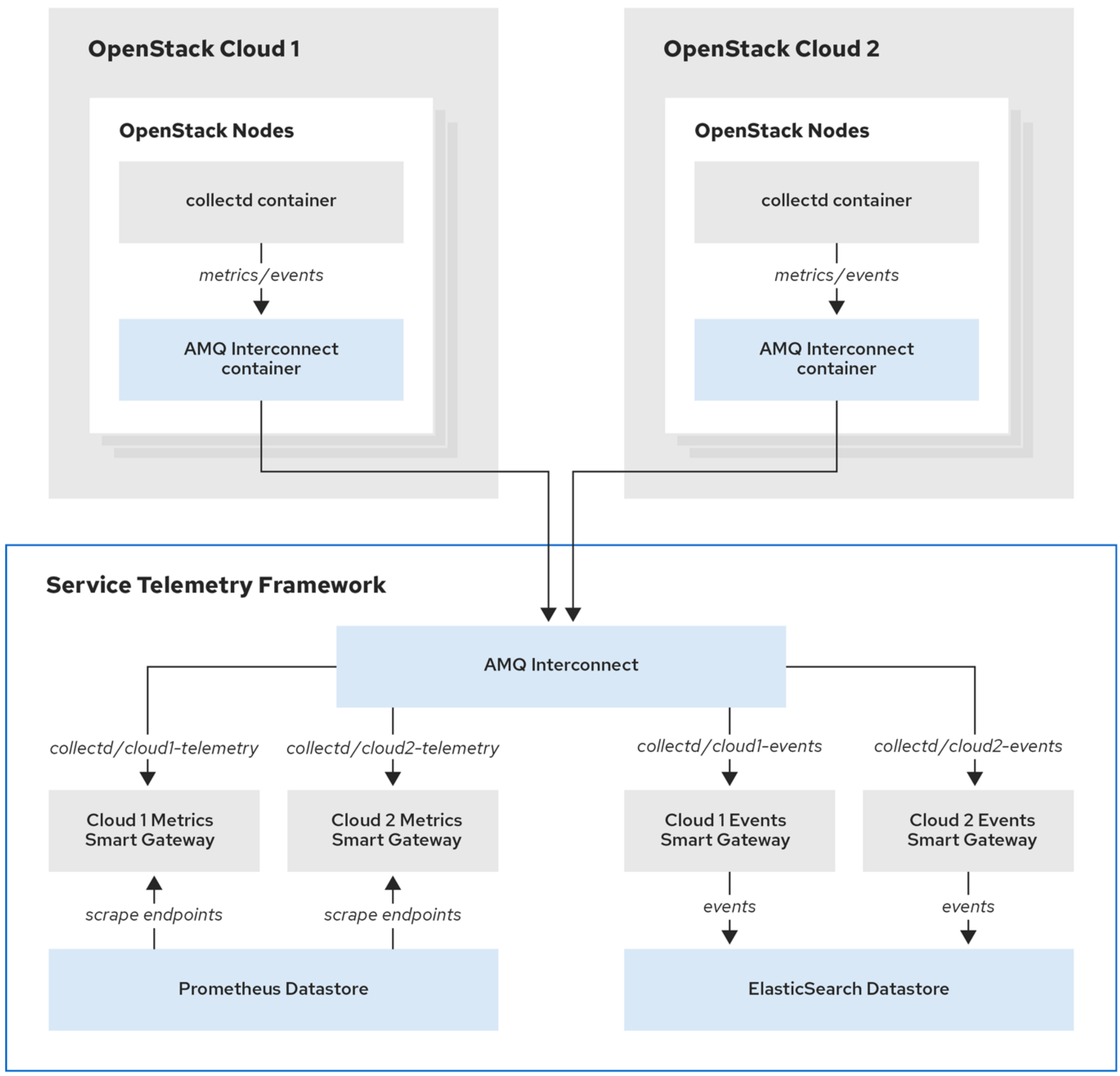
This example configuration is on a CephStorage leaf role:

```
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 clouds to target a single instance of Service Telemetry Framework (STF). When configuring 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 in order 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 created by each of the Smart Gateways.

Figure 4.1. Two Red Hat OpenStack Platform clouds connect to STF



To configure the Red Hat OpenStack Platform 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 OpenStack environment file for multiple clouds" .

4.4.1. Planning AMQP address prefixes

By default, Red Hat OpenStack Platform 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 monitoring 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 deploying 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 under **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 Section 3.2.2, "clouds".

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 can cause the maximum pod name of 63 characters to be exceeded. Ensure that the combination of the **ServiceTelemetry** name **default** and the **clouds.name** does not exceed 19 characters. Topic addresses have no character limitation and can be different than 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 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 have been 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 to be managed by the Service Telemetry Operator. For more information, see Section 3.2.2, “clouds”.

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 router connections from Red Hat OpenStack Platform 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:

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

3. Add this environment file to your deployment. For more information, see Section 4.4.5, "Creating the OpenStack environment file for multiple clouds" and Core overcloud parameters in the *Overcloud Parameters* guide.

4.4.5. Creating the OpenStack 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 5.7, "Monitoring container health and API status".

Prerequisites

- 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".
- You have created the **enable-stf.yaml** to set environment defaults for STF. For more information, see Section 4.1.2, "Creating the base configuration for STF".

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 (1)

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: true

  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, along 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 validating 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 attached 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. ADVANCED FEATURES

The following optional features can provide additional functionality to the Service Telemetry Framework (STF):

- Section 5.1, “Customizing the deployment”
- Section 5.2, “Alerts”
- Section 5.3, “Configuring SNMP Traps”
- Section 5.4, “High availability”
- Section 5.5, “Dashboards”
- Section 5.8, “Ephemeral storage”
- Section 5.6, “Monitoring the resource usage of Red Hat OpenStack Platform services”
- Section 5.9, “Creating a route in Red Hat OpenShift Container Platform”

5.1. CUSTOMIZING THE DEPLOYMENT

The Service Telemetry Operator watches for a **ServiceTelemetry** manifest to load into Red Hat OpenShift Container Platform (OCP). The Operator then creates other objects in memory, which results in the dependent Operators creating the workloads they are responsible for managing.

> **WARNING**
>
> When you override the manifest, you must provide the entire manifest contents, including object names or namespaces. There is no dynamic parameter substitution when you override a manifest.
>
> Use manifest overrides only as a last resort short circuit.

To override a manifest successfully with Service Telemetry Framework (STF), deploy a default environment using the core options only. For more information about the core options, see Section 3.3, “Creating a ServiceTelemetry object in OCP”. When you deploy STF, use the **oc get** command to retrieve the default deployed manifest. When you use a manifest that was originally generated by Service Telemetry Operator, the manifest is compatible with the other objects that are managed by the Operators.

For example, when the **backends.metrics.prometheus.enabled: true** parameter is configured in the **ServiceTelemetry** manifest, the Service Telemetry Operator requests components for metrics retrieval and storage using the default manifests. In some cases, you might want to override the default manifest. For more information, see Section 5.1.1, “Manifest override parameters”.

5.1.1. Manifest override parameters

This table describes the available parameters that you can use to override a manifest, along with the corresponding retrieval commands.

Table 5.1. Manifest override parameters

Override parameter	Description	Retrieval command
alertmanagerManifest	Override the **Alertmanager** object creation. The Prometheus Operator watches for **Alertmanager** objects.	**oc get alertmanager default -oyaml**
alertmanagerConfigManifest	Override the **Secret** that contains the Alertmanager configuration. The Prometheus Operator uses a secret named **alertmanager-{{ alertmanager-name }}**, for example, **default**, to provide the **alertmanager.yaml** configuration to Alertmanager.	**oc get secret alertmanager-default -oyaml**
elasticsearchManifest	Override the **ElasticSearch** object creation. The Elastic Cloud on Kubernetes Operator watches for **ElasticSearch** objects.	**oc get elasticsearch elasticsearch -oyaml**
interconnectManifest	Override the **Interconnect** object creation. The AMQ Interconnect Operator watches for **Interconnect** objects.	**oc get interconnect default-interconnect -oyaml**
prometheusManifest	Override the **Prometheus** object creation. The Prometheus Operator watches for **Prometheus** objects.	**oc get prometheus default -oyaml**
servicemonitorManifest	Override the **ServiceMonitor** object creation. The Prometheus Operator watches for **ServiceMonitor** objects.	**oc get servicemonitor default -oyaml**

5.1.2. Overriding a managed manifest

Edit the **ServiceTelemetry** object and provide a parameter and manifest. For a list of available manifest override parameters, see Section 5.1, "Customizing the deployment". The default **ServiceTelemetry** object is **default**. Use **oc get servicetelemetry** to list the available STF deployments.

TIP

The **oc edit** command loads the default system editor. To override the default editor, pass or set the environment variable **EDITOR** to the preferred editor. For example, **EDITOR=nano oc edit servicetelemetry default**.

Procedure

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

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

3. Load the **ServiceTelemetry** object into an editor:

```
$ oc edit servicetelemetry default
```

4. To modify the **ServiceTelemetry** object, provide a manifest override parameter and the contents of the manifest to write to OCP instead of the defaults provided by STF.

> **NOTE**
>
> The trailing pipe (|) after entering the manifest override parameter indicates that the value provided is multi-line.

```
$ oc edit stf default

apiVersion: infra.watch/v1beta1
kind: ServiceTelemetry
metadata:
  ...
spec:
  alertmanagerConfigManifest: | (1)
    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' (2)
status:
  ...
```

1. Manifest override parameter is defined in the **spec** of the **ServiceTelemetry** object.
2. End of the manifest override content.

5. Save and close.

5.2. ALERTS

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 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 5.2.1, "Creating an alert rule in Prometheus".
2. Create an alert route in Alertmanager. For more information, see Section 5.2.3, "Creating an alert route 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

5.2.1. 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: Metric Listener down
          expr: collectd_qpid_router_status < 1 (1)
EOF
```

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

4. To verify that the rules have been loaded into Prometheus by the Operator, create a pod with access to **curl**:

```
$ oc run curl --generator=run-pod/v1 --image=radial/busyboxplus:curl -i --tty
```

5. Run **curl** to access the **prometheus-operated** service to return the rules loaded into memory:

```
[ root@curl:/ ]$ curl prometheus-operated:9090/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":[{"name":"Metric Listener
down","query":"collectd_qpid_router_status \u003c 1","duration":0,"labels":{},"annotations":
{},"alerts":[],"health":"ok","type":"alerting"}],"interval":30}]}}
```

6. To verify that the output shows the rules loaded into the **PrometheusRule** object, for example the output contains the defined **./openstack.rules**, exit from the pod:

```
[ root@curl:/ ]$ exit
```

7. Clean up the environment by deleting the **curl** pod:

```
$ oc delete pod curl

pod "curl" deleted
```

Additional resources

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

5.2.2. Configuring custom alerts

You can add custom alerts to the **PrometheusRule** object that you created in Section 5.2.1, “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.

Additional resources

- For more information about configuring 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

5.2.3. Creating an 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 an Red Hat OpenShift Container Platform (OCP) secret. STF by default 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, an **alertmanagerConfigManifest** parameter must be passed to the Service Telemetry Operator that results in an updated secret, managed by the Prometheus Operator.

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

NOTE

This step loads the default template that is already managed by Service Telemetry Operator. 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 **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 was 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. To verify the configuration has been loaded into Alertmanager, create a pod with access to **curl**:

```
$ oc run curl --generator=run-pod/v1 --image=radial/busyboxplus:curl -i --tty
```

7. Run **curl** against the **alertmanager-operated** service to retrieve the status and **configYAML** contents and review the supplied configuration matches the configuration loaded into Alertmanager:

```
[ root@curl:/ ]$ curl alertmanager-operated:9093/api/v1/status

{"status":"success","data":{"configYAML":"global:\n  resolve_timeout: 10m\n  http_config: {}\n
smtp_hello: localhost\n  smtp_require_tls: true\n  pagerduty_url:
https://events.pagerduty.com/v2/enqueue\n  hipchat_api_url: https://api.hipchat.com/\n
opsgenie_api_url: https://api.opsgenie.com/\n  wechat_api_url:
https://qyapi.weixin.qq.com/cgi-bin/\n  victorops_api_url:
https://alert.victorops.com/integrations/generic/20131114/alert/\nroute:\n  receiver: \"null\"\n
group_by:\n  - job\n  group_wait: 30s\n  group_interval: 5m\n  repeat_interval:
12h\nreceivers:\n- name: \"null\"\ntemplates: []\n",...}}
```

8. Verify that the **configYAML** field contains the expected changes. Exit from the pod:

```
[ root@curl:/ ]$ exit
```

9. 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 Alerting.

5.3. CONFIGURING SNMP TRAPS

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

For more information about configuring alerts, see Section 5.2, “Alerts”.

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.

5.4. HIGH AVAILABILITY

High availability is the ability of Service Telemetry Framework (STF) to rapidly recover from failures in its component services. Although Red Hat OpenShift Container Platform (OCP) 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, reducing recovery time to approximately 2 seconds. To protect against failure of an OCP node, deploy STF to an OCP 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.

5.4.1. Configuring high availability

To configure STF for high availability, add **highAvailability.enabled: true** to the ServiceTelemetry object in OCP. You can this 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.

5.5. DASHBOARDS

Use third-party application Grafana to visualize system-level metrics gathered by collectd for each individual host node.

For more information about configuring collectd, see Section 4.1, "Configuring 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 for monitoring service resource usage, API stats, and cloud events. You must enable API health monitoring and service monitoring to provide the data for this dashboard.

- For more information about API health monitoring, see Section 5.7, "Monitoring container health and API status".
- For more information about service monitoring, see Section 5.6, "Monitoring the resource usage of Red Hat OpenStack Platform services".

5.5.1. Setting up 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. Using the Service Telemetry Operator to deploy Grafana results in a Grafana instance and the configuration of the default data sources for the local STF deployment.

Prerequisites

Enable OperatorHub.io catalog source for the Grafana Operator. For more information, see Section 3.1.5, "Enabling the OperatorHub.io Community Catalog Source" .

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. To verify that the operator launched successfully, run the **oc get csv** command. If the value of the PHASE column is **Succeeded**, the operator launched successfully:

```
$ oc get csv

NAME                          DISPLAY                          VERSION   REPLACES
PHASE
grafana-operator.v3.2.0            Grafana Operator                        3.2.0
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 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
```

5.5.2. 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:

```
$ 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
```

> **WARNING**
>
> Some panels in the cloud dashboard require that you set 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.
>
> ```
> parameter_defaults:
>     ExtraConfig:
>         collectd::plugin::virt::hostname_format: name uuid hostname
> ```

4. Verify that the dashboards are available:

```
$ oc get grafanadashboards

NAME                 AGE
rhos-dashboard       7d21h
rhos-cloud-dashboard 7d21h
```

5. Retrieve the Grafana route address:

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

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

6. Navigate to https://*<grafana_route_address>* in a web browser. Replace *<grafana_route_address>* with the value that you retrieved in the previous step.

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

5.5.3. Retrieving and setting Grafana login credentials

STF sets default login credentials when Grafana is enabled. The credentials can be overridden 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. To retrieve the default username and password, describe the Grafana object:

```
$ oc describe grafana default
```

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.

5.6. MONITORING THE RESOURCE USAGE OF RED HAT OPENSTACK PLATFORM SERVICES

Monitor the resource usage of the Red Hat OpenStack Platform services, such as the APIs and other infrastructure processes, to identify bottlenecks in the overcloud by showing services running out of compute power. Enable the **collectd-libpod-stats** plugin to gather CPU and memory usage metrics for every container running in the overcloud.

Prerequisites

- You have created the **stf-connectors.yaml** file. For more information, see Section 4.1, "Configuring Red Hat OpenStack Platform overcloud for Service Telemetry Framework".
- You are using the most current version of Red Hat OpenStack Platform: 16.1.

Procedure

To disable the monitoring of container resource usage disable **CollectdEnableLibpodstats**.

1. Open the **stf-connectors.yaml** file and add the **CollectdEnableLibpodstats** parameter to override the setting in **enable-stf.yaml**

```
  CollectdEnableLibpodstats: false
```

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

5.7. MONITORING CONTAINER HEALTH AND API STATUS

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

To monitor healthchecks in Service Telemetry Framework (STF), you must enable and configure the **collectd-sensubility** plugin to work with the **amqp1** protocol and configure the the amqp1 address. Container health monitoring is enabled by default in **enable-stf.yaml**. For more information, see Section 4.1.2, "Creating the base configuration for STF" .

Prerequisites

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

- You are using the most current version of Red Hat OpenStack Platform 16.1.

Procedure

To disable the monitoring of container health status disable **CollectdEnableSensubility**.

1. Open the **stf-connectors.yaml** and add the **CollectdEnableSensubility** parameter to override the setting in **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".

5.8. EPHEMERAL STORAGE

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

5.8.1. 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 backend, 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.

5.9. CREATING A ROUTE IN RED HAT OPENSHIFT CONTAINER PLATFORM

In Red Hat OpenShift Container Platform, you can expose applications to the external network via a route. For more information, see Configuring ingress cluster traffic.

In Service Telemetry Framework (STF), routes are not exposed by default to limit the attack surface of STF deployments. To access some services deployed in STF, you must expose the services in OCP for access.

A common service to expose in STF is Prometheus, as shown in the following example:

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 services in the **service-telemetry** project:

```
$ oc get services
```

```
NAME                                   TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
alertmanager-operated                    ClusterIP   None            <none>
9093/TCP,9094/TCP,9094/UDP                  93m
default-cloud1-ceil-meter-smartgateway   ClusterIP   172.30.114.195   <none>        8081/TCP
93m
default-cloud1-coll-meter-smartgateway   ClusterIP   172.30.133.180   <none>        8081/TCP
93m
default-interconnect                   ClusterIP   172.30.3.241     <none>
5672/TCP,8672/TCP,55671/TCP,5671/TCP,5673/TCP   93m
ibm-auditlogging-operator-metrics        ClusterIP   172.30.216.249   <none>
8383/TCP,8686/TCP                         11h
prometheus-operated                    ClusterIP   None            <none>        9090/TCP
93m
service-telemetry-operator-metrics       ClusterIP   172.30.11.66     <none>
8383/TCP,8686/TCP                         11h
smart-gateway-operator-metrics           ClusterIP   172.30.145.199   <none>
8383/TCP,8686/TCP                         11h
```

4. Take note of the port and service name to expose as a route, for example, service **prometheus-operated** and port **9090**.

5. Expose the **prometheus-operated** service as an edge route and redirect insecure traffic to the secure endpoint of port **9090**:

```
$ oc create route edge metrics-store --service=prometheus-operated --insecure-policy="Redirect" --port=9090
```

```
route.route.openshift.io/metrics-store created
```

6. To verify and find the exposed external DNS for the route, use the **oc get route** command:

```
$ oc get route metrics-store -ogo-template='{{.spec.host}}'
```

```
metrics-store-service-telemetry.apps.infra.watch
```

The **prometheus-operated** service is now available at the exposed DNS address, for example, https://metrics-store-service-telemetry.apps.infra.watch

> **NOTE**
>
> The address of the route must be resolvable and configuration is environment specific.

Additional resources

- For more information about Red Hat OpenShift Container Platform networking, see Understanding networking
- For more information about route configuration, see Route configuration
- For more information about ingress cluster traffic, see Configuring ingress cluster traffic overview

CHAPTER 6. UPGRADING SERVICE TELEMETRY FRAMEWORK TO VERSION 1.3

To migrate from Service Telemetry Framework (STF) v1.2 to STF v1.3, you must replace the **ClusterServiceVersion** and **Subscription** objects in the **service-telemetry** namespace on your Red Hat OpenShift Container Platform (OCP) environment.

Prerequisites

- You have upgraded your OCP environment to v4.6. STF v1.3 does not run on OCP v4.5 and lower.STF v1.2 does not run on OCP v4.7 and higher.
- You have backed up your data prior to any upgrade of the environment. Upgrading STF v1.2 to v1.3 results in a brief outage while the Smart Gateways are upgraded. Additionally, changes to the **ServiceTelemetry** and **SmartGateway** objects do not have any effect while the Operators are being replaced.

To upgrade from STF v1.2 to v1.3, complete the following procedures:

Procedure

1. Remove the STF 1.2 Operators.
2. Subscribe to the Service Telemetry Operator.

6.1. REMOVING SERVICE TELEMETRY FRAMEWORK 1.2 OPERATORS

Remove the Operators from STF v1.2, Smart Gateway Operator, and Service Telemetry Operator.

> **NOTE**
>
> You must temporarily remove the **clouds** parameters because of changes in the API interface. This results in the removal of all Smart Gateways until after the upgrade and the inability to deliver metrics and events during the upgrade.

Procedure

1. Retrieve the current **ServiceTelemetry** object and note the contents, in particular the **clouds** parameter because you must remove this parameter before you upgrade the Operators.

   ```
   $ oc get stf default -oyaml
   ```

2. Modify the ServiceTelemetry object to clear the **clouds** parameter and set it to an empty list. Set **cloudsRemoveOnMissing** to **true** to remove all Smart Gateways.

> **WARNING**
>
> This command stops all monitoring functions until after the upgrade is completed and the **clouds** object is redefined. If you use the default clouds configuration, it is not defined in your ServiceTelemetry object.

```
$ oc patch stf default --patch $'spec:\n  clouds: []\n  cloudsRemoveOnMissing: true' --
type=merge
```

3. Monitor the Smart Gateway pods until they are fully terminated and removed:

```
$ oc get pods --selector app=smart-gateway --watch

NAME                                                  READY   STATUS    RESTARTS   AGE
default-cloud1-ceil-meter-smartgateway-58cc854f4-hgk92    1/1     Running   0          2m42s
default-cloud1-coll-meter-smartgateway-6c76f9786d-crn9b   2/2     Running   0          2m55s
default-cloud1-coll-meter-smartgateway-6c76f9786d-crn9b   2/2     Terminating   0
3m12s
default-cloud1-ceil-meter-smartgateway-58cc854f4-hgk92    1/1     Terminating   0          3m
...
```

4. Retrieve the **Subscription** name of the Smart Gateway Operator:

```
$ oc get sub smart-gateway-operator-stable-1.2-redhat-operators-openshift-marketplace

NAME                                                                       PACKAGE              SOURCE
CHANNEL
smart-gateway-operator-stable-1.2-redhat-operators-openshift-marketplace   smart-gateway-
operator   redhat-operators   stable-1.2
```

5. Delete the Smart Gateway Operator subscription:

```
$ oc delete sub smart-gateway-operator-stable-1.2-redhat-operators-openshift-marketplace

subscription.operators.coreos.com "smart-gateway-operator-stable-1.2-redhat-operators-
openshift-marketplace" deleted
```

6. Retrieve the Smart Gateway Operator ClusterServiceVersion:

```
$ oc get csv -o name | grep -E 'smart-gateway'

clusterserviceversion.operators.coreos.com/smart-gateway-operator.v2.2.1623675667
```

7. Delete the Smart Gateway Operator ClusterServiceVersion:

```
$ oc delete clusterserviceversion.operators.coreos.com/smart-gateway-
operator.v2.2.1623675667
```

```
clusterserviceversion.operators.coreos.com "smart-gateway-operator.v2.2.1623675667"
deleted
```

8. Delete the SmartGateway Custom Resource Definition:

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

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

9. Patch the Service Telemetry Operator Subscription to use the stable-1.3 channel:

```
$ oc patch sub service-telemetry-operator --patch $'spec:\n  channel: stable-1.3' --
type=merge

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

10. Monitor the output of the **oc get csv** command until the Smart Gateway Operator is installed and Service Telemetry Operator is **Pending** for version 1.2 and 1.3:

```
$ oc get csv

NAME                                    DISPLAY                                         VERSION
REPLACES                                    PHASE
amq7-cert-manager.v1.0.0                     Red Hat Integration - AMQ Certificate Manager   1.0.0
Succeeded
amq7-interconnect-operator.v1.2.4            Red Hat Integration - AMQ Interconnect         1.2.4
amq7-interconnect-operator.v1.2.3            Succeeded
elastic-cloud-eck.v1.6.0                  Elasticsearch (ECK) Operator                  1.6.0
elastic-cloud-eck.v1.5.0                  Succeeded
prometheusoperator.0.47.0                    Prometheus Operator                           0.47.0
prometheusoperator.0.37.0                    Succeeded
service-telemetry-operator.v1.2.1623675667   Service Telemetry Operator
1.2.1623675667                               Pending
service-telemetry-operator.v1.3.1622734200   Service Telemetry Operator
1.3.1622734200   service-telemetry-operator.v1.2.1623675667   Pending
smart-gateway-operator.v3.0.1622734308       Smart Gateway Operator
3.0.1622734308                               Succeeded
```

11. Delete the Service Telemetry Operator v1.2 ClusterServiceVersion:

```
$ oc delete csv service-telemetry-operator.v1.2.1623675667

clusterserviceversion.operators.coreos.com "service-telemetry-operator.v1.2.1623675667"
deleted
```

12. Edit the ServiceTelemetry object and insert the contents of your previously noted **clouds** parameter. If the **clouds** parameter was not previously defined because you used the default Smart Gateway instances, remove the **clouds: []** parameter.

```
$ oc edit stf default
```

13. Validate that the Smart Gateways have been restored:

```
$ oc get pods --selector app=smart-gateway

NAME                                                READY   STATUS    RESTARTS   AGE
default-cloud1-ceil-meter-smartgateway-6484b98b68-sl7mb   2/2     Running   0          5m56s
default-cloud1-coll-meter-smartgateway-799f687658-nfzr6   2/2     Running   0          6m6s
```

6.2. SUBSCRIBING TO THE SERVICE TELEMETRY OPERATOR

You must subscribe to the Service Telemetry Operator, which manages the STF instances.

Procedure

1. To create the Service Telemetry Operator subscription, enter the **oc create -f** command:

```
$ oc create -f - <<EOF
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: service-telemetry-operator
  namespace: service-telemetry
spec:
  channel: stable-1.3
  installPlanApproval: Automatic
  name: service-telemetry-operator
  source: redhat-operators
  sourceNamespace: openshift-marketplace
EOF
```

2. To validate the Service Telemetry Operator and the dependent operators, enter the following command:

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

NAME                                   DISPLAY                                        VERSION
REPLACES                               PHASE
amq7-cert-manager.v1.0.0               Red Hat Integration - AMQ Certificate Manager   1.0.0
Succeeded
amq7-interconnect-operator.v1.2.3      Red Hat Integration - AMQ Interconnect          1.2.3
amq7-interconnect-operator.v1.2.2   Succeeded
elastic-cloud-eck.v1.6.0               Elasticsearch (ECK) Operator                   1.6.0
elastic-cloud-eck.v1.5.0          Succeeded
prometheusoperator.0.47.0              Prometheus Operator                            0.47.0
prometheusoperator.0.37.0          Succeeded
service-telemetry-operator.v1.3.1622734200   Service Telemetry Operator
1.3.1622734200                         Succeeded
smart-gateway-operator.v3.0.1622734308       Smart Gateway Operator
3.0.1622734308                         Succeeded
```

When the new Operators start, they reconcile the existing **ServiceTelemetry** and **SmartGateway** objects, resulting in the restart of Smart Gateway containers.

- To check the state of the Smart Gateway containers, use the **oc get pods** command:

```
oc get pods
```

```
NAME                                          READY   STATUS        RESTARTS   AGE
...
default-cloud1-ceil-meter-smartgateway-5849c4cdb5-xgl42   1/1     Running       0          35s
default-cloud1-coll-meter-smartgateway-749674f75c-k7pm7   2/2     Terminating   0
56m
default-cloud1-coll-meter-smartgateway-868476456b-ksh9b   2/2     Running       0          26s
...
```

CHAPTER 7. COLLECTD PLUGINS

IMPORTANT

Red Hat is currently updating the plugin information in this guide for this release.

You can configure multiple collectd plugins to suit your Red Hat OpenStack Platform 16.1 environment. Reference the tables of available parameters for specific plugins, such as in the following example:

```
ExtraConfig:
  collectd::plugin::example_plugin::<parameter>: <value>
```

Reference the metrics tables of specific plugins for Prometheus or Grafana queries.

collectd::plugin::aggregation

You can aggregate several values into one with the **aggregation** plugin. Use the aggregation functions such as **sum**, **average**, **min**, and **max** to calculate metrics, for example average and total CPU statistics.

- collectd::plugin::aggregation::aggregators
- collectd::plugin::aggregation::interval

collectd::plugin::ampq
collectd::plugin::amqp1

Use the **amqp1** plugin to write values to an amqp1 message bus, for example, AMQ Interconnect.

Table 7.1. amqp1 parameters

Parameter	Type
manage_package	Boolean
transport	String
host	string
port	integer
user	String
password	String
address	String
instances	Hash
retry_delay	Integer
send_queue_limit	Integer

Parameter	Type
interval	Integer

Example configuration

```
Parameter_defaults:
  CollectdExtraPlugins:
    - amqp1
  ExtraConfig:
    collectd::plugin::amqp1::send_queue_limit: 50
```

collectd::plugin::apache

Use the **apache** plugin to collect Apache data.

Table 7.2. apache parameters

Parameter	Type
instances	Hash
interval	Integer
manage-package	Boolean
package_install_options	List

Example configuration

```
parameter_defaults:
    ExtraConfig:
        collectd::plugin::apache:
          localhost:
              url: "http://10.0.0.111/status?auto"
```

Additional resources

For more information about configuring the **apache** plugin, see apache.

collectd::plugin::battery

Use the **battery** plugin to report the remaining capacity, power, or voltage of laptop batteries.

Table 7.3. battery parameters

Parameter	Type
values_percentage	Boolean

Parameter	Type
report_degraded	Boolean
query_state_fs	Boolean
interval	Integer

Additional resources

For more information about configuring the **battery** plugin, see battery.

collectd::plugin::bind

Use the **bind** plugin to retrieve encoded statistics about queries and responses from a DNS server. The plugin submits the values to collectd.

collectd::plugin::ceph

Use the **ceph** plugin to gather data from ceph daemons.

Table 7.4. ceph parameters

Parameter	Type
daemons	Array
longrunavglatency	Boolean
convertspecialmetrictypes	Boolean
manage_package	Boolean
package_name	String

Example configuration

```
parameter_defaults:
    ExtraConfig:
        collectd::plugin::ceph::daemons:
          - ceph-osd.0
          - ceph-osd.1
          - ceph-osd.2
          - ceph-osd.3
          - ceph-osd.4
```

> **NOTE**
>
> If an Object Storage Daemon (OSD) is not on every node, you must list the OSDs.

NOTE

When you deploy collectd, the **ceph** plugin is added to the ceph nodes. Do not add the **ceph** plugin on ceph nodes to **CollectdExtraPlugins**, because this results in a deployment failure.

Additional resources

For more information about configuring the **ceph** plugin, see ceph.

collectd::plugins::cgroups

Use the **cgroups** plugin to collect information for processes in a cgroup.

Table 7.5. cgroups parameters

Parameter	Type
ignore_selected	Boolean
interval	Integer
cgroups	List

Additional resources

For more information about configuring the **cgroups** plugin, see cgroups.

collectd::plugin::connectivity

Use the connectivity plugin to monitor the state of network interfaces.

NOTE

If no interfaces are listed, all interfaces are monitored by default.

Table 7.6. connectivity parameters

Parameter	Type
interfaces	Array

Example configuration

```
parameter_defaults:
  ExtraConfig:
    collectd::plugin::connectivity::interfaces:
    - eth0
    - eth1
```

Additional resources

For more information about configuring the **connectivity** plugin, see connectivity.

collectd::plugin::conntrack

Use the **conntrack** plugin to track the number of entries in the Linux connection-tracking table. There are no parameters for this plugin.

collectd::plugin::contextswitch

Use the **ContextSwitch** plugin to collect the number of context switches handled by the system.

Additional resources

For more information about configuring the **contextswitch** plugin, see contextswitch.

collectd::plugin::cpu

Use the **cpu** plugin to monitor the time the CPU spends in various states, for example, idle, executing user code, executing system code, waiting for IO-operations, and other states.

The **cpu** plugin collects **_jiffies_**, not percentage values. The value of a jiffy depends on the clock frequency of your hardware platform, and therefore is not an absolute time interval unit.

To report a percentage value, set the Boolean parameters **reportbycpu** and **reportbystate** to **true**, and then set the Boolean parameter **valuespercentage** to true.

Table 7.7. cpu metrics

Name	Description	Query
idle	Amount of idle time	collectd_cpu_total{...,type_instance=*idle*}
interrupt	CPU blocked by interrupts	collectd_cpu_total{...,type_instance=*interrupt*}
nice	Amount of time running low priority processes	collectd_cpu_total{...,type_instance=*nice*}
softirq	Amount of cycles spent in servicing interrupt requests	collectd_cpu_total{...,type_instance=*waitirq*}
steal	The percentage of time a virtual CPU waits for a real CPU while the hypervisor is servicing another virtual processor	collectd_cpu_total{...,type_instance=*steal*}
system	Amount of time spent on system level (kernel)	collectd_cpu_total{...,type_instance=*system*}
user	Jiffies used by user processes	collectd_cpu_total{...,type_instance=*user*}
wait	CPU waiting on outstanding I/O request	collectd_cpu_total{...,type_instance=*wait*}

Table 7.8. cpu parameters

Parameter	Type
reportbystate	Boolean
valuespercentage	Boolean
reportbycpu	Boolean
reportnumcpu	Boolean
reportgueststate	Boolean
subtractgueststate	Boolean
interval	Integer

Example configuration

```
parameter_defaults:
   CollectdExtraPlugins:
     - cpu
   ExtraConfig:
      collectd::plugin::cpu::reportbystate: true
```

Additional resources

For more information about configuring the **cpu** plugin, see cpu.

collectd::plugin::cpufreq

- None

collectd::plugin::cpusleep
collectd::plugin::csv

- collectd::plugin::csv::datadir
- collectd::plugin::csv::storerates
- collectd::plugin::csv::interval

collectd::plugin::curl_json
collectd::plugin::curl
collectd::plugin::curl_xml
collectd::plugin::dbi
collectd::plugin::df

Use the **df** plugin to collect disk space usage information for file systems.

Table 7.9. df metrics

Name	Description	Query
free	Amount of free disk space	collectd_df_df_complex{..., type_instance="free"}
reserved	Amount of reserved disk space	collectd_df_df_complex{..., type_instance="reserved"}
used	Amount of used disk space	collectd_df_df_complex{..., type_instance="used"}

Table 7.10. df parameters

Parameter	Type
devices	Array
fstypes	Array
ignoreselected	Boolean
mountpoints	Array
reportbydevice	Boolean
reportinodes	Boolean
reportreserved	Boolean
valuesabsolute	Boolean
valuespercentage	Boolean

Example configuration

```
parameter_defaults:
  CollectdExtraPlugins:
   - df
  ExtraConfig:
    collectd::plugin::df::FStype: "ext4"
```

Additional resources

For more information about configuring the **df** plugin, see df.

collectd::plugin::disk

Use the **disk** plugin to collect performance statistics of hard-disks and, if supported, partitions. This plugin is enabled by default.

Table 7.11. disk parameters

Parameter	Type
disks	Array
ignoreselected	Boolean
udevnameattr	String

Table 7.12. disk metrics

Name	Description
merged	The number of operations that can be merged together, already queued operations, for example, one physical disk access served two or more logical operations.
time	The average time an I/O-operation takes to complete. Since this is a little messy to calculate take the actual values with a grain of salt.
io_time	Time spent doing I/Os (ms). You can make use of this metric as a device load percentage. A value of 1 second matches 100% of load.
weighted_io_time	Measure of both I/O completion time and the backlog that might be accumulating.
pending_operations	Shows queue size of pending I/O operations.

Example configuration

```
parameter_defaults:
   ExtraConfig:
      collectd::plugin::disk::disk: "sda"
      collectd::plugin::disk::ignoreselected: false
```

Additional resources

For more information about configuring the **disk** plugin, see disk.

collectd::plugin::dns
collectd::plugin::dpdk_telemetry
collectd::plugin::entropy

- collectd::plugin::entropy::interval

collectd::plugin::ethstat

- collectd::plugin::ethstat::interfaces
- collectd::plugin::ethstat::maps

- collectd::plugin::ethstat::mappedonly
- collectd::plugin::ethstat::interval

collectd::plugin::exec

- collectd::plugin::exec::commands
- collectd::plugin::exec::commands_defaults
- collectd::plugin::exec::globals
- collectd::plugin::exec::interval

collectd::plugin::fhcount

- collectd::plugin::fhcount::valuesabsolute
- collectd::plugin::fhcount::valuespercentage
- collectd::plugin::fhcount::interval

collectd::plugin::filecount

- collectd::plugin::filecount::directories
- collectd::plugin::filecount::interval

collectd::plugin::fscache

- None

collectd-hddtemp

- collectd::plugin::hddtemp::host
- collectd::plugin::hddtemp::port
- collectd::plugin::hddtemp::interval

collectd::plugin::hugepages

Use the hugepages plugin to collect hugepages information. This plugin is enabled by default.

Table 7.13. hugepages parameters

Parameter	Type
Defaults	report_per_node_hp
Boolean	true
report_root_hp	Boolean
true	values_pages

Parameter	Type
Boolean	true
values_bytes	Boolean
false	values_percentage
Boolean	false

Example configuration

```
parameter_defaults:
   ExtraConfig:
      collectd::plugin::hugepages::values_percentage: true
```

Additional resources

- For more information about configuring the **hugepages** plugin, see hugepages.

collectd::plugin::intel_rdt

collectd::plugin::interface

Use the **interface** plugin to measure interface traffic in octets, packets per second, and error rate per second. This plugin is enabled by default.

Table 7.14. interface parameters

Parameter	Type
Default	interfaces
Array	[]
ignoreselected	Boolean
false	reportinactive
Boolean	true

Example configuration

```
parameter_defaults:
   ExtraConfig:
      collectd::plugin::interface::interfaces:
        - lo
      collectd::plugin::interface::ignoreselected: true
```

Additional resources

- For more information about configuring the **interfaces** plugin, see interfaces.

collectd::plugin::ipc

- None

collectd::plugin::ipmi

- collectd::plugin::ipmi::ignore_selected
- collectd::plugin::ipmi::notify_sensor_add
- collectd::plugin::ipmi::notify_sensor_remove
- collectd::plugin::ipmi::notify_sensor_not_present
- collectd::plugin::ipmi::sensors
- collectd::plugin::ipmi::interval

collectd::plugin::iptables

collectd::plugin::irq

- collectd::plugin::irq::irqs
- collectd::plugin::irq::ignoreselected
- collectd::plugin::irq::interval

collectd::plugin::load

Use the **load** plugin to collect the system load and to get overview on system use. This plugin is enabled by default.

Table 7.15. plugin parameters

Parameter	Type
report_relative	Boolean

Example configuration

```
parameter_defaults:
  ExtraConfig:
    collectd::plugin::load::report_relative: false
```

Additional resources

- For more information about configuring the **load** plugin, see load.

collectd::plugin::logfile

- collectd::plugin::logfile::log_level
- collectd::plugin::logfile::log_file

- collectd::plugin::logfile::log_timestamp
- collectd::plugin::logfile::print_severity
- collectd::plugin::logfile::interval

collectd::plugin::log_logstash

collectd::plugin::madwifi

collectd::plugin::match_empty_counter

collectd::plugin::match_hashed

collectd::plugin::match_regex

collectd::plugin::match_timediff

collectd::plugin::match_value

collectd::plugin::mbmon

collectd::plugin::mcelog

Use the **mcelog** plugin to send notifications and statistics relevant to Machine Check Exceptions when they occur. Configure **mcelog** to run in daemon mode and ensure that logging capabilities are enabled.

Table 7.16. mcelog parameters

Parameter	Type
Mcelogfile	String
Memory	Hash { mcelogclientsocket[string], persistentnotification[boolean] }

Example configuration

```
parameter_defaults:
   CollectdExtraPlugins: mcelog
   CollectdEnableMcelog: true
```

Additional resources

- For more information about configuring the **mcelog** plugin, see mcelog.

collectd::plugin::md

collectd::plugin::memcachec

collectd::plugin::memcached

- collectd::plugin::memcached::instances
- collectd::plugin::memcached::interval

collectd::plugin::memory

The **memory** plugin provides information about the memory of the system. This plugin is enabled by default.

Table 7.17. memory parameters

Parameter	Type
valuesabsolute	Boolean
valuespercentage	Boolean

Example configuration

```
parameter_defaults:
  ExtraConfig:
    collectd::plugin::memory::valuesabsolute: true
    collectd::plugin::memory::valuespercentage: false
```

Additional resources

- For more information about configuring the **memory** plugin, see memory.

collectd::plugin::multimeter
collectd::plugin::mysql

- collectd::plugin::mysql::interval

collectd::plugin::netlink

- collectd::plugin::netlink::interfaces
- collectd::plugin::netlink::verboseinterfaces
- collectd::plugin::netlink::qdiscs
- collectd::plugin::netlink::classes
- collectd::plugin::netlink::filters
- collectd::plugin::netlink::ignoreselected
- collectd::plugin::netlink::interval

collectd::plugin::network

- collectd::plugin::network::timetolive
- collectd::plugin::network::maxpacketsize
- collectd::plugin::network::forward
- collectd::plugin::network::reportstats
- collectd::plugin::network::listeners
- collectd::plugin::network::servers
- collectd::plugin::network::interval

collectd::plugin::nfs

- collectd::plugin::nfs::interval

collectd::plugin::notify_nagios
collectd::plugin::ntpd

- collectd::plugin::ntpd::host
- collectd::plugin::ntpd::port
- collectd::plugin::ntpd::reverselookups
- collectd::plugin::ntpd::includeunitid
- collectd::plugin::ntpd::interval

collectd::plugin::numa

- None

collectd::plugin::olsrd
collectd::plugin::openldap
collectd::plugin::openvpn

- collectd::plugin::openvpn::statusfile
- collectd::plugin::openvpn::improvednamingschema
- collectd::plugin::openvpn::collectcompression
- collectd::plugin::openvpn::collectindividualusers
- collectd::plugin::openvpn::collectusercount
- collectd::plugin::openvpn::interval

collectd::plugin::ovs_stats

Use the **ovs_stats** plugin to collect statistics of OVS-connected interfaces. The **ovs_stats** plugin uses the OVSDB management protocol (RFC7047) monitor mechanism to get statistics from OVSDB.

Table 7.18. ovs_stats parameters

Parameter	Type
address	String
bridges	List
port	Integer
socket	String

Example configuration

The following example shows how to enable the **ovs_stats** plugin. If you deploy your overcloud with OVS, you do not need to enable the **ovs_stats** plugin.

```
parameter_defaults:
    CollectdExtraPlugins:
      - ovs_stats
    ExtraConfig:
      collectd::plugin::ovs_stats::socket: '/run/openvswitch/db.sock'
```

Additional resources

- For more information about configuring the **ovs_stats** plugin, see ovs_stats.

collectd::plugin::pcie_errors

Use the **pcie_errors** plugin to poll PCI config space for baseline and Advanced Error Reporting (AER) errors, and to parse syslog for AER events. Errors are reported through notifications.

Table 7.19. pcie_errors parameters

Parameter	Type
source	Enum (sysfs, proc)
access	String
reportmasked	Boolean
persistent_notifications	Boolean

Example configuration

```
parameter_defaults:
    CollectdExtraPlugins:
      - pcie_errors
```

Additional resources

- For more information about configuring the **pcie_errors** plugin, see pcie_errors.

collectd::plugin::ping

- collectd::plugin::ping::hosts
- collectd::plugin::ping::timeout
- collectd::plugin::ping::ttl
- collectd::plugin::ping::source_address
- collectd::plugin::ping::device
- collectd::plugin::ping::max_missed

- collectd::plugin::ping::size
- collectd::plugin::ping::interval

collectd::plugin::powerdns

- collectd::plugin::powerdns::interval
- collectd::plugin::powerdns::servers
- collectd::plugin::powerdns::recursors
- collectd::plugin::powerdns::local_socket
- collectd::plugin::powerdns::interval

collectd::plugin::processes

- collectd::plugin::processes::processes
- collectd::plugin::processes::process_matches
- collectd::plugin::processes::collect_context_switch
- collectd::plugin::processes::collect_file_descriptor
- collectd::plugin::processes::collect_memory_maps
- collectd::plugin::powerdns::interval

collectd::plugin::processes

The **processes** plugin provides information about processes of the system. This plugin is enabled by default.

Table 7.20. plugin parameters

Parameter	Type
processes	Array
process_matches	Array
collect_context_switch	Boolean
collect_file_descriptor	Boolean
collect_memory_maps	Boolean

Additional resources

- For more information about configuring the **processes** plugin, see processes.

collectd::plugin::protocols

- collectd::plugin::protocols::ignoreselected

- collectd::plugin::protocols::values

collectd::plugin::python

collectd::plugin::sensors

collectd::plugin::serial

collectd::plugin::smart

- collectd::plugin::smart::disks
- collectd::plugin::smart::ignoreselected
- collectd::plugin::smart::interval

collectd::plugin::snmp

collectd::plugin::snmp_agent

Use the **snmp_agent** plugin as an SNMP subagent to map collectd metrics to relevant OIDs. The snmp agent also requires a running snmpd service.

Example configuration:

```
parameter_defaults:
  CollectdExtraPlugins:
    snmp_agent
resource_registry:
  OS::TripleO::Services::Snmp: /usr/share/openstack-tripleo-heat-
templates/deployment/snmp/snmp-baremetal-puppet.yaml
```

Additional resources:

For more information about how to configure **snmp_agent**, see snmp_agent.

collectd::plugin::statsd

- collectd::plugin::statsd::host
- collectd::plugin::statsd::port
- collectd::plugin::statsd::deletecounters
- collectd::plugin::statsd::deletetimers
- collectd::plugin::statsd::deletegauges
- collectd::plugin::statsd::deletesets
- collectd::plugin::statsd::countersum
- collectd::plugin::statsd::timerpercentile
- collectd::plugin::statsd::timerlower
- collectd::plugin::statsd::timerupper
- collectd::plugin::statsd::timersum

- collectd::plugin::statsd::timercount
- collectd::plugin::statsd::interval

collectd::plugin::swap

- collectd::plugin::swap::reportbydevice
- collectd::plugin::swap::reportbytes
- collectd::plugin::swap::valuesabsolute
- collectd::plugin::swap::valuespercentage
- collectd::plugin::swap::reportio
- collectd::plugin::swap::interval

collectd::plugin::sysevent

collectd::plugin::syslog

- collectd::plugin::syslog::log_level
- collectd::plugin::syslog::notify_level
- collectd::plugin::syslog::interval

collectd::plugin::table

- collectd::plugin::table::tables
- collectd::plugin::table::interval

collectd::plugin::tail

- collectd::plugin::tail::files
- collectd::plugin::tail::interval

collectd::plugin::tail_csv

- collectd::plugin::tail_csv::metrics
- collectd::plugin::tail_csv::files

collectd::plugin::target_notification

collectd::plugin::target_replace

collectd::plugin::target_scale

collectd::plugin::target_set

collectd::plugin::target_v5upgrade

collectd::plugin::tcpconns

- collectd::plugin::tcpconns::localports
- collectd::plugin::tcpconns::remoteports
- collectd::plugin::tcpconns::listening

- collectd::plugin::tcpconns::allportssummary
- collectd::plugin::tcpconns::interval

collectd::plugin::ted

collectd::plugin::thermal

- collectd::plugin::thermal::devices
- collectd::plugin::thermal::ignoreselected
- collectd::plugin::thermal::interval

collectd::plugin::threshold

- collectd::plugin::threshold::types
- collectd::plugin::threshold::plugins
- collectd::plugin::threshold::hosts
- collectd::plugin::threshold::interval

collectd::plugin::turbostat

- collectd::plugin::turbostat::core_c_states
- collectd::plugin::turbostat::package_c_states
- collectd::plugin::turbostat::system_management_interrupt
- collectd::plugin::turbostat::digital_temperature_sensor
- collectd::plugin::turbostat::tcc_activation_temp
- collectd::plugin::turbostat::running_average_power_limit
- collectd::plugin::turbostat::logical_core_names

collectd::plugin::unixsock

collectd::plugin::uptime

- collectd::plugin::uptime::interval

collectd::plugin::users

- collectd::plugin::users::interval

collectd::plugin::uuid

- collectd::plugin::uuid::uuid_file
- collectd::plugin::uuid::interval

collectd::plugin::virt

Use the **virt** plugin to collect CPU, disk, network load, and other metrics through the **libvirt** API for virtual machines on the host.

Table 7.21. virt parameters

Parameter	Type
connection	String
refresh_interval	Hash
domain	String
block_device	String
interface_device	String
ignore_selected	Boolean
plugin_instance_format	String
hostname_format	String
interface_format	String
extra_stats	String

Example configuration

```
ExtraConfig:
  collectd::plugin::virt::plugin_instance_format: name
```

Additional resources

For more information about configuring the **virt** plugin, see virt.

collectd::plugin::vmem

- collectd::plugin::vmem::verbose
- collectd::plugin::vmem::interval

collectd::plugin::vserver
collectd::plugin::wireless
collectd::plugin::write_graphite

- collectd::plugin::write_graphite::carbons
- collectd::plugin::write_graphite::carbon_defaults
- collectd::plugin::write_graphite::globals

collectd::plugin::write_http

Use the **write_http** output plugin to submit values to an HTTP server by using POST requests and encoding metrics with JSON, or by using the **PUTVAL** command.

Table 7.22. write_http parameters

Parameter	Type
ensure	Enum[*present*, *absent*]
nodes	Hash[String, Hash[String, Scalar]]
urls	Hash[String, Hash[String, Scalar]]
manage_package	Boolean

Example configuration

```
parameter_defaults:
    CollectdExtraPlugins:
      - write_http
    ExtraConfig:
        collectd::plugin::write_http::nodes:
            collectd:
                url: “http://collectd.tld.org/collectd”
                metrics: true
                header: “X-Custom-Header: custom_value"
```

Additional resources

- For more information about configuring the **write_http** plugin, see write_http.

collectd::plugin::write_kafka

Use the **write_kafka** plugin to send values to a Kafka topic. Configure the **write_kafka** plugin with one or more topic blocks. For each topic block, you must specify a unique name and one Kafka producer. You can use the following per-topic parameters inside the topic block:

Table 7.23. write_kafka parameters

Parameter	Type
kafka_hosts	Array[String]
kafka_port	Integer
topics	Hash
properties	Hash
meta	Hash

Example configuration:

```
parameter_defaults:
    CollectdExtraPlugins:
      - write_kafka
    ExtraConfig:
      collectd::plugin::write_kafka::kafka_hosts:
        - nodeA
        - nodeB
      collectd::plugin::write_kafka::topics:
        some_events:
          format: JSON
```

Additional resources:

For more information about how to configure the **write_kafka** plugin, see write_kafka.

collectd::plugin::write_log

- collectd::plugin::write_log::format

collectd::plugin::zfs_arc

- None