OpenShift Container Platform 4.19

Red Hat build of OpenTelemetry

Configuring and using the Red Hat build of OpenTelemetry in OpenShift Container Platform

Last Updated: 2025-10-01

OpenShift Container Platform 4.19 Red Hat build of OpenTelemetry

Configuring and using the Red Hat build of OpenTelemetry in OpenShift Container Platform

Legal Notice

Abstract

Use the Red Hat build of the open source OpenTelemetry project to collect unified, standardized, and vendor-neutral telemetry data for cloud-native software in OpenShift Container Platform.

Table of Contents

CHAPTER 1. RELEASE NOTES FOR THE RED HAT BUILD OF OPENTELEMETRY 3.7

1.1. ABOUT THIS RELEASE

Red Hat build of OpenTelemetry 3.7 is provided through the Red Hat build of OpenTelemetry Operator 0.135.0 and based on the open source OpenTelemetry release 0.135.0.

NOTE

Some linked Jira tickets are accessible only with Red Hat credentials.

NOTE

Only supported features are documented. Undocumented features are currently unsupported. If you need assistance with a feature, contact Red Hat's support.

1.2. NEW FEATURES AND ENHANCEMENTS

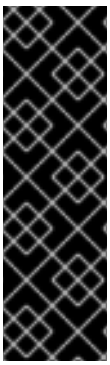
Network policy to restrict API access

With this update, the Red Hat build of OpenTelemetry Operator creates a network policy for itself and the OpenTelemetry Collector to restrict access to the used APIs.

Native sidecars

With this update, the Red Hat build of OpenTelemetry Operator uses native sidecars on OpenShift Container Platform 4.16 or later.

1.3. TECHNOLOGY PREVIEW FEATURES



IMPORTANT

Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

Probabilistic Sampling Processor (Technology Preview)

This release introduces the Probabilistic Sampling Processor as a Technology Preview feature for the Red Hat build of OpenTelemetry Collector. The Probabilistic Sampling Processor samples a specified percentage of trace spans or log records statelessly and per request. You can use the Probabilistic Sampling Processor if you handle high volumes of telemetry data and seek to reduce costs by reducing processed data volumes.

1.4. DEPRECATED FEATURES

The OpenCensus Receiver is deprecated

The OpenCensus Receiver, which provided backward compatibility with the OpenCensus format, is deprecated and might be removed in a future release.

The Collector's service metrics telemetry address is deprecated

The **metrics.address** field in the **OpenTelemetryCollector** custom resource (CR) is deprecated and might be removed in a future release. As an alternative, use the **metrics.readers** field instead.

Example of using the **readers** field:

```
# ...
  config:
    service:
      telemetry:
        metrics:
          readers:
          - pull:
              exporter:
                prometheus:
                  host: 0.0.0.0
                  port: 8888
# ...
```

1.5. REMOVED FEATURES

The LokiStack Exporter is removed

The LokiStack Exporter, which exported data to a LokiStack instance, is removed and no longer supported. You can export data to a LokiStack instance by using the OTLP HTTP Exporter instead.

The Routing Processor is removed

The Routing Processor, which routed telemetry data to an exporter is removed and no longer supported. You can route telemetry data by using the Routing Connector instead.

1.6. GETTING SUPPORT

If you experience difficulty with a procedure described in this documentation, or with OpenShift Container Platform in general, visit the Red Hat Customer Portal .

From the Customer Portal, you can:

- Search or browse through the Red Hat Knowledgebase of articles and solutions relating to Red Hat products.
- Submit a support case to Red Hat Support.
- Access other product documentation.

To identify issues with your cluster, you can use Insights in OpenShift Cluster Manager. Insights provides details about issues and, if available, information on how to solve a problem.

If you have a suggestion for improving this documentation or have found an error, submit a Jira issue for the most relevant documentation component. Please provide specific details, such as the section name and OpenShift Container Platform version.

CHAPTER 2. ABOUT RED HAT BUILD OF OPENTELEMETRY

2.1. RED HAT BUILD OF OPENTELEMETRY OVERVIEW

Red Hat build of OpenTelemetry is based on the open source OpenTelemetry project, which aims to provide unified, standardized, and vendor-neutral telemetry data collection for cloud-native software. Red Hat build of OpenTelemetry provides support for deploying and managing the OpenTelemetry Collector and simplifying the workload instrumentation.

The OpenTelemetry Collector can receive, process, and forward telemetry data in multiple formats, making it the ideal component for telemetry processing and interoperability between telemetry systems. The Collector provides a unified solution for collecting and processing metrics, traces, and logs.

The OpenTelemetry Collector provides several features including the following:

Data Collection and Processing Hub

It acts as a central component that gathers telemetry data like metrics and traces from various sources. This data can be created from instrumented applications and infrastructure.

Customizable telemetry data pipeline

The OpenTelemetry Collector is customizable and supports various processors, exporters, and receivers.

Auto-instrumentation features

Automatic instrumentation simplifies the process of adding observability to applications. Developers do not need to manually instrument their code for basic telemetry data.

Here are some of the use cases for the OpenTelemetry Collector:

Centralized data collection

In a microservices architecture, the Collector can be deployed to aggregate data from multiple services.

Data enrichment and processing

Before forwarding data to analysis tools, the Collector can enrich, filter, and process this data.

Multi-backend receiving and exporting

The Collector can receive and send data to multiple monitoring and analysis platforms simultaneously.

You can use Red Hat build of OpenTelemetry in combination with Red Hat OpenShift Distributed Tracing Platform.

CHAPTER 3. INSTALLING

Installing the Red Hat build of OpenTelemetry involves the following steps:

1. Installing the Red Hat build of OpenTelemetry Operator.
2. Creating a namespace for an OpenTelemetry Collector instance.
3. Creating an **OpenTelemetryCollector** custom resource to deploy the OpenTelemetry Collector instance.

3.1. INSTALLING THE RED HAT BUILD OF OPENTELEMETRY FROM THE WEB CONSOLE

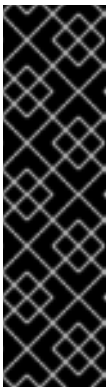
You can install the Red Hat build of OpenTelemetry from the **Administrator** view of the web console.

Prerequisites

- You are logged in to the web console as a cluster administrator with the **cluster-admin** role.
- For Red Hat OpenShift Dedicated, you must be logged in using an account with the **dedicated-admin** role.

Procedure

1. Install the Red Hat build of OpenTelemetry Operator:
   a. Go to **Operators → OperatorHub** and search for **Red Hat build of OpenTelemetry Operator**.
   b. Select the **Red Hat build of OpenTelemetry Operator** that is **provided by Red Hat → Install → Install → View Operator**.



IMPORTANT

This installs the Operator with the default presets:

- **Update channel → stable**
- **Installation mode → All namespaces on the cluster**
- **Installed Namespace → openshift-opentelemetry-operator**
- **Update approval → Automatic**

   c. In the **Details** tab of the installed Operator page, under **ClusterServiceVersion details**, verify that the installation **Status** is **Succeeded**.
2. Create a permitted project of your choice for the **OpenTelemetry Collector** instance that you will create in the next step by going to **Home → Projects → Create Project**. Project names beginning with the **openshift-** prefix are not permitted.
3. Create an **OpenTelemetry Collector** instance.

a. Go to **Operators → Installed Operators**.

b. Select **OpenTelemetry Collector → Create OpenTelemetry Collector → YAML view**.

c. In the **YAML view**, customize the **OpenTelemetryCollector** custom resource (CR):

Example OpenTelemetryCollector CR

```
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: <permitted_project_of_opentelemetry_collector_instance> 1
spec:
  mode: <deployment_mode> 2
  config:
    receivers: 3
      otlp:
        protocols:
          grpc:
          http:
      jaeger:
        protocols:
          grpc: {}
          thrift_binary: {}
          thrift_compact: {}
          thrift_http: {}
      zipkin: {}
    processors: 4
      batch: {}
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
    exporters: 5
      debug: {}
    service:
      pipelines:
        traces:
          receivers: [otlp,jaeger,zipkin]
          processors: [memory_limiter,batch]
          exporters: [debug]
```

1. The project that you have chosen for the **OpenTelemetryCollector** deployment. Project names beginning with the **openshift-** prefix are not permitted.
2. The deployment mode with the following supported values: the default **deployment**, **daemonset**, **statefulset**, or **sidecar**. For details, see *Deployment Modes*.
3. For details, see *Receivers*.
4. For details, see *Processors*.
5. For details, see *Exporters*.

d. Select **Create**.

Verification

1. Use the **Project:** dropdown list to select the project of the **OpenTelemetry Collector** instance.
2. Go to **Operators → Installed Operators** to verify that the **Status** of the **OpenTelemetry Collector** instance is **Condition: Ready**.
3. Go to **Workloads → Pods** to verify that all the component pods of the **OpenTelemetry Collector** instance are running.

3.2. INSTALLING THE RED HAT BUILD OF OPENTELEMETRY BY USING THE CLI

You can install the Red Hat build of OpenTelemetry from the command line.

Prerequisites

- An active OpenShift CLI (**oc**) session by a cluster administrator with the **cluster-admin** role.

  TIP

  - Ensure that your OpenShift CLI (**oc**) version is up to date and matches your OpenShift Container Platform version.
  - Run **oc login**:

    ```
    $ oc login --username=<your_username>
    ```

Procedure

1. Install the Red Hat build of OpenTelemetry Operator:
   a. Create a project for the Red Hat build of OpenTelemetry Operator by running the following command:

   ```
   $ oc apply -f - << EOF
   apiVersion: project.openshift.io/v1
   kind: Project
   metadata:
     labels:
       kubernetes.io/metadata.name: openshift-opentelemetry-operator
       openshift.io/cluster-monitoring: "true"
     name: openshift-opentelemetry-operator
   EOF
   ```

   b. Create an Operator group by running the following command:

   ```
   $ oc apply -f - << EOF
   apiVersion: operators.coreos.com/v1
   kind: OperatorGroup
   metadata:
   ```

```
  name: openshift-opentelemetry-operator
  namespace: openshift-opentelemetry-operator
spec:
  upgradeStrategy: Default
EOF
```

c. Create a subscription by running the following command:

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

d. Check the Operator status by running the following command:

```
$ oc get csv -n openshift-opentelemetry-operator
```

2. Create a permitted project of your choice for the OpenTelemetry Collector instance that you will create in a subsequent step:

- To create a permitted project without metadata, run the following command:

```
$ oc new-project <permitted_project_of_opentelemetry_collector_instance> 1
```

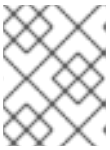
1 Project names beginning with the **openshift-** prefix are not permitted.

- To create a permitted project with metadata, run the following command:

```
$ oc apply -f - << EOF
apiVersion: project.openshift.io/v1
kind: Project
metadata:
  name: <permitted_project_of_opentelemetry_collector_instance> 1
EOF
```

1 Project names beginning with the **openshift-** prefix are not permitted.

3. Create an OpenTelemetry Collector instance in the project that you created for it.



NOTE

You can create multiple OpenTelemetry Collector instances in separate projects on the same cluster.

a. Customize the **OpenTelemetryCollector** custom resource (CR):

Example OpenTelemetryCollector CR

```
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: <permitted_project_of_opentelemetry_collector_instance> 1
spec:
  mode: <deployment_mode> 2
  config:
    receivers: 3
      otlp:
        protocols:
          grpc:
          http:
      jaeger:
        protocols:
          grpc: {}
          thrift_binary: {}
          thrift_compact: {}
          thrift_http: {}
      zipkin: {}
    processors: 4
      batch: {}
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
    exporters: 5
      debug: {}
    service:
      pipelines:
        traces:
          receivers: [otlp,jaeger,zipkin]
          processors: [memory_limiter,batch]
          exporters: [debug]
```

1. The project that you have chosen for the **OpenTelemetryCollector** deployment. Project names beginning with the **openshift-** prefix are not permitted.
2. The deployment mode with the following supported values: the default **deployment**, **daemonset**, **statefulset**, or **sidecar**. For details, see *Deployment Modes*.
3. For details, see *Receivers*.
4. For details, see *Processors*.
5. For details, see *Exporters*.

b. Apply the customized CR by running the following command:

```
$ oc apply -f - << EOF
<OpenTelemetryCollector_custom_resource>
EOF
```

Verification

1. Verify that the **status.phase** of the OpenTelemetry Collector pod is **Running** and the **conditions** are **type: Ready** by running the following command:

```
$ oc get pod -l app.kubernetes.io/managed-by=opentelemetry-
operator,app.kubernetes.io/instance=<namespace>.<instance_name> -o yaml
```

2. Get the OpenTelemetry Collector service by running the following command:

```
$ oc get service -l app.kubernetes.io/managed-by=opentelemetry-
operator,app.kubernetes.io/instance=<namespace>.<instance_name>
```

3.3. USING TAINTS AND TOLERATIONS

To schedule the OpenTelemetry pods on dedicated nodes, see How to deploy the different OpenTelemetry components on infra nodes using nodeSelector and tolerations in OpenShift 4

3.4. CREATING THE REQUIRED RBAC RESOURCES AUTOMATICALLY

Some Collector components require configuring the RBAC resources.

Procedure

- Add the following permissions to the **opentelemetry-operator-controller-manage** service account so that the Red Hat build of OpenTelemetry Operator can create them automatically:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: generate-processors-rbac
rules:
- apiGroups:
  - rbac.authorization.k8s.io
  resources:
  - clusterrolebindings
  - clusterroles
  verbs:
  - create
  - delete
  - get
  - list
  - patch
  - update
  - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
```

```
  name: generate-processors-rbac
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: generate-processors-rbac
subjects:
- kind: ServiceAccount
  name: opentelemetry-operator-controller-manager
  namespace: openshift-opentelemetry-operator
```

3.5. ADDITIONAL RESOURCES

- Creating a cluster admin
- OperatorHub.io
- Accessing the web console
- Installing from OperatorHub using the web console
- Creating applications from installed Operators
- Getting started with the OpenShift CLI

CHAPTER 4. CONFIGURING THE COLLECTOR

4.1. CONFIGURING THE COLLECTOR

The Red Hat build of OpenTelemetry Operator uses a custom resource definition (CRD) file that defines the architecture and configuration settings to be used when creating and deploying the Red Hat build of OpenTelemetry resources. You can install the default configuration or modify the file.

4.1.1. Deployment modes

The **OpenTelemetryCollector** custom resource allows you to specify one of the following deployment modes for the OpenTelemetry Collector:

Deployment

The default.

StatefulSet

If you need to run stateful workloads, for example when using the Collector's File Storage Extension or Tail Sampling Processor, use the StatefulSet deployment mode.

DaemonSet

If you need to scrape telemetry data from every node, for example by using the Collector's Filelog Receiver to read container logs, use the DaemonSet deployment mode.

Sidecar

If you need access to log files inside a container, inject the Collector as a sidecar, and use the Collector's Filelog Receiver and a shared volume such as **emptyDir**.

If you need to configure an application to send telemetry data via **localhost**, inject the Collector as a sidecar, and set up the Collector to forward the telemetry data to an external service via an encrypted and authenticated connection. The Collector runs in the same pod as the application when injected as a sidecar.

NOTE

If you choose the sidecar deployment mode, then in addition to setting the **spec.mode: sidecar** field in the **OpenTelemetryCollector** custom resource CR, you must also set the **sidecar.opentelemetry.io/inject** annotation as a pod annotation or namespace annotation. If you set this annotation on both the pod and namespace, the pod annotation takes precedence if it is set to either **false** or the **OpenTelemetryCollector** CR name.

As a pod annotation, the **sidecar.opentelemetry.io/inject** annotation supports several values:

```
apiVersion: v1
kind: Pod
metadata:
  ...
  annotations:
    sidecar.opentelemetry.io/inject: "<supported_value>" 1
...
```

1 Supported values:

false
: Does not inject the Collector. This is the default if the annotation is missing.

true
: Injects the Collector with the configuration of the **OpenTelemetryCollector** CR in the same namespace.

<collector_name>
: Injects the Collector with the configuration of the **<collector_name>** **OpenTelemetryCollector** CR in the same namespace.

<namespace>/<collector_name>
: Injects the Collector with the configuration of the **<collector_name>** **OpenTelemetryCollector** CR in the **<namespace>** namespace.

4.1.2. OpenTelemetry Collector configuration options

The OpenTelemetry Collector consists of five types of components that access telemetry data:

- Receivers
- Processors
- Exporters
- Connectors
- Extensions

You can define multiple instances of components in a custom resource YAML file. When configured, these components must be enabled through pipelines defined in the **spec.config.service** section of the YAML file. As a best practice, only enable the components that you need.

Example of the OpenTelemetry Collector custom resource file

```
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
metadata:
  name: cluster-collector
  namespace: tracing-system
spec:
  mode: deployment
  observability:
    metrics:
      enableMetrics: true
  config:
    receivers:
      otlp:
        protocols:
          grpc: {}
          http: {}
    processors: {}
    exporters:
      otlp:
        endpoint: otel-collector-headless.tracing-system.svc:4317
        tls:
          ca_file: "/var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt"
      prometheus:
        endpoint: 0.0.0.0:8889
        resource_to_telemetry_conversion:
          enabled: true # by default resource attributes are dropped
    service: 1
      pipelines:
        traces:
          receivers: [otlp]
          processors: []
          exporters: [otlp]
        metrics:
          receivers: [otlp]
          processors: []
          exporters: [prometheus]
```

1 If a component is configured but not defined in the **service** section, the component is not enabled.

Table 4.1. Parameters used by the Operator to define the OpenTelemetry Collector

Parameter	Description	Values	Default

Parameter	Description	Values	Default
receivers:	A receiver is how data gets into the Collector. By default, no receivers are configured. There must be at least one enabled receiver for a configuration to be considered valid. Receivers are enabled by being added to a pipeline.	**otlp**, **jaeger**, **prometheus**, **zipkin**, **kafka**, **opencensus**	None
processors:	Processors run through the received data before it is exported. By default, no processors are enabled.	**batch**, **memory_limiter**, **resourcedetection**, **attributes**, **span**, **k8sattributes**, **filter**, **routing**	None
exporters:	An exporter sends data to one or more back ends or destinations. By default, no exporters are configured. There must be at least one enabled exporter for a configuration to be considered valid. Exporters are enabled by being added to a pipeline. Exporters might be used with their default settings, but many require configuration to specify at least the destination and security settings.	**otlp**, **otlphttp**, **debug**, **prometheus**, **kafka**	None
connectors:	Connectors join pairs of pipelines by consuming data as end-of-pipeline exporters and emitting data as start-of-pipeline receivers. Connectors can be used to summarize, replicate, or route consumed data.	**spanmetrics**	None

Parameter	Description	Values	Default
extensions:	Optional components for tasks that do not involve processing telemetry data.	**bearertokenauth**, **oauth2client**, **jaegerremotesampling**, **pprof**, **health_check**, **memory_ballast**, **zpages**	None
service: pipelines:	Components are enabled by adding them to a pipeline under **services.pipeline**.		
service: pipelines: traces: receivers:	You enable receivers for tracing by adding them under **service.pipelines.traces**.		None
service: pipelines: traces: processors:	You enable processors for tracing by adding them under **service.pipelines.traces**.		None
service: pipelines: traces: exporters:	You enable exporters for tracing by adding them under **service.pipelines.traces**.		None
service: pipelines: metrics: receivers:	You enable receivers for metrics by adding them under **service.pipelines.metrics**.		None
service: pipelines: metrics: processors:	You enable processors for metircs by adding them under **service.pipelines.metrics**.		None

Parameter	Description	Values	Default

`service:`
`  pipelines:`
`  metrics:`
`  exporters:`
	You enable exporters for metrics by adding them under **service.pipelines.metrics**.		None

4.1.3. Creating the required RBAC resources automatically

Some Collector components require configuring the RBAC resources.

Procedure

- Add the following permissions to the **opentelemetry-operator-controller-manage** service account so that the Red Hat build of OpenTelemetry Operator can create them automatically:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: generate-processors-rbac
rules:
- apiGroups:
  - rbac.authorization.k8s.io
  resources:
  - clusterrolebindings
  - clusterroles
  verbs:
  - create
  - delete
  - get
  - list
  - patch
  - update
  - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: generate-processors-rbac
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: generate-processors-rbac
subjects:
- kind: ServiceAccount
  name: opentelemetry-operator-controller-manager
  namespace: openshift-opentelemetry-operator
```

4.2. RECEIVERS

Receivers get data into the Collector. A receiver can be push or pull based. Generally, a receiver accepts

data in a specified format, translates it into the internal format, and passes it to processors and exporters defined in the applicable pipelines. By default, no receivers are configured. One or more receivers must be configured. Receivers may support one or more data sources.

Currently, the following General Availability and Technology Preview receivers are available for the Red Hat build of OpenTelemetry:

- OTLP Receiver
- Jaeger Receiver
- Host Metrics Receiver
- Kubernetes Objects Receiver
- Kubelet Stats Receiver
- Prometheus Receiver
- OTLP JSON File Receiver
- Zipkin Receiver
- Kafka Receiver
- Kubernetes Cluster Receiver
- OpenCensus Receiver
- Filelog Receiver
- Journald Receiver
- Kubernetes Events Receiver

4.2.1. OTLP Receiver

The OTLP Receiver ingests traces, metrics, and logs by using the OpenTelemetry Protocol (OTLP). The OTLP Receiver ingests traces and metrics using the OpenTelemetry protocol (OTLP).

OpenTelemetry Collector custom resource with an enabled OTLP Receiver

```
# ...
  config:
    receivers:
      otlp:
        protocols:
          grpc:
            endpoint: 0.0.0.0:4317 (1)
            tls: (2)
              ca_file: ca.pem
              cert_file: cert.pem
              key_file: key.pem
              client_ca_file: client.pem (3)
              reload_interval: 1h (4)
          http:
```

```
        endpoint: 0.0.0.0:4318 5
        tls: {} 6

    service:
      pipelines:
        traces:
          receivers: [otlp]
        metrics:
          receivers: [otlp]
# ...
```

1. The OTLP gRPC endpoint. If omitted, the default **0.0.0.0:4317** is used.
2. The server-side TLS configuration. Defines paths to TLS certificates. If omitted, the TLS is disabled.
3. The path to the TLS certificate at which the server verifies a client certificate. This sets the value of **ClientCAs** and **ClientAuth** to **RequireAndVerifyClientCert** in the **TLSConfig**. For more information, see the **Config** of the Golang TLS package.
4. Specifies the time interval at which the certificate is reloaded. If the value is not set, the certificate is never reloaded. The **reload_interval** field accepts a string containing valid units of time such as **ns**, **us** (or **µs**), **ms**, **s**, **m**, **h**.
5. The OTLP HTTP endpoint. The default value is **0.0.0.0:4318**.
6. The server-side TLS configuration. For more information, see the **grpc** protocol configuration section.

4.2.2. Jaeger Receiver

The Jaeger Receiver ingests traces in the Jaeger formats.

OpenTelemetry Collector custom resource with an enabled Jaeger Receiver

```
# ...
  config:
    receivers:
      jaeger:
        protocols:
          grpc:
            endpoint: 0.0.0.0:14250 1
          thrift_http:
            endpoint: 0.0.0.0:14268 2
          thrift_compact:
            endpoint: 0.0.0.0:6831 3
          thrift_binary:
            endpoint: 0.0.0.0:6832 4
          tls: {} 5

    service:
      pipelines:
```

```
      traces:
        receivers: [jaeger]
# ...
```

1. The Jaeger gRPC endpoint. If omitted, the default **0.0.0.0:14250** is used.
2. The Jaeger Thrift HTTP endpoint. If omitted, the default **0.0.0.0:14268** is used.
3. The Jaeger Thrift Compact endpoint. If omitted, the default **0.0.0.0:6831** is used.
4. The Jaeger Thrift Binary endpoint. If omitted, the default **0.0.0.0:6832** is used.
5. The server-side TLS configuration. See the OTLP Receiver configuration section for more details.

4.2.3. Host Metrics Receiver

The Host Metrics Receiver ingests metrics in the OTLP format.

OpenTelemetry Collector custom resource with an enabled Host Metrics Receiver

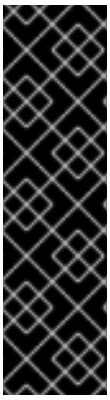
```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: otel-hostfs-daemonset
  namespace: <namespace>
# ...
---
apiVersion: security.openshift.io/v1
kind: SecurityContextConstraints
allowHostDirVolumePlugin: true
allowHostIPC: false
allowHostNetwork: false
allowHostPID: true
allowHostPorts: false
allowPrivilegeEscalation: true
allowPrivilegedContainer: true
allowedCapabilities: null
defaultAddCapabilities:
- SYS_ADMIN
fsGroup:
  type: RunAsAny
groups: []
metadata:
  name: otel-hostmetrics
readOnlyRootFilesystem: true
runAsUser:
  type: RunAsAny
seLinuxContext:
  type: RunAsAny
supplementalGroups:
  type: RunAsAny
users:
- system:serviceaccount:<namespace>:otel-hostfs-daemonset
volumes:
- configMap
```

```
- emptyDir
- hostPath
- projected
# ...
---
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: <namespace>
spec:
  serviceAccount: otel-hostfs-daemonset
  mode: daemonset
  volumeMounts:
    - mountPath: /hostfs
      name: host
      readOnly: true
  volumes:
    - hostPath:
        path: /
      name: host
  config:
    receivers:
      hostmetrics:
        collection_interval: 10s (1)
        initial_delay: 1s (2)
        root_path: / (3)
        scrapers: (4)
          cpu: {}
          memory: {}
          disk: {}
    service:
      pipelines:
        metrics:
          receivers: [hostmetrics]
# ...
```

1. Sets the time interval for host metrics collection. If omitted, the default value is **1m**.
2. Sets the initial time delay for host metrics collection. If omitted, the default value is **1s**.
3. Configures the **root_path** so that the Host Metrics Receiver knows where the root filesystem is. If running multiple instances of the Host Metrics Receiver, set the same **root_path** value for each instance.
4. Lists the enabled host metrics scrapers. Available scrapers are **cpu**, **disk**, **load**, **filesystem**, **memory**, **network**, **paging**, **processes**, and **process**.

4.2.4. Kubernetes Objects Receiver

The Kubernetes Objects Receiver pulls or watches objects to be collected from the Kubernetes API server. This receiver watches primarily Kubernetes events, but it can collect any type of Kubernetes objects. This receiver gathers telemetry for the cluster as a whole, so only one instance of this receiver suffices for collecting all the data.



IMPORTANT

The Kubernetes Objects Receiver is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with an enabled Kubernetes Objects Receiver

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: otel-k8sobj
  namespace: <namespace>
# ...
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otel-k8sobj
  namespace: <namespace>
rules:
- apiGroups:
  - ""
  resources:
  - events
  - pods
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - "events.k8s.io"
  resources:
  - events
  verbs:
  - watch
  - list
# ...
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-k8sobj
subjects:
  - kind: ServiceAccount
    name: otel-k8sobj
    namespace: <namespace>
roleRef:
  kind: ClusterRole
  name: otel-k8sobj
```

```
  apiGroup: rbac.authorization.k8s.io
# ...
---
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
metadata:
  name: otel-k8s-obj
  namespace: <namespace>
spec:
  serviceAccount: otel-k8sobj
  mode: deployment
  config:
    receivers:
      k8sobjects:
        auth_type: serviceAccount
        objects:
          - name: pods (1)
            mode: pull (2)
            interval: 30s (3)
            label_selector: (4)
            field_selector: (5)
            namespaces: [<namespace>,...] (6)
          - name: events
            mode: watch
    exporters:
      debug:
    service:
      pipelines:
        logs:
          receivers: [k8sobjects]
          exporters: [debug]
# ...
```

1. The Resource name that this receiver observes: for example, **pods**, **deployments**, or **events**.
2. The observation mode that this receiver uses: **pull** or **watch**.
3. Only applicable to the pull mode. The request interval for pulling an object. If omitted, the default value is **1h**.
4. The label selector to define targets.
5. The field selector to filter targets.
6. The list of namespaces to collect events from. If omitted, the default value is **all**.

4.2.5. Kubelet Stats Receiver

The Kubelet Stats Receiver extracts metrics related to nodes, pods, containers, and volumes from the kubelet's API server. These metrics are then channeled through the metrics-processing pipeline for additional analysis.

OpenTelemetry Collector custom resource with an enabled Kubelet Stats Receiver

```
# ...
  config:
    receivers:
      kubeletstats:
        collection_interval: 20s
        auth_type: "serviceAccount"
        endpoint: "https://${env:K8S_NODE_NAME}:10250"
        insecure_skip_verify: true
    service:
      pipelines:
        metrics:
          receivers: [kubeletstats]
  env:
    - name: K8S_NODE_NAME (1)
      valueFrom:
        fieldRef:
          fieldPath: spec.nodeName
# ...
```

(1) Sets the **K8S_NODE_NAME** to authenticate to the API.

The Kubelet Stats Receiver requires additional permissions for the service account used for running the OpenTelemetry Collector.

Permissions required by the service account

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otel-collector
rules:
  - apiGroups: ['']
    resources: ['nodes/stats']
    verbs: ['get', 'watch', 'list']
  - apiGroups: [""]
    resources: ["nodes/proxy"] (1)
    verbs: ["get"]
# ...
```

(1) The permissions required when using the **extra_metadata_labels** or **request_utilization** or **limit_utilization** metrics.

4.2.6. Prometheus Receiver

The Prometheus Receiver scrapes the metrics endpoints.

OpenTelemetry Collector custom resource with an enabled Prometheus Receiver

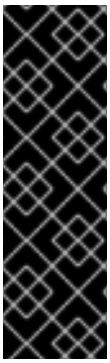
```
# ...
  config:
    receivers:
      prometheus:
        config:
```

```
          scrape_configs: (1)
            - job_name: 'my-app' (2)
              scrape_interval: 5s (3)
              static_configs:
                - targets: ['my-app.example.svc.cluster.local:8888'] (4)
    service:
      pipelines:
        metrics:
          receivers: [prometheus]
# ...
```

1. Scrapes configurations using the Prometheus format.
2. The Prometheus job name.
3. The Interval for scraping the metrics data. Accepts time units. The default value is **1m**.
4. The targets at which the metrics are exposed. This example scrapes the metrics from a **my-app** application in the **example** project.

4.2.7. OTLP JSON File Receiver

The OTLP JSON File Receiver extracts pipeline information from files containing data in the ProtoJSON format and conforming to the OpenTelemetry Protocol specification. The receiver watches a specified directory for changes such as created or modified files to process.



IMPORTANT

The OTLP JSON File Receiver is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with the enabled OTLP JSON File Receiver

```
# ...
  config:
    otlpjsonfile:
      include:
        - "/var/log/*.log" (1)
      exclude:
        - "/var/log/test.log" (2)
# ...
```

1. The list of file path glob patterns to watch.
2. The list of file path glob patterns to ignore.

4.2.8. Zipkin Receiver

The Zipkin Receiver ingests traces in the Zipkin v1 and v2 formats.

OpenTelemetry Collector custom resource with the enabled Zipkin Receiver

```
# ...
  config:
    receivers:
      zipkin:
        endpoint: 0.0.0.0:9411 1
        tls: {} 2
    service:
      pipelines:
        traces:
          receivers: [zipkin]
# ...
```

1. The Zipkin HTTP endpoint. If omitted, the default **0.0.0.0:9411** is used.
2. The server-side TLS configuration. See the OTLP Receiver configuration section for more details.

4.2.9. Kafka Receiver

The Kafka Receiver receives traces, metrics, and logs from Kafka in the OTLP format.

IMPORTANT

The Kafka Receiver is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with the enabled Kafka Receiver

```
# ...
  config:
    receivers:
      kafka:
        brokers: ["localhost:9092"] 1
        protocol_version: 2.0.0 2
        topic: otlp_spans 3
        auth:
          plain_text: 4
            username: example
            password: example
          tls: 5
            ca_file: ca.pem
```

```
        cert_file: cert.pem
        key_file: key.pem
        insecure: false (6)
        server_name_override: kafka.example.corp (7)
    service:
      pipelines:
        traces:
          receivers: [kafka]
# ...
```

1. The list of Kafka brokers. The default is **localhost:9092**.
2. The Kafka protocol version. For example, **2.0.0**. This is a required field.
3. The name of the Kafka topic to read from. The default is **otlp_spans**.
4. The plain text authentication configuration. If omitted, plain text authentication is disabled.
5. The client-side TLS configuration. Defines paths to the TLS certificates. If omitted, TLS authentication is disabled.
6. Disables verifying the server's certificate chain and host name. The default is **false**.
7. ServerName indicates the name of the server requested by the client to support virtual hosting.

4.2.10. Kubernetes Cluster Receiver

The Kubernetes Cluster Receiver gathers cluster metrics and entity events from the Kubernetes API server. It uses the Kubernetes API to receive information about updates. Authentication for this receiver is only supported through service accounts.

IMPORTANT

The Kubernetes Cluster Receiver is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with the enabled Kubernetes Cluster Receiver

```
# ...
  config:
    receivers:
      k8s_cluster:
        distribution: openshift
        collection_interval: 10s
    exporters:
      debug: {}
    service:
```

```
    pipelines:
      metrics:
        receivers: [k8s_cluster]
        exporters: [debug]
      logs/entity_events:
        receivers: [k8s_cluster]
        exporters: [debug]
# ...
```

This receiver requires a configured service account, RBAC rules for the cluster role, and the cluster role binding that binds the RBAC with the service account.

ServiceAccount object

```
apiVersion: v1
kind: ServiceAccount
metadata:
  labels:
    app: otelcontribcol
  name: otelcontribcol
# ...
```

RBAC rules for the ClusterRole object

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otelcontribcol
  labels:
    app: otelcontribcol
rules:
- apiGroups:
  - quota.openshift.io
  resources:
  - clusterresourcequotas
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - ""
  resources:
  - events
  - namespaces
  - namespaces/status
  - nodes
  - nodes/spec
  - pods
  - pods/status
  - replicationcontrollers
  - replicationcontrollers/status
  - resourcequotas
  - services
  verbs:
  - get
```

```
  - list
  - watch
- apiGroups:
  - apps
  resources:
  - daemonsets
  - deployments
  - replicasets
  - statefulsets
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - extensions
  resources:
  - daemonsets
  - deployments
  - replicasets
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - batch
  resources:
  - jobs
  - cronjobs
  verbs:
  - get
  - list
  - watch
- apiGroups:
    - autoscaling
  resources:
    - horizontalpodautoscalers
  verbs:
    - get
    - list
    - watch
# ...
```

ClusterRoleBinding object

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otelcontribcol
  labels:
    app: otelcontribcol
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: otelcontribcol
subjects:
- kind: ServiceAccount
```

```
  name: otelcontribcol
  namespace: default
# ...
```

4.2.11. OpenCensus Receiver

The OpenCensus Receiver provides backwards compatibility with the OpenCensus project for easier migration of instrumented codebases. It receives metrics and traces in the OpenCensus format via gRPC or HTTP and Json.

WARNING

The OpenCensus Receiver is deprecated and might be removed in a future release.

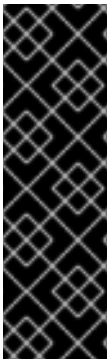
OpenTelemetry Collector custom resource with the enabled OpenCensus Receiver

```
# ...
  config:
    receivers:
      opencensus:
        endpoint: 0.0.0.0:9411 (1)
        tls: (2)
        cors_allowed_origins: (3)
          - https://*.<example>.com
    service:
      pipelines:
        traces:
          receivers: [opencensus]
# ...
```

1. The OpenCensus endpoint. If omitted, the default is **0.0.0.0:55678**.
2. The server-side TLS configuration. See the OTLP Receiver configuration section for more details.
3. You can also use the HTTP JSON endpoint to optionally configure CORS, which is enabled by specifying a list of allowed CORS origins in this field. Wildcards with * are accepted under the **cors_allowed_origins**. To match any origin, enter only *.

4.2.12. Filelog Receiver

The Filelog Receiver tails and parses logs from files.



IMPORTANT

The Filelog Receiver is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with the enabled Filelog Receiver that tails a text file

```
# ...
  config:
    receivers:
      filelog:
        include: [ /simple.log ] 1
        operators: 2
          - type: regex_parser
            regex: '^(?P<time>\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}) (?P<sev>[A-Z]*) (?P<msg>.*)$'
            timestamp:
              parse_from: attributes.time
              layout: '%Y-%m-%d %H:%M:%S'
            severity:
              parse_from: attributes.sev
# ...
```

1 A list of file glob patterns that match the file paths to be read.

2 An array of Operators. Each Operator performs a simple task such as parsing a timestamp or JSON. To process logs into a desired format, chain the Operators together.

4.2.13. Journald Receiver

The Journald Receiver parses **journald** events from the **systemd** journal and sends them as logs.

IMPORTANT

The Journald Receiver is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with the enabled Journald Receiver

```
apiVersion: v1
kind: Namespace
```

```
metadata:
  name: otel-journald
  labels:
    security.openshift.io/scc.podSecurityLabelSync: "false"
    pod-security.kubernetes.io/enforce: "privileged"
    pod-security.kubernetes.io/audit: "privileged"
    pod-security.kubernetes.io/warn: "privileged"
# ...
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: privileged-sa
  namespace: otel-journald
# ...
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-journald-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:openshift:scc:privileged
subjects:
- kind: ServiceAccount
  name: privileged-sa
  namespace: otel-journald
# ...
---
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
metadata:
  name: otel-journald-logs
  namespace: otel-journald
spec:
  mode: daemonset
  serviceAccount: privileged-sa
  securityContext:
    allowPrivilegeEscalation: false
    capabilities:
      drop:
      - CHOWN
      - DAC_OVERRIDE
      - FOWNER
      - FSETID
      - KILL
      - NET_BIND_SERVICE
      - SETGID
      - SETPCAP
      - SETUID
    readOnlyRootFilesystem: true
    seLinuxOptions:
      type: spc_t
    seccompProfile:
      type: RuntimeDefault
```

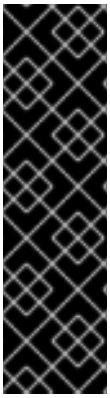
```
config:
  receivers:
    journald:
      files: /var/log/journal/*/*
      priority: info 1
      units: 2
        - kubelet
        - crio
        - init.scope
        - dnsmasq
      all: true 3
      retry_on_failure:
        enabled: true 4
        initial_interval: 1s 5
        max_interval: 30s 6
        max_elapsed_time: 5m 7
  processors:
  exporters:
    debug: {}
  service:
    pipelines:
      logs:
        receivers: [journald]
        exporters: [debug]
volumeMounts:
- name: journal-logs
  mountPath: /var/log/journal/
  readOnly: true
volumes:
- name: journal-logs
  hostPath:
    path: /var/log/journal
tolerations:
- key: node-role.kubernetes.io/master
  operator: Exists
  effect: NoSchedule
# ...
```

1. Filters output by message priorities or priority ranges. The default value is **info**.
2. Lists the units to read entries from. If empty, entries are read from all units.
3. Includes very long logs and logs with unprintable characters. The default value is **false**.
4. If set to **true**, the receiver pauses reading a file and attempts to resend the current batch of logs when encountering an error from downstream components. The default value is **false**.
5. The time interval to wait after the first failure before retrying. The default value is **1s**. The units are **ms**, **s**, **m**, **h**.
6. The upper bound for the retry backoff interval. When this value is reached, the time interval between consecutive retry attempts remains constant at this value. The default value is **30s**. The supported units are **ms**, **s**, **m**, **h**.
7. The maximum time interval, including retry attempts, for attempting to send a logs batch to a

downstream consumer. When this value is reached, the data are discarded. If the set value is **0**, retrying never stops. The default value is **5m**. The supported units are **ms**, **s**, **m**, **h**.

4.2.14. Kubernetes Events Receiver

The Kubernetes Events Receiver collects events from the Kubernetes API server. The collected events are converted into logs.



IMPORTANT

The Kubernetes Events Receiver is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenShift Container Platform permissions required for the Kubernetes Events Receiver

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otel-collector
  labels:
    app: otel-collector
rules:
- apiGroups:
  - ""
  resources:
  - events
  - namespaces
  - namespaces/status
  - nodes
  - nodes/spec
  - pods
  - pods/status
  - replicationcontrollers
  - replicationcontrollers/status
  - resourcequotas
  - services
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - apps
  resources:
  - daemonsets
  - deployments
  - replicasets
  - statefulsets
```

```
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - extensions
  resources:
  - daemonsets
  - deployments
  - replicasets
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - batch
  resources:
  - jobs
  - cronjobs
  verbs:
  - get
  - list
  - watch
- apiGroups:
    - autoscaling
  resources:
    - horizontalpodautoscalers
  verbs:
    - get
    - list
    - watch
# ...
```

OpenTelemetry Collector custom resource with the enabled Kubernetes Event Receiver

```
# ...
  serviceAccount: otel-collector 1
  config:
    receivers:
      k8s_events:
        namespaces: [project1, project2] 2
    service:
      pipelines:
        logs:
          receivers: [k8s_events]
# ...
```

1 The service account of the Collector that has the required ClusterRole **otel-collector** RBAC.

2 The list of namespaces to collect events from. The default value is empty, which means that all namespaces are collected.

4.2.15. Additional resources

- OpenTelemetry Protocol (OTLP) (OpenTelemetry Documentation)

4.3. PROCESSORS

Processors process the data between it is received and exported. Processors are optional. By default, no processors are enabled. Processors must be enabled for every data source. Not all processors support all data sources. Depending on the data source, multiple processors might be enabled. Note that the order of processors matters.

Currently, the following General Availability and Technology Preview processors are available for the Red Hat build of OpenTelemetry:

- Batch Processor
- Memory Limiter Processor
- Resource Detection Processor
- Attributes Processor
- Resource Processor
- Span Processor
- Kubernetes Attributes Processor
- Filter Processor
- Cumulative-to-Delta Processor
- Group-by-Attributes Processor
- Transform Processor
- Tail Sampling Processor

4.3.1. Batch Processor

The Batch Processor batches traces and metrics to reduce the number of outgoing connections needed to transfer the telemetry information.

Example of the OpenTelemetry Collector custom resource when using the Batch Processor

```
# ...
  config:
    processors:
      batch:
        timeout: 5s
        send_batch_max_size: 10000
    service:
      pipelines:
        traces:
          processors: [batch]
        metrics:
          processors: [batch]
# ...
```

Table 4.2. Parameters used by the Batch Processor

Parameter	Description	Default
timeout	Sends the batch after a specific time duration and irrespective of the batch size.	**200ms**
send_batch_size	Sends the batch of telemetry data after the specified number of spans or metrics.	**8192**
send_batch_max_size	The maximum allowable size of the batch. Must be equal or greater than the **send_batch_size**.	**0**
metadata_keys	When activated, a batcher instance is created for each unique set of values found in the **client.Metadata**.	**[]**
metadata_cardinality_limit	When the **metadata_keys** are populated, this configuration restricts the number of distinct metadata key-value combinations processed throughout the duration of the process.	**1000**

4.3.2. Memory Limiter Processor

The Memory Limiter Processor periodically checks the Collector's memory usage and pauses data processing when the soft memory limit is reached. This processor supports traces, metrics, and logs. The preceding component, which is typically a receiver, is expected to retry sending the same data and may apply a backpressure to the incoming data. When memory usage exceeds the hard limit, the Memory Limiter Processor forces garbage collection to run.

Example of the OpenTelemetry Collector custom resource when using the Memory Limiter Processor

```
# ...
  config:
    processors:
      memory_limiter:
        check_interval: 1s
        limit_mib: 4000
        spike_limit_mib: 800
    service:
      pipelines:
        traces:
          processors: [batch]
        metrics:
          processors: [batch]
```

```
# ...
```

Table 4.3. Parameters used by the Memory Limiter Processor

Parameter	Description	Default
check_interval	Time between memory usage measurements. The optimal value is **1s**. For spiky traffic patterns, you can decrease the **check_interval** or increase the **spike_limit_mib**.	**0s**
limit_mib	The hard limit, which is the maximum amount of memory in MiB allocated on the heap. Typically, the total memory usage of the OpenTelemetry Collector is about 50 MiB greater than this value.	**0**
spike_limit_mib	Spike limit, which is the maximum expected spike of memory usage in MiB. The optimal value is approximately 20% of **limit_mib**. To calculate the soft limit, subtract the **spike_limit_mib** from the **limit_mib**.	20% of **limit_mib**
limit_percentage	Same as the **limit_mib** but expressed as a percentage of the total available memory. The **limit_mib** setting takes precedence over this setting.	**0**
spike_limit_percentage	Same as the **spike_limit_mib** but expressed as a percentage of the total available memory. Intended to be used with the **limit_percentage** setting.	**0**

4.3.3. Resource Detection Processor

The Resource Detection Processor identifies host resource details in alignment with OpenTelemetry's resource semantic standards. Using the detected information, this processor can add or replace the resource values in telemetry data. This processor supports traces and metrics. You can use this processor with multiple detectors such as the Docket metadata detector or the **OTEL_RESOURCE_ATTRIBUTES** environment variable detector.

IMPORTANT

The Resource Detection Processor is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenShift Container Platform permissions required for the Resource Detection Processor

```
kind: ClusterRole
metadata:
  name: otel-collector
rules:
- apiGroups: ["config.openshift.io"]
  resources: ["infrastructures", "infrastructures/status"]
  verbs: ["get", "watch", "list"]
# ...
```

OpenTelemetry Collector using the Resource Detection Processor

```
# ...
  config:
    processors:
      resourcedetection:
        detectors: [openshift]
        override: true
    service:
      pipelines:
        traces:
          processors: [resourcedetection]
        metrics:
          processors: [resourcedetection]
# ...
```

OpenTelemetry Collector using the Resource Detection Processor with an environment variable detector

```
# ...
  config:
    processors:
      resourcedetection/env:
        detectors: [env] 1
        timeout: 2s
        override: false
# ...
```

1 Specifies which detector to use. In this example, the environment detector is specified.

4.3.4. Attributes Processor

The Attributes Processor can modify attributes of a span, log, or metric. You can configure this processor to filter and match input data and include or exclude such data for specific actions.

This processor operates on a list of actions, executing them in the order specified in the configuration. The following actions are supported:

Insert

Inserts a new attribute into the input data when the specified key does not already exist.

Update

Updates an attribute in the input data if the key already exists.

Upsert

Combines the insert and update actions: Inserts a new attribute if the key does not exist yet. Updates the attribute if the key already exists.

Delete

Removes an attribute from the input data.

Hash

Hashes an existing attribute value as SHA1.

Extract

Extracts values by using a regular expression rule from the input key to the target keys defined in the rule. If a target key already exists, it is overridden similarly to the Span Processor's **to_attributes** setting with the existing attribute as the source.

Convert

Converts an existing attribute to a specified type.

OpenTelemetry Collector using the Attributes Processor

```
# ...
  config:
    processors:
      attributes/example:
        actions:
          - key: db.table
            action: delete
          - key: redacted_span
            value: true
            action: upsert
          - key: copy_key
            from_attribute: key_original
            action: update
          - key: account_id
            value: 2245
            action: insert
          - key: account_password
            action: delete
          - key: account_email
            action: hash
          - key: http.status_code
```

```
          action: convert
          converted_type: int
# ...
```

4.3.5. Resource Processor

The Resource Processor applies changes to the resource attributes. This processor supports traces, metrics, and logs.

OpenTelemetry Collector using the Resource Detection Processor

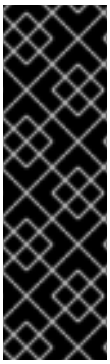
```
# ...
  config:
    processors:
      attributes:
      - key: cloud.availability_zone
        value: "zone-1"
        action: upsert
      - key: k8s.cluster.name
        from_attribute: k8s-cluster
        action: insert
      - key: redundant-attribute
        action: delete
# ...
```

Attributes represent the actions that are applied to the resource attributes, such as delete the attribute, insert the attribute, or upsert the attribute.

4.3.6. Span Processor

The Span Processor modifies the span name based on its attributes or extracts the span attributes from the span name. This processor can also change the span status and include or exclude spans. This processor supports traces.

Span renaming requires specifying attributes for the new name by using the **from_attributes** configuration.



IMPORTANT

The Span Processor is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector using the Span Processor for renaming a span

```
# ...
  config:
    processors:
      span:
```

```
      name:
        from_attributes: [<key1>, <key2>, ...] 1
        separator: <value> 2
# ...
```

1 Defines the keys to form the new span name.

2 An optional separator.

You can use this processor to extract attributes from the span name.

OpenTelemetry Collector using the Span Processor for extracting attributes from a span name

```
# ...
  config:
    processors:
      span/to_attributes:
        name:
          to_attributes:
            rules:
              - ^\/api\/v1\/document\/(?P<documentId>.*)\/update$ 1
# ...
```

1 This rule defines how the extraction is to be executed. You can define more rules: for example, in this case, if the regular expression matches the name, a **documentID** attibute is created. In this example, if the input span name is **/api/v1/document/12345678/update**, this results in the **/api/v1/document/{documentId}/update** output span name, and a new **"documentId"="12345678"** attribute is added to the span.

You can have the span status modified.

OpenTelemetry Collector using the Span Processor for status change

```
# ...
  config:
    processors:
      span/set_status:
        status:
          code: Error
          description: "<error_description>"
# ...
```

4.3.7. Kubernetes Attributes Processor

The Kubernetes Attributes Processor enables automatic configuration of spans, metrics, and log resource attributes by using the Kubernetes metadata. This processor supports traces, metrics, and logs. This processor automatically identifies the Kubernetes resources, extracts the metadata from them, and incorporates this extracted metadata as resource attributes into relevant spans, metrics, and logs. It utilizes the Kubernetes API to discover all pods operating within a cluster, maintaining records of their IP addresses, pod UIDs, and other relevant metadata.

Minimum OpenShift Container Platform permissions required for the Kubernetes Attributes Processor

```
kind: ClusterRole
metadata:
  name: otel-collector
rules:
  - apiGroups: ['']
    resources: ['pods', 'namespaces']
    verbs: ['get', 'watch', 'list']
  - apiGroups: ['apps']
    resources: ['replicasets']
    verbs: ['get', 'watch', 'list']
# ...
```

OpenTelemetry Collector using the Kubernetes Attributes Processor

```
# ...
  config:
    processors:
        k8sattributes:
            filter:
                node_from_env_var: KUBE_NODE_NAME
# ...
```

4.3.8. Filter Processor

The Filter Processor leverages the OpenTelemetry Transformation Language to establish criteria for discarding telemetry data. If any of these conditions are satisfied, the telemetry data are discarded. You can combine the conditions by using the logical OR operator. This processor supports traces, metrics, and logs.

IMPORTANT

The Filter Processor is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with an enabled OTLP Exporter

```
# ...
  config:
    processors:
      filter/ottl:
        error_mode: ignore (1)
        traces:
          span:
```

```
        - 'attributes["container.name"] == "app_container_1"' 2
        - 'resource.attributes["host.name"] == "localhost"' 3
# ...
```

1. Defines the error mode. When set to **ignore**, ignores errors returned by conditions. When set to **propagate**, returns the error up the pipeline. An error causes the payload to be dropped from the Collector.
2. Filters the spans that have the **container.name == app_container_1** attribute.
3. Filters the spans that have the **host.name == localhost** resource attribute.

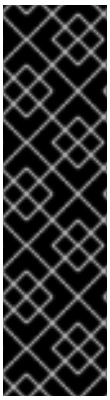
4.3.9. Cumulative-to-Delta Processor

The Cumulative-to-Delta Processor converts monotonic, cumulative-sum, and histogram metrics to monotonic delta metrics.

You can filter metrics by using the **include:** or **exclude:** fields and specifying the **strict** or **regexp** metric name matching.

Because this processor calculates delta by storing the previous value of a metric, you must set up the metric source to send the metric data to a single stateful Collector instance rather than a deployment of multiple Collectors.

This processor does not convert non-monotonic sums and exponential histograms.



IMPORTANT

The Cumulative-to-Delta Processor is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

Example of an OpenTelemetry Collector custom resource with an enabled Cumulative-to-Delta Processor

```
# ...
mode: sidecar 1
config:
  processors:
    cumulativetodelta:
      include: 2
        match_type: strict 3
        metrics: 4
        - <metric_1_name>
        - <metric_2_name>
      exclude: 5
        match_type: regexp
```

```
      metrics:
      - "<regular_expression_for_metric_names>"
# ...
```

1. To tie the Collector's lifecycle to the metric source, you can run the Collector as a sidecar to the application that emits the cumulative temporality metrics.
2. Optional: You can limit which metrics the processor converts by explicitly defining which metrics you want converted in this stanza. If you omit this field, the processor converts all metrics, except the metrics that are listed in the **exclude** field.
3. Defines the value that you provided in the **metrics** field as an exact match by using the **strict** parameter or a regular expression by using the **regex** parameter.
4. Lists the names of the metrics that you want to convert. The processor converts exact matches or matches for regular expressions. If a metric matches both the **include** and **exclude** filters, the **exclude** filter takes precedence.
5. Optional: You can exclude certain metrics from conversion by explicitly defining them here.

4.3.10. Group-by-Attributes Processor

The Group-by-Attributes Processor groups all spans, log records, and metric datapoints that share the same attributes by reassigning them to a Resource that matches those attributes.

IMPORTANT

The Group-by-Attributes Processor is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

At minimum, configuring this processor involves specifying an array of attribute keys to be used to group spans, log records, or metric datapoints together, as in the following example:

Example of the OpenTelemetry Collector custom resource when using the Group-by-Attributes Processor

```
# ...
  config:
    processors:
      groupbyattrs:
        keys: 1
          - <key1> 2
          - <key2>
# ...
```

1. Specifies attribute keys to group by.

2 If a processed span, log record, or metric datapoint contains at least one of the specified attribute keys, it is reassigned to a Resource that shares the same attribute values; and if no such Resource

4.3.11. Transform Processor

The Transform Processor enables modification of telemetry data according to specified rules and in the OpenTelemetry Transformation Language (OTTL). For each signal type, the processor processes a series of conditions and statements associated with a specific OTTL Context type and then executes them in sequence on incoming telemetry data as specified in the configuration. Each condition and statement can access and modify telemetry data by using various functions, allowing conditions to dictate if a function is to be executed.

All statements are written in the OTTL. You can configure multiple context statements for different signals, traces, metrics, and logs. The value of the **context** type specifies which OTTL Context the processor must use when interpreting the associated statements.

IMPORTANT

The Transform Processor is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

Configuration summary

```
# ...
config:
  processors:
    transform:
      error_mode: ignore 1
      <trace|metric|log>_statements: 2
        - context: <string> 3
          conditions: 4
            - <string>
            - <string>
          statements: 5
            - <string>
            - <string>
            - <string>
        - context: <string>
          statements:
            - <string>
            - <string>
            - <string>
# ...
```

1 Optional: See the following table "Values for the optional **error_mode** field".

2 Indicates a signal to be transformed.

3 See the following table "Values for the **context** field".

4 Optional: Conditions for performing a transformation.

Example of the OpenTelemetry Collector custom resource when using the Transform Processor

```
# ...
  config:
    transform:
      error_mode: ignore
      trace_statements: 1
        - context: resource
          statements:
            - keep_keys(attributes, ["service.name", "service.namespace", "cloud.region",
"process.command_line"]) 2
            - replace_pattern(attributes["process.command_line"], "password\\=[^\\s]*(\\s?)",
"password=***") 3
            - limit(attributes, 100, [])
            - truncate_all(attributes, 4096)
        - context: span 4
          statements:
            - set(status.code, 1) where attributes["http.path"] == "/health"
            - set(name, attributes["http.route"])
            - replace_match(attributes["http.target"], "/user/*/list/*", "/user/{userId}/list/{listId}")
            - limit(attributes, 100, [])
            - truncate_all(attributes, 4096)
# ...
```

1 Transforms a trace signal.

2 Keeps keys on the resources.

3 Replaces attributes and replaces string characters in password fields with asterisks.

4 Performs transformations at the span level.

Table 4.4. Values for the **context** field

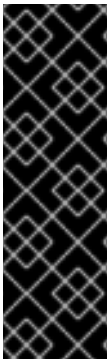
Signal Statement	Valid Contexts
trace_statements	**resource**, **scope**, **span**, **spanevent**
metric_statements	**resource**, **scope**, **metric**, **datapoint**
log_statements	**resource**, **scope**, **log**

Table 4.5. Values for the optional **error_mode** field

Value	Description
ignore	Ignores and logs errors returned by statements and then continues to the next statement.
silent	Ignores and doesn't log errors returned by statements and then continues to the next statement.
propagate	Returns errors up the pipeline and drops the payload. Implicit default.

4.3.12. Tail Sampling Processor

The Tail Sampling Processor samples traces according to user-defined policies when all of the spans are completed. Tail-based sampling enables you to filter the traces of interest and reduce your data ingestion and storage costs.



IMPORTANT

The Tail Sampling Processor is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

This processor reassembles spans into new batches and strips spans of their original context.

TIP

- In pipelines, place this processor downstream of any processors that rely on context: for example, after the Kubernetes Attributes Processor.
- If scaling the Collector, ensure that one Collector instance receives all spans of the same trace so that this processor makes correct sampling decisions based on the specified sampling policies. You can achieve this by setting up two layers of Collectors: the first layer of Collectors with the Load Balancing Exporter, and the second layer of Collectors with the Tail Sampling Processor.

Example of the OpenTelemetry Collector custom resource when using the Tail Sampling Processor

```
# ...
config:
  processors:
    tail_sampling: 1
      decision_wait: 30s 2
      num_traces: 50000 3
```

```
    expected_new_traces_per_sec: 10 4
    policies: 5
     [
      {
       <definition_of_policy_1>
      },
      {
       <definition_of_policy_2>
      },
      {
       <definition_of_policy_3>
      },
     ]
# ...
```

1. Processor name.
2. Optional: Decision delay time, counted from the time of the first span, before the processor makes a sampling decision on each trace. Defaults to **30s**.
3. Optional: The number of traces kept in memory. Defaults to **50000**.
4. Optional: The expected number of new traces per second, which is helpful for allocating data structures. Defaults to **0**.
5. Definitions of the policies for trace evaluation. The processor evaluates each trace against all of the specified policies and then either samples or drops the trace.

You can choose and combine policies from the following list:

- The following policy samples all traces:

```
# ...
    policies:
     [
      {
       name: <always_sample_policy>,
       type: always_sample,
      },
     ]
# ...
```

- The following policy samples only traces of a duration that is within a specified range:

```
# ...
    policies:
     [
      {
       name: <latency_policy>,
       type: latency,
       latency: {threshold_ms: 5000, upper_threshold_ms: 10000} 1
      },
     ]
# ...
```

1 The provided **5000** and **10000** values are examples. You can estimate the desired latency values by looking at the earliest start time value and latest end time value. If you omit the **upper_threshold_ms** field, this policy samples all latencies greater than the specified **threshold_ms** value.

- The following policy samples traces by numeric value matches for resource and record attributes:

```
# ...
    policies:
      [
        {
          name: <numeric_attribute_policy>,
          type: numeric_attribute,
          numeric_attribute: {key: <key1>, min_value: 50, max_value: 100} 1
        },
      ]
# ...
```

1 The provided **50** and **100** values are examples.

- The following policy samples only a percentage of traces:

```
# ...
    policies:
      [
        {
          name: <probabilistic_policy>,
          type: probabilistic,
          probabilistic: {sampling_percentage: 10} 1
        },
      ]
# ...
```

1 The provided **10** value is an example.

- The following policy samples traces by the status code: **OK**, **ERROR**, or **UNSET**:

```
# ...
    policies:
      [
        {
          name: <status_code_policy>,
          type: status_code,
          status_code: {status_codes: [ERROR, UNSET]}
        },
      ]
# ...
```

- The following policy samples traces by string value matches for resource and record attributes:

```
# ...
```

```
      policies:
        [
          {
            name: <string_attribute_policy>,
            type: string_attribute,
            string_attribute: {key: <key2>, values: [<value1>, <val>*], enabled_regex_matching:
true, cache_max_size: 10} (1)
          },
        ]
# ...
```

(1) This policy definition supports both exact and regular-expression value matches. The provided **10** value in the **cache_max_size** field is an example.

- The following policy samples traces by the rate of spans per second:

```
# ...
    policies:
      [
        {
          name: <rate_limiting_policy>,
          type: rate_limiting,
          rate_limiting: {spans_per_second: 35} (1)
        },
      ]
# ...
```

(1) The provided **35** value is an example.

- The following policy samples traces by the minimum and maximum number of spans inclusively:

```
# ...
    policies:
      [
        {
          name: <span_count_policy>,
          type: span_count,
          span_count: {min_spans: 2, max_spans: 20} (1)
        },
      ]
# ...
```

(1) If the sum of all spans in the trace is outside the range threshold, the trace is not sampled. The provided **2** and **20** values are examples.

- The following policy samples traces by **TraceState** value matches:

```
# ...
    policies:
      [
        {
          name: <trace_state_policy>,
```

```
        type: trace_state,
        trace_state: { key: <key3>, values: [<value1>, <value2>] }
      },
    ]
# ...
```

- The following policy samples traces by a boolean attribute (resource and record):

```
# ...
    policies:
      [
        {
          name: <bool_attribute_policy>,
          type: boolean_attribute,
          boolean_attribute: {key: <key4>, value: true}
        },
      ]
# ...
```

- The following policy samples traces by a given boolean OTTL condition for a span or span event:

```
# ...
    policies:
      [
        {
          name: <ottl_policy>,
          type: ottl_condition,
          ottl_condition: {
            error_mode: ignore,
            span: [
              "attributes[\"<test_attr_key_1>\"] == \"<test_attr_value_1>\"",
              "attributes[\"<test_attr_key_2>\"] != \"<test_attr_value_1>\"",
            ],
            spanevent: [
              "name != \"<test_span_event_name>\"",
              "attributes[\"<test_event_attr_key_2>\"] != \"<test_event_attr_value_1>\"",
            ]
          }
        },
      ]
# ...
```

- The following is an **AND** policy that samples traces based on a combination of multiple policies:

```
# ...
    policies:
      [
        {
          name: <and_policy>,
          type: and,
          and: {
            and_sub_policy:
            [
              {
                name: <and_policy_1>,
```

```
            type: numeric_attribute,
            numeric_attribute: { key: <key1>, min_value: 50, max_value: 100 } (1)
          },
          {
            name: <and_policy_2>,
            type: string_attribute,
            string_attribute: { key: <key2>, values: [ <value1>, <value2> ] }
          },
        ]
      }
    },
  ]
# ...
```

(1) The provided **50** and **100** values are examples.

- The following is a **DROP** policy that drops traces from sampling based on a combination of multiple policies:

```
# ...
    policies:
      [
        {
          name: <drop_policy>,
          type: drop,
          drop: {
            drop_sub_policy:
            [
              {
                name: <drop_policy_1>,
                type: string_attribute,
                string_attribute: {key: url.path, values: [\/health, \/metrics],
enabled_regex_matching: true}
              }
            ]
          }
        },
      ]
# ...
```

- The following policy samples traces by a combination of the previous samplers and with ordering and rate allocation per sampler:

```
# ...
    policies:
      [
        {
          name: <composite_policy>,
          type: composite,
          composite:
            {
              max_total_spans_per_second: 100, (1)
              policy_order: [<composite_policy_1>, <composite_policy_2>,
<composite_policy_3>],
```

```
            composite_sub_policy:
             [
              {
               name: <composite_policy_1>,
               type: numeric_attribute,
               numeric_attribute: {key: <key1>, min_value: 50}
              },
              {
               name: <composite_policy_2>,
               type: string_attribute,
               string_attribute: {key: <key2>, values: [<value1>, <value2>]}
              },
              {
               name: <composite_policy_3>,
               type: always_sample
              }
             ],
             rate_allocation:
             [
              {
               policy: <composite_policy_1>,
               percent: 50 (2)
              },
              {
               policy: <composite_policy_2>,
               percent: 25
              }
             ]
           }
         },
       ]
# ...
```

(1) (2) Allocates percentages of spans according to the order of applied policies. For example, if you set the **100** value in the **max_total_spans_per_second** field, you can set the following values in the **rate_allocation** section: the **50** percent value in the **policy: <composite_policy_1>** section to allocate 50 spans per second, and the **25** percent value in the **policy: <composite_policy_2>** section to allocate 25 spans per second. To fill the remaining capacity, you can set the **always_sample** value in the **type** field of the **name: <composite_policy_3>** section.

Additional resources

- Tail Sampling with OpenTelemetry: Why it's useful, how to do it, and what to consider (OpenTelemetry Blog)
- Gateway (OpenTelemetry Documentation)

4.3.13. Probabilistic Sampling Processor

If you handle high volumes of telemetry data and seek to reduce costs by reducing processed data volumes, you can use the Probabilistic Sampling Processor as an alternative to the Tail Sampling Processor.

IMPORTANT

Probabilistic Sampling Processor is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

The processor samples a specified percentage of trace spans or log records statelessly and per request.

The processor adds the information about the used effective sampling probability into the telemetry data:

- In trace spans, the processor encodes the threshold and optional randomness information in the W3C Trace Context **tracestate** fields.
- In log records, the processor encodes the threshold and randomness information as attributes.

The following is an example **OpenTelemetryCollector** custom resource configuration for the Probabilistic Sampling Processor for sampling trace spans:

```
# ...
  config:
    processors:
      probabilistic_sampler: 1
        sampling_percentage: 15.3 2
        mode: "proportional" 3
        hash_seed: 22 4
        sampling_precision: 14 5
        fail_closed: true 6
# ...
service:
  pipelines:
    traces:
      processors: [probabilistic_sampler]
# ...
```

1. For trace pipelines, the source of randomness is the hashed value of the span trace ID.
2. Required. Accepts a 32-bit floating-point percentage value at which spans are to be sampled.
3. Optional. Accepts a supported string value for a sampling logic mode: the default **hash_seed**, **proportional**, or **equalizing**. The **hash_seed** mode applies the Fowler–Noll–Vo (FNV) hash function to the trace ID and weighs the hashed value against the sampling percentage value. You can also use the **hash_seed** mode with units of telemetry other than the trace ID. The **proportional** mode samples a strict, probability-based ratio of the total span quantity, and is based on the OpenTelemetry and World Wide Web Consortium specifications. The **equalizing** mode is useful for lowering the sampling probability to a minimum value across a whole pipeline or applying a uniform sampling probability in Collector deployments where client SDKs have mixed sampling configurations.

4 Optional. Accepts a 32-bit unsigned integer, which is used to compute the hash algorithm. When this field is not configured, the default seed value is **0**. If you use multiple tiers of Collector

5 Optional. Determines the number of hexadecimal digits used to encode the sampling threshold. Accepts an integer value. The supported values are **1**-**14**. The default value **4** causes the threshold to be rounded if it contains more than 16 significant bits, which is the case of the **proportional** mode that uses 56 bits. If you select the **proportional** mode, use a greater value for the purpose of preserving precision applied by preceding samplers.

6 Optional. Rejects spans with sampling errors. Accepts a boolean value. The default value is **true**.

The following is an example **OpenTelemetryCollector** custom resource configuration for the Probabilistic Sampling Processor for sampling log records:

```
# ...
  config:
    processors:
      probabilistic_sampler/logs:
        sampling_percentage: 15.3 1
        mode: "hash_seed" 2
        hash_seed: 22 3
        sampling_precision: 4 4
        attribute_source: "record" 5
        from_attribute: "<log_record_attribute_name>" 6
        fail_closed: true 7
# ...
service:
  pipelines:
    logs:
      processors: [ probabilistic_sampler/logs ]
# ...
```

1 Required. Accepts a 32-bit floating-point percentage value at which spans are to be sampled.

2 Optional. Accepts a supported string value for a sampling logic mode: the default **hash_seed**, **equalizing**, or **proportional**. The **hash_seed** mode applies the Fowler–Noll–Vo (FNV) hash function to the trace ID or a specified log record attribute and then weighs the hashed value against the sampling percentage value. You can also use **hash_seed** mode with other units of telemetry than trace ID, for example to use the **service.instance.id** resource attribute for collecting log records from a percentage of pods. The **equalizing** mode is useful for lowering the sampling probability to a minimum value across a whole pipeline or applying a uniform sampling probability in Collector deployments where client SDKs have mixed sampling configurations. The **proportional** mode samples a strict, probability-based ratio of the total span quantity, and is based on the OpenTelemetry and World Wide Web Consortium specifications.

3 Optional. Accepts a 32-bit unsigned integer, which is used to compute the hash algorithm. When this field is not configured, the default seed value is **0**. If you use multiple tiers of Collector instances, you must configure all Collectors of the same tier to the same seed value.

4 Optional. Determines the number of hexadecimal digits used to encode the sampling threshold. Accepts an integer value. The supported values are **1**-**14**. The default value **4** causes the threshold to be rounded if it contains more than 16 significant bits, which is the case of the **proportional** mode that uses 56 bits. If you select the **proportional** mode, use a greater value for the purpose of

preserving precision applied by preceding samplers.

5 Optional. Defines where to look for the log record attribute in **from_attribute**. The log record attribute is used as the source of randomness. Accept the default **traceID** value or the **record** value.

6 Optional. The name of a log record attribute to be used to compute the sampling hash, such as a unique log record ID. Accepts a string value. The default value is **""**. Use this field only if you need to specify a log record attribute as the source of randomness in those situations where the trace ID is absent or trace ID sampling is disabled or the **attribute_source** field is set to the **record** value.

7 Optional. Rejects spans with sampling errors. Accepts a boolean value. The default value is **true**.

4.3.14. Additional resources

- OpenTelemetry Protocol (OTLP) (OpenTelemetry Documentation)

4.4. EXPORTERS

Exporters send data to one or more back ends or destinations. An exporter can be push or pull based. By default, no exporters are configured. One or more exporters must be configured. Exporters can support one or more data sources. Exporters might be used with their default settings, but many exporters require configuration to specify at least the destination and security settings.

Currently, the following General Availability and Technology Preview exporters are available for the Red Hat build of OpenTelemetry:

- OTLP Exporter
- OTLP HTTP Exporter
- Debug Exporter
- Load Balancing Exporter
- Prometheus Exporter
- Prometheus Remote Write Exporter
- Kafka Exporter
- AWS CloudWatch Logs Exporter
- AWS EMF Exporter
- AWS X-Ray Exporter
- File Exporter

4.4.1. OTLP Exporter

The OTLP gRPC Exporter exports traces and metrics by using the OpenTelemetry protocol (OTLP).

OpenTelemetry Collector custom resource with the enabled OTLP Exporter

-

```
# ...
 config:
   exporters:
     otlp:
       endpoint: tempo-ingester:4317 (1)
       tls: (2)
         ca_file: ca.pem
         cert_file: cert.pem
         key_file: key.pem
         insecure: false (3)
         insecure_skip_verify: false # (4)
         reload_interval: 1h (5)
         server_name_override: <name> (6)
       headers: (7)
         X-Scope-OrgID: "dev"
   service:
     pipelines:
       traces:
         exporters: [otlp]
       metrics:
         exporters: [otlp]
# ...
```

1. The OTLP gRPC endpoint. If the **https://** scheme is used, then client transport security is enabled and overrides the **insecure** setting in the **tls**.
2. The client-side TLS configuration. Defines paths to TLS certificates.
3. Disables client transport security when set to **true**. The default value is **false** by default.
4. Skips verifying the certificate when set to **true**. The default value is **false**.
5. Specifies the time interval at which the certificate is reloaded. If the value is not set, the certificate is never reloaded. The **reload_interval** accepts a string containing valid units of time such as **ns**, **us** (or **µs**), **ms**, **s**, **m**, **h**.
6. Overrides the virtual host name of authority such as the authority header field in requests. You can use this for testing.
7. Headers are sent for every request performed during an established connection.

4.4.2. OTLP HTTP Exporter

The OTLP HTTP Exporter exports traces and metrics by using the OpenTelemetry protocol (OTLP).

OpenTelemetry Collector custom resource with the enabled OTLP Exporter

```
# ...
 config:
   exporters:
     otlphttp:
       endpoint: http://tempo-ingester:4318 (1)
       tls: (2)
```

```
      headers: 3
        X-Scope-OrgID: "dev"
      disable_keep_alives: false 4

  service:
    pipelines:
      traces:
        exporters: [otlphttp]
      metrics:
        exporters: [otlphttp]
# ...
```

1 The OTLP HTTP endpoint. If the **https://** scheme is used, then client transport security is enabled and overrides the **insecure** setting in the **tls**.

2 The client side TLS configuration. Defines paths to TLS certificates.

3 Headers are sent in every HTTP request.

4 If true, disables HTTP keep-alives. It will only use the connection to the server for a single HTTP request.

4.4.3. Debug Exporter

The Debug Exporter prints traces and metrics to the standard output.

OpenTelemetry Collector custom resource with the enabled Debug Exporter

```
# ...
  config:
    exporters:
      debug:
        verbosity: detailed 1
        sampling_initial: 5 2
        sampling_thereafter: 200 3
        use_internal_logger: true 4
    service:
      pipelines:
        traces:
          exporters: [debug]
        metrics:
          exporters: [debug]
# ...
```

1 Verbosity of the debug export: **detailed**, **normal**, or **basic**. When set to **detailed**, pipeline data are verbosely logged. Defaults to **normal**.

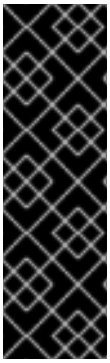
2 Initial number of messages logged per second. The default value is **2** messages per second.

3 Sampling rate after the initial number of messages, the value in **sampling_initial**, has been logged. Disabled by default with the default **1** value. Sampling is enabled with values greater than **1**. For more information, see the page for the sampler function in the **zapcore** package on the Go Project's website.

4 When set to **true**, enables output from the Collector's internal logger for the exporter.

4.4.4. Load Balancing Exporter

The Load Balancing Exporter consistently exports spans, metrics, and logs according to the **routing_key** configuration.



IMPORTANT

The Load Balancing Exporter is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with the enabled Load Balancing Exporter

```
# ...
  config:
    exporters:
      loadbalancing:
        routing_key: "service" (1)
        protocol:
          otlp: (2)
            timeout: 1s
        resolver: (3)
          static: (4)
            hostnames:
            - backend-1:4317
            - backend-2:4317
          dns: (5)
            hostname: otelcol-headless.observability.svc.cluster.local
          k8s: (6)
            service: lb-svc.kube-public
            ports:
              - 15317
              - 16317
# ...
```

1 The **routing_key: service** exports spans for the same service name to the same Collector instance to provide accurate aggregation. The **routing_key: traceID** exports spans based on their **traceID**. The implicit default is **traceID** based routing.

2 The OTLP is the only supported load-balancing protocol. All options of the OTLP exporter are supported.

3 You can configure only one resolver.

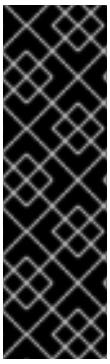
4 The static resolver distributes the load across the listed endpoints.

5 You can use the DNS resolver only with a Kubernetes headless service.

6 The Kubernetes resolver is recommended.

4.4.5. Prometheus Exporter

The Prometheus Exporter exports metrics in the Prometheus or OpenMetrics formats.



IMPORTANT

The Prometheus Exporter is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with the enabled Prometheus Exporter

```
# ...
  config:
    exporters:
      prometheus:
        endpoint: 0.0.0.0:8889 1
        tls: 2
          ca_file: ca.pem
          cert_file: cert.pem
          key_file: key.pem
        namespace: prefix 3
        const_labels: 4
          label1: value1
        enable_open_metrics: true 5
        resource_to_telemetry_conversion: 6
          enabled: true
        metric_expiration: 180m 7
        add_metric_suffixes: false 8
    service:
      pipelines:
        metrics:
          exporters: [prometheus]
# ...
```

1 The network endpoint where the metrics are exposed. The Red Hat build of OpenTelemetry Operator automatically exposes the port specified in the **endpoint** field to the **<instance_name>-collector** service.

2 The server-side TLS configuration. Defines paths to TLS certificates.

3 If set, exports metrics under the provided value.

4 Key-value pair labels that are applied for every exported metric.

5 If **true**, metrics are exported by using the OpenMetrics format. Exemplars are only exported in the OpenMetrics format and only for histogram and monotonic sum metrics such as **counter**. Disabled by default.

6 If **enabled** is **true**, all the resource attributes are converted to metric labels. Disabled by default.

7 Defines how long metrics are exposed without updates. The default is **5m**.

8 Adds the metrics types and units suffixes. Must be disabled if the monitor tab in the Jaeger console is enabled. The default is **true**.

NOTE

When the **spec.observability.metrics.enableMetrics** field in the **OpenTelemetryCollector** custom resource (CR) is set to **true**, the **OpenTelemetryCollector** CR automatically creates a Prometheus **ServiceMonitor** or **PodMonitor** CR to enable Prometheus to scrape your metrics.

4.4.6. Prometheus Remote Write Exporter

The Prometheus Remote Write Exporter exports metrics to compatible back ends.

IMPORTANT

The Prometheus Remote Write Exporter is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with the enabled Prometheus Remote Write Exporter

```
# ...
  config:
    exporters:
      prometheusremotewrite:
        endpoint: "https://my-prometheus:7900/api/v1/push" 1
        tls: 2
          ca_file: ca.pem
          cert_file: cert.pem
          key_file: key.pem
        target_info: true 3
        export_created_metric: true 4
        max_batch_size_bytes: 3000000 5
    service:
```

```
    pipelines:
      metrics:
        exporters: [prometheusremotewrite]
# ...
```

1. Endpoint for sending the metrics.
2. Server-side TLS configuration. Defines paths to TLS certificates.
3. When set to **true**, creates a **target_info** metric for each resource metric.
4. When set to **true**, exports a **_created** metric for the Summary, Histogram, and Monotonic Sum metric points.
5. Maximum size of the batch of samples that is sent to the remote write endpoint. Exceeding this value results in batch splitting. The default value is **3000000**, which is approximately 2.861 megabytes.

WARNING

- This exporter drops non-cumulative monotonic, histogram, and summary OTLP metrics.
- You must enable the **--web.enable-remote-write-receiver** feature flag on the remote Prometheus instance. Without it, pushing the metrics to the instance using this exporter fails.

4.4.7. Kafka Exporter

The Kafka Exporter exports logs, metrics, and traces to Kafka. This exporter uses a synchronous producer that blocks and does not batch messages. You must use it with batch and queued retry processors for higher throughput and resiliency.

OpenTelemetry Collector custom resource with the enabled Kafka Exporter

```
# ...
 config:
   exporters:
     kafka:
       brokers: ["localhost:9092"] 1
       protocol_version: 2.0.0 2
       topic: otlp_spans 3
       auth:
         plain_text: 4
           username: example
           password: example
         tls: 5
           ca_file: ca.pem
           cert_file: cert.pem
```

```
        key_file: key.pem
        insecure: false 6
        server_name_override: kafka.example.corp 7
    service:
      pipelines:
        traces:
          exporters: [kafka]
# ...
```

1. The list of Kafka brokers. The default is **localhost:9092**.
2. The Kafka protocol version. For example, **2.0.0**. This is a required field.
3. The name of the Kafka topic to read from. The following are the defaults: **otlp_spans** for traces, **otlp_metrics** for metrics, **otlp_logs** for logs.
4. The plain text authentication configuration. If omitted, plain text authentication is disabled.
5. The client-side TLS configuration. Defines paths to the TLS certificates. If omitted, TLS authentication is disabled.
6. Disables verifying the server's certificate chain and host name. The default is **false**.
7. ServerName indicates the name of the server requested by the client to support virtual hosting.

4.4.8. AWS CloudWatch Logs Exporter

The AWS CloudWatch Logs Exporter sends logs data to the Amazon CloudWatch Logs service and signs requests by using the AWS SDK for Go and the default credential provider chain.

IMPORTANT

The AWS CloudWatch Logs Exporter is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with the enabled AWS CloudWatch Logs Exporter

```
# ...
  config:
    exporters:
      awscloudwatchlogs:
        log_group_name: "<group_name_of_amazon_cloudwatch_logs>" 1
        log_stream_name: "<log_stream_of_amazon_cloudwatch_logs>" 2
        region: <aws_region_of_log_stream> 3
```

```
      endpoint: <protocol><service_endpoint_of_amazon_cloudwatch_logs> 4
      log_retention: <supported_value_in_days> 5
# ...
```

1. Required. If the log group does not exist yet, it is automatically created.
2. Required. If the log stream does not exist yet, it is automatically created.
3. Optional. If the AWS region is not already set in the default credential chain, you must specify it.
4. Optional. You can override the default Amazon CloudWatch Logs service endpoint to which the requests are forwarded. You must include the protocol, such as **https://**, as part of the endpoint value. For the list of service endpoints by region, see Amazon CloudWatch Logs endpoints and quotas (AWS General Reference).
5. Optional. With this parameter, you can set the log retention policy for new Amazon CloudWatch log groups. If this parameter is omitted or set to **0**, the logs never expire by default. Supported values for retention in days are **1**, **3**, **5**, **7**, **14**, **30**, **60**, **90**, **120**, **150**, **180**, **365**, **400**, **545**, **731**, **1827**, **2192**, **2557**, **2922**, **3288**, or **3653**.

Additional resources

- What is Amazon CloudWatch Logs? (Amazon CloudWatch Logs User Guide)
- Specifying Credentials (AWS SDK for Go Developer Guide)
- Amazon CloudWatch Logs endpoints and quotas (AWS General Reference)

4.4.9. AWS EMF Exporter

The AWS EMF Exporter converts the following OpenTelemetry metrics datapoints to the AWS CloudWatch Embedded Metric Format (EMF):

- **Int64DataPoints**
- **DoubleDataPoints**
- **SummaryDataPoints**

The EMF metrics are then sent directly to the Amazon CloudWatch Logs service by using the **PutLogEvents** API.

One of the benefits of using this exporter is the possibility to view logs and metrics in the Amazon CloudWatch console at https://console.aws.amazon.com/cloudwatch/.

IMPORTANT

The AWS EMF Exporter is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with the enabled AWS EMF Exporter

```
# ...
  config:
    exporters:
      awsemf:
        log_group_name: "<group_name_of_amazon_cloudwatch_logs>" 1
        log_stream_name: "<log_stream_of_amazon_cloudwatch_logs>" 2
        resource_to_telemetry_conversion: 3
          enabled: true
        region: <region> 4
        endpoint: <protocol><endpoint> 5
        log_retention: <supported_value_in_days> 6
        namespace: <custom_namespace> 7
# ...
```

1. Customized log group name.
2. Customized log stream name.
3. Optional. Converts resource attributes to telemetry attributes such as metric labels. Disabled by default.
4. The AWS region of the log stream. If a region is not already set in the default credential provider chain, you must specify the region.
5. Optional. You can override the default Amazon CloudWatch Logs service endpoint to which the requests are forwarded. You must include the protocol, such as **https://**, as part of the endpoint value. For the list of service endpoints by region, see Amazon CloudWatch Logs endpoints and quotas (AWS General Reference).
6. Optional. With this parameter, you can set the log retention policy for new Amazon CloudWatch log groups. If this parameter is omitted or set to **0**, the logs never expire by default. Supported values for retention in days are **1**, **3**, **5**, **7**, **14**, **30**, **60**, **90**, **120**, **150**, **180**, **365**, **400**, **545**, **731**, **1827**, **2192**, **2557**, **2922**, **3288**, or **3653**.
7. Optional. A custom namespace for the Amazon CloudWatch metrics.

Log group name
The **log_group_name** parameter allows you to customize the log group name and supports the default **/metrics/default** value or the following placeholders:

/aws/metrics/{ClusterName}

This placeholder is used to search for the **ClusterName** or **aws.ecs.cluster.name** resource attribute in the metrics data and replace it with the actual cluster name.

{NodeName}

This placeholder is used to search for the **NodeName** or **k8s.node.name** resource attribute.

{TaskId}

This placeholder is used to search for the **TaskId** or **aws.ecs.task.id** resource attribute.

If no resource attribute is found in the resource attribute map, the placeholder is replaced by the **undefined** value.

Log stream name

The **log_stream_name** parameter allows you to customize the log stream name and supports the default **otel-stream** value or the following placeholders:

{ClusterName}

This placeholder is used to search for the **ClusterName** or **aws.ecs.cluster.name** resource attribute.

{ContainerInstanceId}

This placeholder is used to search for the **ContainerInstanceId** or **aws.ecs.container.instance.id** resource attribute. This resource attribute is valid only for the AWS ECS EC2 launch type.

{NodeName}

This placeholder is used to search for the **NodeName** or **k8s.node.name** resource attribute.

{TaskDefinitionFamily}

This placeholder is used to search for the **TaskDefinitionFamily** or **aws.ecs.task.family** resource attribute.

{TaskId}

This placeholder is used to search for the **TaskId** or **aws.ecs.task.id** resource attribute in the metrics data and replace it with the actual task ID.

If no resource attribute is found in the resource attribute map, the placeholder is replaced by the **undefined** value.

Additional resources

- Specification: Embedded metric format (Amazon CloudWatch User Guide)
- PutLogEvents (Amazon CloudWatch Logs API Reference)
- Amazon CloudWatch Logs endpoints and quotas (AWS General Reference)

4.4.10. AWS X-Ray Exporter

The AWS X-Ray Exporter converts OpenTelemetry spans to AWS X-Ray Segment Documents and then sends them directly to the AWS X-Ray service. The AWS X-Ray Exporter uses the **PutTraceSegments** API and signs requests by using the AWS SDK for Go and the default credential provider chain.

IMPORTANT

The AWS X-Ray Exporter is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with the enabled AWS X-Ray Exporter

```
# ...
  config:
    exporters:
      awsxray:
        region: "<region>" 1
        endpoint: <protocol><endpoint> 2
        resource_arn: "<aws_resource_arn>" 3
        role_arn: "<iam_role>" 4
        indexed_attributes: [ "<indexed_attr_0>", "<indexed_attr_1>" ] 5
        aws_log_groups: ["<group1>", "<group2>"] 6
        request_timeout_seconds: 120 7
# ...
```

1. The destination region for the X-Ray segments sent to the AWS X-Ray service. For example, **eu-west-1**.
2. Optional. You can override the default Amazon CloudWatch Logs service endpoint to which the requests are forwarded. You must include the protocol, such as **https://**, as part of the endpoint value. For the list of service endpoints by region, see Amazon CloudWatch Logs endpoints and quotas (AWS General Reference).
3. The Amazon Resource Name (ARN) of the AWS resource that is running the Collector.
4. The AWS Identity and Access Management (IAM) role for uploading the X-Ray segments to a different account.
5. The list of attribute names to be converted to X-Ray annotations.
6. The list of log group names for Amazon CloudWatch Logs.
7. Time duration in seconds before timing out a request. If omitted, the default value is **30**.

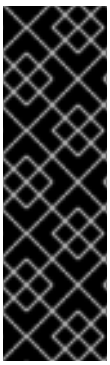
Additional resources

- What is AWS X-Ray? (AWS X-Ray Developer Guide)
- AWS SDK for Go API Reference (AWS Documentation)
- Specifying Credentials (AWS SDK for Go Developer Guide)

- IAM roles (AWS Identity and Access Management User Guide)

4.4.11. File Exporter

The File Exporter writes telemetry data to files in persistent storage and supports file operations such as rotation, compression, and writing to multiple files. With this exporter, you can also use a resource attribute to control file naming. The only required setting is **path**, which specifies the destination path for telemetry files in the persistent-volume file system.



IMPORTANT

The File Exporter is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with the enabled File Exporter

```
# ...
  config: |
    exporters:
      file:
        path: /data/metrics.json 1
        rotation: 2
          max_megabytes: 10 3
          max_days: 3 4
          max_backups: 3 5
          localtime: true 6
        format: proto 7
        compression: zstd 8
        flush_interval: 5 9
# ...
```

1. The file-system path where the data is to be written. There is no default.
2. File rotation is an optional feature of this exporter. By default, telemetry data is exported to a single file. Add the **rotation** setting to enable file rotation.
3. The **max_megabytes** setting is the maximum size a file is allowed to reach until it is rotated. The default is **100**.
4. The **max_days** setting is for how many days a file is to be retained, counting from the timestamp in the file name. There is no default.
5. The **max_backups** setting is for retaining several older files. The defalt is **100**.
6. The **localtime** setting specifies the local-time format for the timestamp, which is appended to the file name in front of any extension, when the file is rotated. The default is the Coordinated Universal Time (UTC).

7 The format for encoding the telemetry data before writing it to a file. The default format is **json**. The **proto** format is also supported.

8 File compression is optional and not set by default. This setting defines the compression algorithm for the data that is exported to a file. Currently, only the **zstd** compression algorithm is supported. There is no default.

9 The time interval between flushes. A value without a unit is set in nanoseconds. This setting is ignored when file rotation is enabled through the **rotation** settings.

4.4.12. Additional resources

- OpenTelemetry Protocol (OTLP) (OpenTelemetry Documentation)

4.5. CONNECTORS

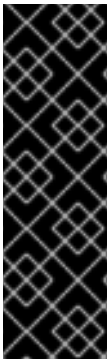
A connector connects two pipelines. It consumes data as an exporter at the end of one pipeline and emits data as a receiver at the start of another pipeline. It can consume and emit data of the same or different data type. It can generate and emit data to summarize the consumed data, or it can merely replicate or route data.

Currently, the following General Availability and Technology Preview connectors are available for the Red Hat build of OpenTelemetry:

- Count Connector
- Routing Connector
- Forward Connector
- Spanmetrics Connector

4.5.1. Count Connector

The Count Connector counts trace spans, trace span events, metrics, metric data points, and log records in exporter pipelines.



IMPORTANT

The Count Connector is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

The following are the default metric names:

- **trace.span.count**
- **trace.span.event.count**

- **metric.count**
- **metric.datapoint.count**
- **log.record.count**

You can also expose custom metric names.

OpenTelemetry Collector custom resource (CR) with an enabled Count Connector

```
# ...
  config:
    receivers:
      otlp:
        protocols:
          grpc:
            endpoint: 0.0.0.0:4317
    exporters:
      prometheus:
        endpoint: 0.0.0.0:8889
    connectors:
      count: {}
    service:
      pipelines: (1)
        traces/in:
          receivers: [otlp]
          exporters: [count] (2)
        metrics/out:
          receivers: [count] (3)
          exporters: [prometheus]
# ...
```

(1) It is important to correctly configure the Count Connector as an exporter or receiver in the pipeline and to export the generated metrics to the correct exporter.

(2) The Count Connector is configured to receive spans as an exporter.

(3) The Count Connector is configured to emit generated metrics as a receiver.

TIP

If the Count Connector is not generating the expected metrics, you can check whether the OpenTelemetry Collector is receiving the expected spans, metrics, and logs, and whether the telemetry data flow through the Count Connector as expected. You can also use the Debug Exporter to inspect the incoming telemetry data.

The Count Connector can count telemetry data according to defined conditions and expose those data as metrics when configured by using such fields as **spans**, **spanevents**, **metrics**, **datapoints**, or **logs**. See the next example.

Example OpenTelemetry Collector CR for the Count Connector to count spans by conditions

```
# ...
  config:
    connectors:
      count:
        spans: 1
          <custom_metric_name>: 2
            description: "<custom_metric_description>"
            conditions:
              - 'attributes["env"] == "dev"'
              - 'name == "devevent"'
# ...
```

1 In this example, the exposed metric counts spans with the specified conditions.

2 You can specify a custom metric name such as **cluster.prod.event.count**.

TIP

Write conditions correctly and follow the required syntax for attribute matching or telemetry field conditions. Improperly defined conditions are the most likely sources of errors.

The Count Connector can count telemetry data according to defined attributes when configured by using such fields as **spans**, **spanevents**, **metrics**, **datapoints**, or **logs**. See the next example. The attribute keys are injected into the telemetry data. You must define a value for the **default_value** field for missing attributes.

Example OpenTelemetry Collector CR for the Count Connector to count logs by attributes

```
# ...
  config:
    connectors:
      count:
        logs: 1
          <custom_metric_name>: 2
            description: "<custom_metric_description>"
            attributes:
              - key: env
                default_value: unknown 3
# ...
```

1 Specifies attributes for logs.

2 You can specify a custom metric name such as **my.log.count**.

3 Defines a default value when the attribute is not set.

4.5.2. Routing Connector

The Routing Connector routes logs, metrics, and traces to specified pipelines according to resource attributes and their routing conditions, which are written as OpenTelemetry Transformation Language (OTTL) statements.

IMPORTANT

The Routing Connector is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with an enabled Routing Connector

```
# ...
  config:
    connectors:
      routing:
        table: (1)
          - statement: route() where attributes["X-Tenant"] == "dev" (2)
            pipelines: [traces/dev] (3)
          - statement: route() where attributes["X-Tenant"] == "prod"
            pipelines: [traces/prod]
        default_pipelines: [traces/dev] (4)
        error_mode: ignore (5)
        match_once: false (6)
    service:
      pipelines:
        traces/in:
          receivers: [otlp]
          exporters: [routing]
        traces/dev:
          receivers: [routing]
          exporters: [otlp/dev]
        traces/prod:
          receivers: [routing]
          exporters: [otlp/prod]
# ...
```

(1) Connector routing table.

(2) Routing conditions written as OTTL statements.

(3) Destination pipelines for routing the matching telemetry data.

(4) Destination pipelines for routing the telemetry data for which no routing condition is satisfied.

(5) Error-handling mode: The **propagate** value is for logging an error and dropping the payload. The **ignore** value is for ignoring the condition and attempting to match with the next one. The **silent** value is the same as **ignore** but without logging the error. The default is **propagate**.

(6) When set to **true**, the payload is routed only to the first pipeline whose routing condition is met. The default is **false**.

4.5.3. Forward Connector

The Forward Connector merges two pipelines of the same type.

IMPORTANT

The Forward Connector is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with an enabled Forward Connector

```
# ...
  config:
    receivers:
      otlp:
        protocols:
          grpc:
      jaeger:
        protocols:
          grpc:
    processors:
      batch:
    exporters:
      otlp:
        endpoint: tempo-simplest-distributor:4317
        tls:
          insecure: true
    connectors:
      forward: {}
    service:
      pipelines:
        traces/regiona:
          receivers: [otlp]
          processors: []
          exporters: [forward]
        traces/regionb:
          receivers: [jaeger]
          processors: []
          exporters: [forward]
        traces:
          receivers: [forward]
          processors: [batch]
          exporters: [otlp]
# ...
```

4.5.4. Spanmetrics Connector

The Spanmetrics Connector aggregates Request, Error, and Duration (R.E.D) OpenTelemetry metrics from span data.

OpenTelemetry Collector custom resource with an enabled Spanmetrics Connector

```
# ...
  config:
    connectors:
      spanmetrics:
        metrics_flush_interval: 15s (1)
    service:
      pipelines:
        traces:
          exporters: [spanmetrics]
        metrics:
          receivers: [spanmetrics]
# ...
```

(1) Defines the flush interval of the generated metrics. Defaults to **15s**.

4.5.5. Additional resources

- OpenTelemetry Protocol (OTLP) (OpenTelemetry Documentation)

4.6. EXTENSIONS

Extensions add capabilities to the Collector. For example, authentication can be added to the receivers and exporters automatically.

Currently, the following General Availability and Technology Preview extensions are available for the Red Hat build of OpenTelemetry:

- BearerTokenAuth Extension
- OAuth2Client Extension
- File Storage Extension
- OIDC Auth Extension
- Jaeger Remote Sampling Extension
- Performance Profiler Extension
- Health Check Extension
- zPages Extension

4.6.1. BearerTokenAuth Extension

The BearerTokenAuth Extension is an authenticator for receivers and exporters that are based on the HTTP and the gRPC protocol. You can use the OpenTelemetry Collector custom resource to configure client authentication and server authentication for the BearerTokenAuth Extension on the receiver and exporter side. This extension supports traces, metrics, and logs.

OpenTelemetry Collector custom resource with client and server authentication configured for the BearerTokenAuth Extension

```
# ...
  config:
    extensions:
      bearertokenauth:
        scheme: "Bearer" 1
        token: "<token>" 2
        filename: "<token_file>" 3

    receivers:
      otlp:
        protocols:
          http:
            auth:
              authenticator: bearertokenauth 4
    exporters:
      otlp:
        auth:
          authenticator: bearertokenauth 5

    service:
      extensions: [bearertokenauth]
      pipelines:
        traces:
          receivers: [otlp]
          exporters: [otlp]
# ...
```

1. You can configure the BearerTokenAuth Extension to send a custom **scheme**. The default is **Bearer**.
2. You can add the BearerTokenAuth Extension token as metadata to identify a message.
3. Path to a file that contains an authorization token that is transmitted with every message.
4. You can assign the authenticator configuration to an OTLP Receiver.
5. You can assign the authenticator configuration to an OTLP Exporter.

4.6.2. OAuth2Client Extension

The OAuth2Client Extension is an authenticator for exporters that are based on the HTTP and the gRPC protocol. Client authentication for the OAuth2Client Extension is configured in a separate section in the OpenTelemetry Collector custom resource. This extension supports traces, metrics, and logs.

IMPORTANT

The OAuth2Client Extension is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with client authentication configured for the OAuth2Client Extension

```
# ...
  config:
    extensions:
      oauth2client:
        client_id: <client_id> 1
        client_secret: <client_secret> 2
        endpoint_params: 3
          audience: <audience>
        token_url: https://example.com/oauth2/default/v1/token 4
        scopes: ["api.metrics"] 5
        # tls settings for the token client
        tls: 6
          insecure: true 7
          ca_file: /var/lib/mycert.pem 8
          cert_file: <cert_file> 9
          key_file: <key_file> 10
        timeout: 2s 11

    receivers:
      otlp:
        protocols:
          http: {}

    exporters:
      otlp:
        auth:
          authenticator: oauth2client 12

    service:
      extensions: [oauth2client]
      pipelines:
        traces:
          receivers: [otlp]
          exporters: [otlp]
# ...
```

1. Client identifier, which is provided by the identity provider.
2. Confidential key used to authenticate the client to the identity provider.

3. Further metadata, in the key-value pair format, which is transferred during authentication. For example, **audience** specifies the intended audience for the access token, indicating the recipient
4. The URL of the OAuth2 token endpoint, where the Collector requests access tokens.
5. The scopes define the specific permissions or access levels requested by the client.
6. The Transport Layer Security (TLS) settings for the token client, which is used to establish a secure connection when requesting tokens.
7. When set to **true**, configures the Collector to use an insecure or non-verified TLS connection to call the configured token endpoint.
8. The path to a Certificate Authority (CA) file that is used to verify the server's certificate during the TLS handshake.
9. The path to the client certificate file that the client must use to authenticate itself to the OAuth2 server if required.
10. The path to the client's private key file that is used with the client certificate if needed for authentication.
11. Sets a timeout for the token client's request.
12. You can assign the authenticator configuration to an OTLP exporter.

4.6.3. File Storage Extension

The File Storage Extension supports traces, metrics, and logs. This extension can persist the state to the local file system. This extension persists the sending queue for the OpenTelemetry Protocol (OTLP) exporters that are based on the HTTP and the gRPC protocols. This extension requires the read and write access to a directory. This extension can use a default directory, but the default directory must already exist.

IMPORTANT

The File Storage Extension is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with a configured File Storage Extension that persists an OTLP sending queue

```
# ...
  config:
    extensions:
      file_storage/all_settings:
        directory: /var/lib/otelcol/mydir 1
        timeout: 1s 2
```

```
      compaction:
        on_start: true 3
        directory: /tmp/ 4
        max_transaction_size: 65_536 5
      fsync: false 6

  exporters:
    otlp:
      sending_queue:
        storage: file_storage/all_settings 7

  service:
    extensions: [file_storage/all_settings] 8
    pipelines:
      traces:
        receivers: [otlp]
        exporters: [otlp]
# ...
```

1. Specifies the directory in which the telemetry data is stored.
2. Specifies the timeout time interval for opening the stored files.
3. Starts compaction when the Collector starts. If omitted, the default is **false**.
4. Specifies the directory in which the compactor stores the telemetry data.
5. Defines the maximum size of the compaction transaction. To ignore the transaction size, set to zero. If omitted, the default is **65536** bytes.
6. When set, forces the database to perform an **fsync** call after each write operation. This helps to ensure database integrity if there is an interruption to the database process, but at the cost of performance.
7. Buffers the OTLP Exporter data on the local file system.
8. Starts the File Storage Extension by the Collector.

4.6.4. OIDC Auth Extension

The OIDC Auth Extension authenticates incoming requests to receivers by using the OpenID Connect (OIDC) protocol. It validates the ID token in the authorization header against the issuer and updates the authentication context of the incoming request.

IMPORTANT

The OIDC Auth Extension is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with the configured OIDC Auth Extension

```
# ...
  config:
    extensions:
      oidc:
        attribute: authorization 1
        issuer_url: https://example.com/auth/realms/opentelemetry 2
        issuer_ca_path: /var/run/tls/issuer.pem 3
        audience: otel-collector 4
        username_claim: email 5
    receivers:
      otlp:
        protocols:
          grpc:
            auth:
              authenticator: oidc
    exporters:
      debug: {}
    service:
      extensions: [oidc]
      pipelines:
        traces:
          receivers: [otlp]
          exporters: [debug]
# ...
```

1. The name of the header that contains the ID token. The default name is **authorization**.
2. The base URL of the OIDC provider.
3. Optional: The path to the issuer's CA certificate.
4. The audience for the token.
5. The name of the claim that contains the username. The default name is **sub**.

4.6.5. Jaeger Remote Sampling Extension

The Jaeger Remote Sampling Extension enables serving sampling strategies after Jaeger's remote sampling API. You can configure this extension to proxy requests to a backing remote sampling server such as a Jaeger collector down the pipeline or to a static JSON file from the local file system.

IMPORTANT

The Jaeger Remote Sampling Extension is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with a configured Jaeger Remote Sampling Extension

```
# ...
  config:
    extensions:
      jaegerremotesampling:
        source:
          reload_interval: 30s 1
          remote:
            endpoint: jaeger-collector:14250 2
          file: /etc/otelcol/sampling_strategies.json 3

    receivers:
      otlp:
        protocols:
          http: {}

    exporters:
      debug: {}

    service:
      extensions: [jaegerremotesampling]
      pipelines:
        traces:
          receivers: [otlp]
          exporters: [debug]
# ...
```

1. The time interval at which the sampling configuration is updated.
2. The endpoint for reaching the Jaeger remote sampling strategy provider.
3. The path to a local file that contains a sampling strategy configuration in the JSON format.

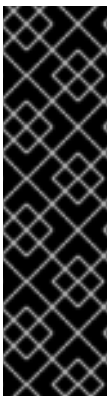
Example of a Jaeger Remote Sampling strategy file

```
{
  "service_strategies": [
    {
      "service": "foo",
      "type": "probabilistic",
```

```
      "param": 0.8,
      "operation_strategies": [
        {
          "operation": "op1",
          "type": "probabilistic",
          "param": 0.2
        },
        {
          "operation": "op2",
          "type": "probabilistic",
          "param": 0.4
        }
      ]
    },
    {
      "service": "bar",
      "type": "ratelimiting",
      "param": 5
    }
  ],
  "default_strategy": {
    "type": "probabilistic",
    "param": 0.5,
    "operation_strategies": [
      {
        "operation": "/health",
        "type": "probabilistic",
        "param": 0.0
      },
      {
        "operation": "/metrics",
        "type": "probabilistic",
        "param": 0.0
      }
    ]
  }
}
```

4.6.6. Performance Profiler Extension

The Performance Profiler Extension enables the Go **net/http/pprof** endpoint. Developers use this extension to collect performance profiles and investigate issues with the service.



IMPORTANT

The Performance Profiler Extension is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with the configured Performance Profiler Extension

```
# ...
  config:
    extensions:
      pprof:
        endpoint: localhost:1777 (1)
        block_profile_fraction: 0 (2)
        mutex_profile_fraction: 0 (3)
        save_to_file: test.pprof (4)

    receivers:
      otlp:
        protocols:
          http: {}

    exporters:
      debug: {}

    service:
      extensions: [pprof]
      pipelines:
        traces:
          receivers: [otlp]
          exporters: [debug]
# ...
```

1. The endpoint at which this extension listens. Use **localhost:** to make it available only locally or **":"** to make it available on all network interfaces. The default value is **localhost:1777**.
2. Sets a fraction of blocking events to be profiled. To disable profiling, set this to **0** or a negative integer. See the documentation for the **runtime** package. The default value is **0**.
3. Set a fraction of mutex contention events to be profiled. To disable profiling, set this to **0** or a negative integer. See the documentation for the **runtime** package. The default value is **0**.
4. The name of the file in which the CPU profile is to be saved. Profiling starts when the Collector starts. Profiling is saved to the file when the Collector is terminated.

4.6.7. Health Check Extension

The Health Check Extension provides an HTTP URL for checking the status of the OpenTelemetry Collector. You can use this extension as a liveness and readiness probe on OpenShift.

IMPORTANT

The Health Check Extension is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OpenTelemetry Collector custom resource with the configured Health Check Extension

```
# ...
  config:
    extensions:
      health_check:
        endpoint: "0.0.0.0:13133" (1)
        tls: (2)
          ca_file: "/path/to/ca.crt"
          cert_file: "/path/to/cert.crt"
          key_file: "/path/to/key.key"
        path: "/health/status" (3)
        check_collector_pipeline: (4)
          enabled: true (5)
          interval: "5m" (6)
          exporter_failure_threshold: 5 (7)

    receivers:
      otlp:
        protocols:
          http: {}

    exporters:
      debug: {}

    service:
      extensions: [health_check]
      pipelines:
        traces:
          receivers: [otlp]
          exporters: [debug]
# ...
```

1. The target IP address for publishing the health check status. The default is **0.0.0.0:13133**.
2. The TLS server-side configuration. Defines paths to TLS certificates. If omitted, the TLS is disabled.
3. The path for the health check server. The default is /.
4. Settings for the Collector pipeline health check.
5. Enables the Collector pipeline health check. The default is **false**.

6 The time interval for checking the number of failures. The default is **5m**.

7 The threshold of multiple failures until which a container is still marked as healthy. The default is **5**.

4.6.8. zPages Extension

The zPages Extension provides an HTTP endpoint that serves live data for debugging instrumented components in real time. You can use this extension for in-process diagnostics and insights into traces and metrics without relying on an external backend. With this extension, you can monitor and troubleshoot the behavior of the OpenTelemetry Collector and related components by watching the diagnostic information at the provided endpoint.

IMPORTANT

The zPages Extension is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

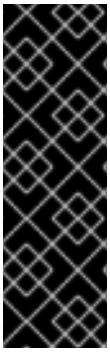
OpenTelemetry Collector custom resource with the configured zPages Extension

```
# ...
  config:
    extensions:
      zpages:
        endpoint: "localhost:55679" 1

    receivers:
      otlp:
        protocols:
          http: {}
    exporters:
      debug: {}

    service:
      extensions: [zpages]
      pipelines:
        traces:
          receivers: [otlp]
          exporters: [debug]
# ...
```

1 Specifies the HTTP endpoint for serving the zPages extension. The default is **localhost:55679**.



IMPORTANT

Accessing the HTTP endpoint requires port-forwarding because the Red Hat build of OpenTelemetry Operator does not expose this route.

You can enable port-forwarding by running the following **oc** command:

```
$ oc port-forward pod/$(oc get pod -l app.kubernetes.io/name=instance-collector -
o=jsonpath='{.items[0].metadata.name}') 55679
```

The Collector provides the following zPages for diagnostics:

ServiceZ

Shows an overview of the Collector services and links to the following zPages: **PipelineZ**, **ExtensionZ**, and **FeatureZ**. This page also displays information about the build version and runtime. An example of this page's URL is **http://localhost:55679/debug/servicez**.

PipelineZ

Shows detailed information about the active pipelines in the Collector. This page displays the pipeline type, whether data are modified, and the associated receivers, processors, and exporters for each pipeline. An example of this page's URL is **http://localhost:55679/debug/pipelinez**.

ExtensionZ

Shows the currently active extensions in the Collector. An example of this page's URL is **http://localhost:55679/debug/extensionz**.

FeatureZ

Shows the feature gates enabled in the Collector along with their status and description. An example of this page's URL is **http://localhost:55679/debug/featurez**.

TraceZ

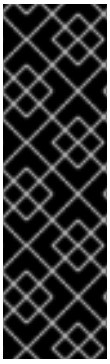
Shows spans categorized by latency. Available time ranges include 0 µs, 10 µs, 100 µs, 1 ms, 10 ms, 100 ms, 1 s, 10 s, 1 m. This page also allows for quick inspection of error samples. An example of this page's URL is **http://localhost:55679/debug/tracez**.

4.6.9. Additional resources

- OpenTelemetry Protocol (OTLP) (OpenTelemetry Documentation)

4.7. TARGET ALLOCATOR

The Target Allocator is an optional component of the OpenTelemetry Operator that shards scrape targets across the deployed fleet of OpenTelemetry Collector instances. The Target Allocator integrates with the Prometheus **PodMonitor** and **ServiceMonitor** custom resources (CR). When the Target Allocator is enabled, the OpenTelemetry Operator adds the **http_sd_config** field to the enabled **prometheus** receiver that connects to the Target Allocator service.



IMPORTANT

The Target Allocator is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

Example OpenTelemetryCollector CR with the enabled Target Allocator

```
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: observability
spec:
  mode: statefulset (1)
  targetAllocator:
    enabled: true (2)
    serviceAccount: (3)
    prometheusCR:
      enabled: true (4)
      scrapeInterval: 10s
      serviceMonitorSelector: (5)
        name: app1
      podMonitorSelector: (6)
        name: app2
  config:
    receivers:
      prometheus: (7)
        config:
          scrape_configs: []
    processors:
    exporters:
      debug: {}
    service:
      pipelines:
        metrics:
          receivers: [prometheus]
          processors: []
          exporters: [debug]
# ...
```

(1) When the Target Allocator is enabled, the deployment mode must be set to **statefulset**.

(2) Enables the Target Allocator. Defaults to **false**.

(3) The service account name of the Target Allocator deployment. The service account needs to have RBAC to get the **ServiceMonitor**, **PodMonitor** custom resources, and other objects from the cluster to properly set labels on scraped metrics. The default service name is **<collector_name>-targetallocator**.

4 Enables integration with the Prometheus **PodMonitor** and **ServiceMonitor** custom resources.

5 Label selector for the Prometheus **ServiceMonitor** custom resources. When left empty, enables all service monitors.

6 Label selector for the Prometheus **PodMonitor** custom resources. When left empty, enables all pod monitors.

7 Prometheus receiver with the minimal, empty **scrape_config: []** configuration option.

The Target Allocator deployment uses the Kubernetes API to get relevant objects from the cluster, so it requires a custom RBAC configuration.

RBAC configuration for the Target Allocator service account

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otel-targetallocator
rules:
  - apiGroups: [""]
    resources:
      - services
      - pods
      - namespaces
    verbs: ["get", "list", "watch"]
  - apiGroups: ["monitoring.coreos.com"]
    resources:
      - servicemonitors
      - podmonitors
      - scrapeconfigs
      - probes
    verbs: ["get", "list", "watch"]
  - apiGroups: ["discovery.k8s.io"]
    resources:
      - endpointslices
    verbs: ["get", "list", "watch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-targetallocator
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: otel-targetallocator
subjects:
  - kind: ServiceAccount
    name: otel-targetallocator 1
    namespace: observability 2
# ...
```

1 The name of the Target Allocator service account mane.

2 The namespace of the Target Allocator service account.

CHAPTER 5. CONFIGURING THE INSTRUMENTATION

The Red Hat build of OpenTelemetry Operator uses an **Instrumentation** custom resource that defines the configuration of the instrumentation.

5.1. AUTO-INSTRUMENTATION IN THE RED HAT BUILD OF OPENTELEMETRY OPERATOR

Auto-instrumentation in the Red Hat build of OpenTelemetry Operator can automatically instrument an application without manual code changes. Developers and administrators can monitor applications with minimal effort and changes to the existing codebase.

Auto-instrumentation runs as follows:

1. The Red Hat build of OpenTelemetry Operator injects an init-container, or a sidecar container for Go, to add the instrumentation libraries for the programming language of the instrumented application.

2. The Red Hat build of OpenTelemetry Operator sets the required environment variables in the application's runtime environment. These variables configure the auto-instrumentation libraries to collect traces, metrics, and logs and send them to the appropriate OpenTelemetry Collector or another telemetry backend.

3. The injected libraries automatically instrument your application by connecting to known frameworks and libraries, such as web servers or database clients, to collect telemetry data. The source code of the instrumented application is not modified.

4. Once the application is running with the injected instrumentation, the application automatically generates telemetry data, which is sent to a designated OpenTelemetry Collector or an external OTLP endpoint for further processing.

Auto-instrumentation enables you to start collecting telemetry data quickly without having to manually integrate the OpenTelemetry SDK into your application code. However, some applications might require specific configurations or custom manual instrumentation.

5.2. OPENTELEMETRY INSTRUMENTATION CONFIGURATION OPTIONS

The Red Hat build of OpenTelemetry injects and configures the OpenTelemetry auto-instrumentation libraries into your workloads. Currently, the Red Hat build of OpenTelemetry supports injecting instrumentation libraries for Go, Java, Node.js, Python, .NET, and the Apache HTTP Server (**httpd**).

IMPORTANT

The Red Hat build of OpenTelemetry Operator only supports the injection mechanism of the instrumentation libraries but does not support instrumentation libraries or upstream images. Customers can build their own instrumentation images or use community images.

5.2.1. Instrumentation options

Instrumentation options are specified in an **Instrumentation** custom resource (CR).

Sample Instrumentation CR

```
apiVersion: opentelemetry.io/v1alpha1
kind: Instrumentation
metadata:
  name: instrumentation
spec:
  env:
    - name: OTEL_EXPORTER_OTLP_TIMEOUT
      value: "20"
  exporter:
    endpoint: http://production-collector.observability.svc.cluster.local:4317
  propagators:
    - tracecontext
    - baggage
  sampler:
    type: parentbased_traceidratio
    argument: "1"
  python: 1
    env: 2
      - name: OTEL_EXPORTER_OTLP_ENDPOINT
        value: http://production-collector.observability.svc.cluster.local:4318
  dotnet: 3
    env: 4
      - name: OTEL_EXPORTER_OTLP_ENDPOINT
        value: http://production-collector.observability.svc.cluster.local:4318
  go: 5
    env: 6
      - name: OTEL_EXPORTER_OTLP_ENDPOINT
        value: http://production-collector.observability.svc.cluster.local:4318
```

1. Python auto-instrumentation uses protocol buffers over HTTP (HTTP/proto or HTTP/protobuf) by default.
2. Required if endpoint is set to **:4317**.
3. .NET auto-instrumentation uses protocol buffers over HTTP (HTTP/proto or HTTP/protobuf) by default.
4. Required if endpoint is set to **:4317**.
5. Go auto-instrumentation uses protocol buffers over HTTP (HTTP/proto or HTTP/protobuf) by default.
6. Required if endpoint is set to **:4317**.

For more information about procol buffers, see Overview (Protocol Buffers Documentation).

Table 5.1. Parameters used by the Operator to define the instrumentation

Parameter	Description	Values
env	Definition of common environment variables for all instrumentation types.	

Parameter	Description	Values
exporter	Exporter configuration.	
propagators	Propagators defines inter-process context propagation configuration.	**tracecontext**, **baggage**, **b3**, **b3multi**, **jaeger**, **ottrace**, **none**
resource	Resource attributes configuration.	
sampler	Sampling configuration.	
apacheHttpd	Configuration for the Apache HTTP Server instrumentation.	
dotnet	Configuration for the .NET instrumentation.	
go	Configuration for the Go instrumentation.	
java	Configuration for the Java instrumentation.	
nodejs	Configuration for the Node.js instrumentation.	
python	Configuration for the Python instrumentation.	Depending on the programming language, environment variables might not work for configuring telemetry. For the SDKs that do not support environment variable configuration, you must add a similar configuration directly in the code. For more information, see Environment Variable Specification (OpenTelemetry Documentation).

Table 5.2. Default protocol for auto-instrumentation

Auto-instrumentation	Default protocol
Java 1.x	**otlp/grpc**
Java 2.x	**otlp/http**
Python	**otlp/http**

Auto-instrumentation	Default protocol
.NET	**otlp/http**
Go	**otlp/http**
Apache HTTP Server	**otlp/grpc**

5.2.2. Configuration of the OpenTelemetry SDK variables

You can use the **instrumentation.opentelemetry.io/inject-sdk** annotation in the OpenTelemetry Collector custom resource to instruct the Red Hat build of OpenTelemetry Operator to inject some of the following OpenTelemetry SDK environment variables, depending on the **Instrumentation** CR, into your pod:

- **OTEL_SERVICE_NAME**
- **OTEL_TRACES_SAMPLER**
- **OTEL_TRACES_SAMPLER_ARG**
- **OTEL_PROPAGATORS**
- **OTEL_RESOURCE_ATTRIBUTES**
- **OTEL_EXPORTER_OTLP_ENDPOINT**
- **OTEL_EXPORTER_OTLP_CERTIFICATE**
- **OTEL_EXPORTER_OTLP_CLIENT_CERTIFICATE**
- **OTEL_EXPORTER_OTLP_CLIENT_KEY**

Table 5.3. Values for the **instrumentation.opentelemetry.io/inject-sdk** annotation

Value	Description
"true"	Injects the **Instrumentation** resource with the default name from the current namespace.
"false"	Injects no **Instrumentation** resource.
"<instrumentation_name>"	Specifies the name of the **Instrumentation** resource to inject from the current namespace.
"<namespace>/<instrumentation_name>"	Specifies the name of the **Instrumentation** resource to inject from another namespace.

5.2.3. Exporter configuration

Although the **Instrumentation** custom resource supports setting up one or more exporters per signal, auto-instrumentation configures only the OTLP Exporter. So you must configure the endpoint to point to the OTLP Receiver on the Collector.

Sample exporter TLS CA configuration using a config map

```
apiVersion: opentelemetry.io/v1alpha1
kind: Instrumentation
# ...
spec
# ...
  exporter:
    endpoint: https://production-collector.observability.svc.cluster.local:4317 (1)
    tls:
      configMapName: ca-bundle (2)
      ca_file: service-ca.crt (3)
# ...
```

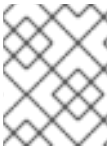
1. Specifies the OTLP endpoint using the HTTPS scheme and TLS.
2. Specifies the name of the config map. The config map must already exist in the namespace of the pod injecting the auto-instrumentation.
3. Points to the CA certificate in the config map or the absolute path to the certificate if the certificate is already present in the workload file system.

Sample exporter mTLS configuration using a Secret

```
apiVersion: opentelemetry.io/v1alpha1
kind: Instrumentation
# ...
spec
# ...
  exporter:
    endpoint: https://production-collector.observability.svc.cluster.local:4317 (1)
    tls:
      secretName: serving-certs (2)
      ca_file: service-ca.crt (3)
      cert_file: tls.crt (4)
      key_file: tls.key (5)
# ...
```

1. Specifies the OTLP endpoint using the HTTPS scheme and TLS.
2. Specifies the name of the Secret for the **ca_file**, **cert_file**, and **key_file** values. The Secret must already exist in the namespace of the pod injecting the auto-instrumentation.
3. Points to the CA certificate in the Secret or the absolute path to the certificate if the certificate is already present in the workload file system.
4. Points to the client certificate in the Secret or the absolute path to the certificate if the certificate is already present in the workload file system.

5 Points to the client key in the Secret or the absolute path to a key if the key is already present in the workload file system.



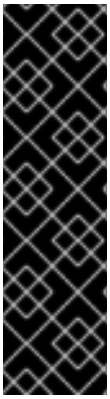
NOTE

You can provide the CA certificate in a config map or Secret. If you provide it in both, the config map takes higher precedence than the Secret.

Example configuration for CA bundle injection by using a config map and Instrumentation CR

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: otelcol-cabundle
  namespace: tutorial-application
  annotations:
    service.beta.openshift.io/inject-cabundle: "true"
# ...
---
apiVersion: opentelemetry.io/v1alpha1
kind: Instrumentation
metadata:
  name: my-instrumentation
spec:
  exporter:
    endpoint: https://simplest-collector.tracing-system.svc.cluster.local:4317
    tls:
      configMapName: otelcol-cabundle
      ca: service-ca.crt
# ...
```

5.2.4. Configuration of the Apache HTTP Server auto-instrumentation



IMPORTANT

The Apache HTTP Server auto-instrumentation is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

Table 5.4. Parameters for the.spec.apacheHttpd field

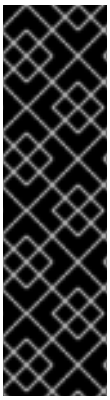
Name	Description	Default

Name	Description	Default
attrs	Attributes specific to the Apache HTTP Server.	
configPath	Location of the Apache HTTP Server configuration.	**/usr/local/apache2/conf**
env	Environment variables specific to the Apache HTTP Server.	
image	Container image with the Apache SDK and auto-instrumentation.	
resourceRequirements	The compute resource requirements.	
version	Apache HTTP Server version.	**2.4**

The PodSpec annotation to enable injection

```
instrumentation.opentelemetry.io/inject-apache-httpd: "true"
```

5.2.5. Configuration of the .NET auto-instrumentation



IMPORTANT

The .NET auto-instrumentation is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

IMPORTANT

By default, this feature injects unsupported, upstream instrumentation libraries.

Name	Description
env	Environment variables specific to .NET.
image	Container image with the .NET SDK and auto-instrumentation.

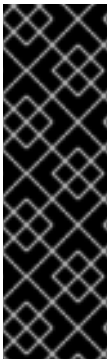
Name	Description
resourceRequirements	The compute resource requirements.

For the .NET auto-instrumentation, the required **OTEL_EXPORTER_OTLP_ENDPOINT** environment variable must be set if the endpoint of the exporters is set to **4317**. The .NET autoinstrumentation uses **http/proto** by default, and the telemetry data must be set to the **4318** port.

The PodSpec annotation to enable injection

```
instrumentation.opentelemetry.io/inject-dotnet: "true"
```

5.2.6. Configuration of the Go auto-instrumentation



IMPORTANT

The Go auto-instrumentation is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

IMPORTANT

By default, this feature injects unsupported, upstream instrumentation libraries.

Name	Description
env	Environment variables specific to Go.
image	Container image with the Go SDK and auto-instrumentation.
resourceRequirements	The compute resource requirements.

The PodSpec annotation to enable injection

```
instrumentation.opentelemetry.io/inject-go: "true"
instrumentation.opentelemetry.io/otel-go-auto-target-exe: "/<path>/<to>/<container>/<executable>"
```
(1)

(1) Sets the value for the required **OTEL_GO_AUTO_TARGET_EXE** environment variable.

Permissions required for the Go auto-instrumentation in the OpenShift cluster

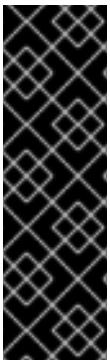
```
apiVersion: security.openshift.io/v1
kind: SecurityContextConstraints
metadata:
  name: otel-go-instrumentation-scc
allowHostDirVolumePlugin: true
allowPrivilegeEscalation: true
allowPrivilegedContainer: true
allowedCapabilities:
- "SYS_PTRACE"
fsGroup:
  type: RunAsAny
runAsUser:
  type: RunAsAny
seLinuxContext:
  type: RunAsAny
seccompProfiles:
- '*'
supplementalGroups:
  type: RunAsAny
```

TIP

The CLI command for applying the permissions for the Go auto-instrumentation in the OpenShift cluster is as follows:

```
$ oc adm policy add-scc-to-user otel-go-instrumentation-scc -z <service_account>
```

5.2.7. Configuration of the Java auto-instrumentation



IMPORTANT

The Java auto-instrumentation is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

IMPORTANT

By default, this feature injects unsupported, upstream instrumentation libraries.

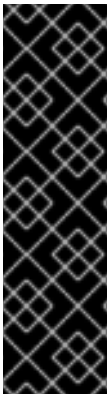
Name	Description
env	Environment variables specific to Java.
image	Container image with the Java SDK and auto-instrumentation.

Name	Description
resourceRequirements	The compute resource requirements.

The PodSpec annotation to enable injection

```
instrumentation.opentelemetry.io/inject-java: "true"
```

5.2.8. Configuration of the Node.js auto-instrumentation



IMPORTANT

The Node.js auto-instrumentation is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

IMPORTANT

By default, this feature injects unsupported, upstream instrumentation libraries.

Name	Description
env	Environment variables specific to Node.js.
image	Container image with the Node.js SDK and auto-instrumentation.
resourceRequirements	The compute resource requirements.

The PodSpec annotations to enable injection

```
instrumentation.opentelemetry.io/inject-nodejs: "true"
```

5.2.9. Configuration of the Python auto-instrumentation

IMPORTANT

The Python auto-instrumentation is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

IMPORTANT

By default, this feature injects unsupported, upstream instrumentation libraries.

Name	Description
env	Environment variables specific to Python.
image	Container image with the Python SDK and auto-instrumentation.
resourceRequirements	The compute resource requirements.

For Python auto-instrumentation, the **OTEL_EXPORTER_OTLP_ENDPOINT** environment variable must be set if the endpoint of the exporters is set to **4317**. Python auto-instrumentation uses **http/proto** by default, and the telemetry data must be set to the **4318** port.

The PodSpec annotation to enable injection

```
instrumentation.opentelemetry.io/inject-python: "true"
```

5.2.10. Multi-container pods

The instrumentation is injected to the first container that is available by default according to the pod specification. You can also specify the target container names for injection.

Pod annotation

```
instrumentation.opentelemetry.io/container-names: "<container_1>,<container_2>" 1
```

1 Use this annotation when you want to inject a single instrumentation in multiple containers.

NOTE

The Go auto-instrumentation does not support multi-container auto-instrumentation injection.

5.2.11. Multi-container pods with multiple instrumentations

Injecting instrumentation for an application language to one or more containers in a multi-container pod requires the following annotation:

```
instrumentation.opentelemetry.io/<application_language>-container-names: "<container_1>,<container_2>" 1
```

1 You can inject instrumentation for only one language per container. For the list of supported **<application_language>** values, see the following table.

Table 5.5. Supported values for the<application_language>

Language	Value for **<application_language>**
ApacheHTTPD	**apache-httpd**
DotNet	**dotnet**
Java	**java**
NGINX	**inject-nginx**
NodeJS	**nodejs**
Python	**python**
SDK	**sdk**

5.2.12. Using the instrumentation CR with Service Mesh

When using the **Instrumentation** custom resource (CR) with Red Hat OpenShift Service Mesh, you must use the **b3multi** propagator.

CHAPTER 6. SENDING TRACES, LOGS, AND METRICS TO THE OPENTELEMETRY COLLECTOR

You can set up and use the Red Hat build of OpenTelemetry to send traces, logs, and metrics to the OpenTelemetry Collector or the **TempoStack** instance.

Sending traces and metrics to the OpenTelemetry Collector is possible with or without sidecar injection.

6.1. SENDING TRACES AND METRICS TO THE OPENTELEMETRY COLLECTOR WITH SIDECAR INJECTION

You can set up sending telemetry data to an OpenTelemetry Collector instance with sidecar injection.

The Red Hat build of OpenTelemetry Operator allows sidecar injection into deployment workloads and automatic configuration of your instrumentation to send telemetry data to the OpenTelemetry Collector.

Prerequisites

- The Red Hat OpenShift Distributed Tracing Platform is installed, and a TempoStack instance is deployed.
- You have access to the cluster through the web console or the OpenShift CLI (**oc**):
  - You are logged in to the web console as a cluster administrator with the **cluster-admin** role.
  - An active OpenShift CLI (**oc**) session by a cluster administrator with the **cluster-admin** role.
  - For Red Hat OpenShift Dedicated, you must have an account with the **dedicated-admin** role.

Procedure

1. Create a project for an OpenTelemetry Collector instance.

```
apiVersion: project.openshift.io/v1
kind: Project
metadata:
  name: observability
```

2. Create a service account.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: otel-collector-sidecar
  namespace: observability
```

3. Grant the permissions to the service account for the **k8sattributes** and **resourcedetection** processors.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
```

```
metadata:
  name: otel-collector
rules:
- apiGroups: ["", "config.openshift.io"]
  resources: ["pods", "namespaces", "infrastructures", "infrastructures/status"]
  verbs: ["get", "watch", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-collector
subjects:
- kind: ServiceAccount
  name: otel-collector-sidecar
  namespace: observability
roleRef:
  kind: ClusterRole
  name: otel-collector
  apiGroup: rbac.authorization.k8s.io
```

4. Deploy the OpenTelemetry Collector as a sidecar.

```
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: observability
spec:
  serviceAccount: otel-collector-sidecar
  mode: sidecar
  config:
    serviceAccount: otel-collector-sidecar
    receivers:
      otlp:
        protocols:
          grpc: {}
          http: {}
    processors:
      batch: {}
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
      resourcedetection:
        detectors: [openshift]
        timeout: 2s
    exporters:
      otlp:
        endpoint: "tempo-<example>-gateway:8090" (1)
        tls:
          insecure: true
    service:
      pipelines:
        traces:
```

```
    receivers: [otlp]
    processors: [memory_limiter, resourcedetection, batch]
    exporters: [otlp]
```

1. This points to the Gateway of the TempoStack instance deployed by using the **<example>** Tempo Operator.

5. Create your deployment using the **otel-collector-sidecar** service account.
6. Add the **sidecar.opentelemetry.io/inject: "true"** annotation to your **Deployment** object. This will inject all the needed environment variables to send data from your workloads to the OpenTelemetry Collector instance.

6.2. SENDING TRACES AND METRICS TO THE OPENTELEMETRY COLLECTOR WITHOUT SIDECAR INJECTION

You can set up sending telemetry data to an OpenTelemetry Collector instance without sidecar injection, which involves manually setting several environment variables.

Prerequisites

- The Red Hat OpenShift Distributed Tracing Platform is installed, and a TempoStack instance is deployed.
- You have access to the cluster through the web console or the OpenShift CLI (**oc**):
  - You are logged in to the web console as a cluster administrator with the **cluster-admin** role.
  - An active OpenShift CLI (**oc**) session by a cluster administrator with the **cluster-admin** role.
  - For Red Hat OpenShift Dedicated, you must have an account with the **dedicated-admin** role.

Procedure

1. Create a project for an OpenTelemetry Collector instance.

```
apiVersion: project.openshift.io/v1
kind: Project
metadata:
  name: observability
```

2. Create a service account.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: otel-collector-deployment
  namespace: observability
```

3. Grant the permissions to the service account for the **k8sattributes** and **resourcedetection** processors.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otel-collector
rules:
- apiGroups: ["", "config.openshift.io"]
  resources: ["pods", "namespaces", "infrastructures", "infrastructures/status"]
  verbs: ["get", "watch", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-collector
subjects:
- kind: ServiceAccount
  name: otel-collector-deployment
  namespace: observability
roleRef:
  kind: ClusterRole
  name: otel-collector
  apiGroup: rbac.authorization.k8s.io
```

4. Deploy the OpenTelemetry Collector instance with the **OpenTelemetryCollector** custom resource.

```
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: observability
spec:
  mode: deployment
  serviceAccount: otel-collector-deployment
  config:
    receivers:
      jaeger:
        protocols:
          grpc: {}
          thrift_binary: {}
          thrift_compact: {}
          thrift_http: {}
      opencensus:
      otlp:
        protocols:
          grpc: {}
          http: {}
      zipkin: {}
    processors:
      batch: {}
      k8sattributes: {}
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
      resourcedetection:
```

```
    detectors: [openshift]
exporters:
  otlp:
    endpoint: "tempo-<example>-distributor:4317" (1)
    tls:
      insecure: true
service:
  pipelines:
    traces:
      receivers: [jaeger, opencensus, otlp, zipkin]
      processors: [memory_limiter, k8sattributes, resourcedetection, batch]
      exporters: [otlp]
```

(1) This points to the Gateway of the TempoStack instance deployed by using the **<example>** Tempo Operator.

5. Set the environment variables in the container with your instrumented application.

Name	Description	Default value
OTEL_SERVICE_NAME	Sets the value of the **service.name** resource attribute.	**""**
OTEL_EXPORTER_OTLP_ENDPOINT	Base endpoint URL for any signal type with an optionally specified port number.	**https://localhost:4317**
OTEL_EXPORTER_OTLP_CERTIFICATE	Path to the certificate file for the TLS credentials of the gRPC client.	**https://localhost:4317**
OTEL_TRACES_SAMPLER	Sampler to be used for traces.	**parentbased_always_on**
OTEL_EXPORTER_OTLP_PROTOCOL	Transport protocol for the OTLP exporter.	**grpc**
OTEL_EXPORTER_OTLP_TIMEOUT	Maximum time interval for the OTLP exporter to wait for each batch export.	**10s**
OTEL_EXPORTER_OTLP_INSECURE	Disables client transport security for gRPC requests. An HTTPS schema overrides it.	**False**

CHAPTER 7. CONFIGURING METRICS FOR THE MONITORING STACK

As a cluster administrator, you can configure the OpenTelemetry Collector custom resource (CR) to perform the following tasks:

- Create a Prometheus **ServiceMonitor** CR for scraping the Collector's pipeline metrics and the enabled Prometheus exporters.
- Configure the Prometheus receiver to scrape metrics from the in-cluster monitoring stack.

7.1. CONFIGURATION FOR SENDING METRICS TO THE MONITORING STACK

You can configure the **OpenTelemetryCollector** custom resource (CR) to create a Prometheus **ServiceMonitor** CR or a **PodMonitor** CR for a sidecar deployment. A **ServiceMonitor** can scrape Collector's internal metrics endpoint and Prometheus exporter metrics endpoints.

Example of the OpenTelemetry Collector CR with the Prometheus exporter

```
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
spec:
  mode: deployment
  observability:
    metrics:
      enableMetrics: true (1)
  config:
    exporters:
      prometheus:
        endpoint: 0.0.0.0:8889
        resource_to_telemetry_conversion:
          enabled: true # by default resource attributes are dropped
    service:
      telemetry:
        metrics:
          readers:
          - pull:
              exporter:
                prometheus:
                  host: 0.0.0.0
                  port: 8888
      pipelines:
        metrics:
          exporters: [prometheus]
```

(1) Configures the Red Hat build of OpenTelemetry Operator to create the Prometheus **ServiceMonitor** CR or **PodMonitor** CR to scrape the Collector's internal metrics endpoint and the Prometheus exporter metrics endpoints.

NOTE

Setting **enableMetrics** to **true** creates the following two **ServiceMonitor** instances:

- One **ServiceMonitor** instance for the **<instance_name>-collector-monitoring** service. This **ServiceMonitor** instance scrapes the Collector's internal metrics.
- One **ServiceMonitor** instance for the **<instance_name>-collector** service. This **ServiceMonitor** instance scrapes the metrics exposed by the Prometheus exporter instances.

Alternatively, a manually created Prometheus **PodMonitor** CR can provide fine control, for example removing duplicated labels added during Prometheus scraping.

Example of the PodMonitor CR that configures the monitoring stack to scrape the Collector metrics

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: otel-collector
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: <cr_name>-collector 1
  podMetricsEndpoints:
  - port: metrics 2
  - port: promexporter 3
    relabelings:
    - action: labeldrop
      regex: pod
    - action: labeldrop
      regex: container
    - action: labeldrop
      regex: endpoint
    metricRelabelings:
    - action: labeldrop
      regex: instance
    - action: labeldrop
      regex: job
```

1. The name of the OpenTelemetry Collector CR.
2. The name of the internal metrics port for the OpenTelemetry Collector. This port name is always **metrics**.
3. The name of the Prometheus exporter port for the OpenTelemetry Collector.

7.2. CONFIGURATION FOR RECEIVING METRICS FROM THE MONITORING STACK

A configured OpenTelemetry Collector custom resource (CR) can set up the Prometheus receiver to scrape metrics from the in-cluster monitoring stack.

Example of the OpenTelemetry Collector CR for scraping metrics from the in-cluster monitoring stack

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-collector
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-monitoring-view (1)
subjects:
  - kind: ServiceAccount
    name: otel-collector
    namespace: observability
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: cabundle
  namespace: observability
  annotations:
    service.beta.openshift.io/inject-cabundle: "true" (2)
---
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: observability
spec:
  volumeMounts:
    - name: cabundle-volume
      mountPath: /etc/pki/ca-trust/source/service-ca
      readOnly: true
  volumes:
    - name: cabundle-volume
      configMap:
        name: cabundle
  mode: deployment
  config:
    receivers:
      prometheus: (3)
        config:
          scrape_configs:
            - job_name: 'federate'
              scrape_interval: 15s
              scheme: https
              tls_config:
                ca_file: /etc/pki/ca-trust/source/service-ca/service-ca.crt
              bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
              honor_labels: false
              params:
                'match[]':
                  - '{__name__="<metric_name>"}' (4)
              metrics_path: '/federate'
```

```
        static_configs:
          - targets:
            - "prometheus-k8s.openshift-monitoring.svc.cluster.local:9091"
exporters:
  debug: 5
    verbosity: detailed
service:
  pipelines:
    metrics:
      receivers: [prometheus]
      processors: []
      exporters: [debug]
```

1 Assigns the **cluster-monitoring-view** cluster role to the service account of the OpenTelemetry Collector so that it can access the metrics data.

2 Injects the OpenShift service CA for configuring the TLS in the Prometheus receiver.

3 Configures the Prometheus receiver to scrape the federate endpoint from the in-cluster monitoring stack.

4 Uses the Prometheus query language to select the metrics to be scraped. See the in-cluster monitoring documentation for more details and limitations of the federate endpoint.

5 Configures the debug exporter to print the metrics to the standard output.

7.3. ADDITIONAL RESOURCES

- Querying metrics by using the federation endpoint for Prometheus

CHAPTER 8. FORWARDING TELEMETRY DATA

You can use the OpenTelemetry Collector to forward your telemetry data.

8.1. FORWARDING TRACES TO A TEMPOSTACK INSTANCE

To configure forwarding traces to a TempoStack instance, you can deploy and configure the OpenTelemetry Collector. You can deploy the OpenTelemetry Collector in the deployment mode by using the specified processors, receivers, and exporters. For other modes, see the OpenTelemetry Collector documentation linked in *Additional resources*.

Prerequisites

- The Red Hat build of OpenTelemetry Operator is installed.
- The Tempo Operator is installed.
- A TempoStack instance is deployed on the cluster.

Procedure

1. Create a service account for the OpenTelemetry Collector.

   #### Example ServiceAccount

   ```
   apiVersion: v1
   kind: ServiceAccount
   metadata:
     name: otel-collector-deployment
   ```

2. Create a cluster role for the service account.

   #### Example ClusterRole

   ```
   apiVersion: rbac.authorization.k8s.io/v1
   kind: ClusterRole
   metadata:
     name: otel-collector
   rules:
   - apiGroups: [""]
     resources: ["pods", "namespaces",]
     verbs: ["get", "watch", "list"] 1
   - apiGroups: ["apps"]
     resources: ["replicasets"]
     verbs: ["get", "watch", "list"] 2
   - apiGroups: ["config.openshift.io"]
     resources: ["infrastructures", "infrastructures/status"]
     verbs: ["get", "watch", "list"] 3
   ```

   1 This example uses the Kubernetes Attributes Processor, which requires these permissions for the **pods** and **namespaces** resources.

   2

Also due to the Kubernetes Attributes Processor, these permissions are required for the **replicasets** resources.

3 This example also uses the Resource Detection Processor, which requires these permissions for the **infrastructures** and **status** resources.

3. Bind the cluster role to the service account.

Example ClusterRoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-collector
subjects:
- kind: ServiceAccount
  name: otel-collector-deployment
  namespace: otel-collector-example
roleRef:
  kind: ClusterRole
  name: otel-collector
  apiGroup: rbac.authorization.k8s.io
```

4. Create the YAML file to define the **OpenTelemetryCollector** custom resource (CR).

Example OpenTelemetryCollector

```
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
metadata:
  name: otel
spec:
  mode: deployment
  serviceAccount: otel-collector-deployment
  config:
    receivers:
      jaeger:
        protocols:
          grpc: {}
          thrift_binary: {}
          thrift_compact: {}
          thrift_http: {}
      opencensus: {}
      otlp:
        protocols:
          grpc: {}
          http: {}
      zipkin: {}
    processors:
      batch: {}
      k8sattributes: {}
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
```

```
    resourcedetection:
      detectors: [openshift]
  exporters:
    otlp:
      endpoint: "tempo-simplest-distributor:4317" (1)
      tls:
        insecure: true
  service:
    pipelines:
      traces:
        receivers: [jaeger, opencensus, otlp, zipkin] (2)
        processors: [memory_limiter, k8sattributes, resourcedetection, batch]
        exporters: [otlp]
```

1. The Collector exporter is configured to export OTLP and points to the Tempo distributor endpoint, **"tempo-simplest-distributor:4317"** in this example, which is already created.
2. The Collector is configured with a receiver for Jaeger traces, OpenCensus traces over the OpenCensus protocol, Zipkin traces over the Zipkin protocol, and OTLP traces over the gRPC protocol.

TIP

You can deploy **telemetrygen** as a test:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: telemetrygen
spec:
  template:
    spec:
      containers:
        - name: telemetrygen
          image: ghcr.io/open-telemetry/opentelemetry-collector-contrib/telemetrygen:latest
          args:
            - traces
            - --otlp-endpoint=otel-collector:4317
            - --otlp-insecure
            - --duration=30s
            - --workers=1
      restartPolicy: Never
  backoffLimit: 4
```

Additional resources

- OpenTelemetry Collector (OpenTelemetry Documentation)
- Deployment examples on GitHub (GitHub)

8.2. FORWARDING LOGS TO A LOKISTACK INSTANCE

You can deploy the OpenTelemetry Collector to forward logs to a **LokiStack** instance by using the **openshift-logging** tenants mode.

Prerequisites

- The Red Hat build of OpenTelemetry Operator is installed.
- The Loki Operator is installed.
- A supported **LokiStack** instance is deployed on the cluster. For more information about the supported **LokiStack** configuration, see *Logging*.

Procedure

1. Create a service account for the OpenTelemetry Collector.

   Example ServiceAccount object

   ```
   apiVersion: v1
   kind: ServiceAccount
   metadata:
     name: otel-collector-deployment
     namespace: openshift-logging
   ```

2. Create a cluster role that grants the Collector's service account the permissions to push logs to the **LokiStack** application tenant.

   Example ClusterRole object

   ```
   apiVersion: rbac.authorization.k8s.io/v1
   kind: ClusterRole
   metadata:
     name: otel-collector-logs-writer
   rules:
    - apiGroups: ["loki.grafana.com"]
      resourceNames: ["logs"]
      resources: ["application"]
      verbs: ["create"]
    - apiGroups: [""]
      resources: ["pods", "namespaces", "nodes"]
      verbs: ["get", "watch", "list"]
    - apiGroups: ["apps"]
      resources: ["replicasets"]
      verbs: ["get", "list", "watch"]
    - apiGroups: ["extensions"]
      resources: ["replicasets"]
      verbs: ["get", "list", "watch"]
   ```

3. Bind the cluster role to the service account.

   Example ClusterRoleBinding object

   ```
   apiVersion: rbac.authorization.k8s.io/v1
   kind: ClusterRoleBinding
   ```

```
metadata:
  name: otel-collector-logs-writer
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: otel-collector-logs-writer
subjects:
  - kind: ServiceAccount
    name: otel-collector-deployment
    namespace: openshift-logging
```

4. Create an **OpenTelemetryCollector** custom resource (CR) object.

Example OpenTelemetryCollector CR object

```
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: openshift-logging
spec:
  serviceAccount: otel-collector-deployment
  config:
    extensions:
      bearertokenauth:
        filename: "/var/run/secrets/kubernetes.io/serviceaccount/token"
    receivers:
      otlp:
        protocols:
          grpc: {}
          http: {}
    processors:
      k8sattributes: {}
      resource:
        attributes: (1)
          - key:  kubernetes.namespace_name
            from_attribute: k8s.namespace.name
            action: upsert
          - key:  kubernetes.pod_name
            from_attribute: k8s.pod.name
            action: upsert
          - key: kubernetes.container_name
            from_attribute: k8s.container.name
            action: upsert
          - key: log_type
            value: application
            action: upsert
      transform:
        log_statements:
          - context: log
            statements:
              - set(attributes["level"], ConvertCase(severity_text, "lower"))
    exporters:
      otlphttp:
        endpoint: https://logging-loki-gateway-http.openshift-
```

```
logging.svc.cluster.local:8080/api/logs/v1/application/otlp
      encoding: json
      tls:
        ca_file: "/var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt"
      auth:
        authenticator: bearertokenauth
    debug:
      verbosity: detailed
  service:
    extensions: [bearertokenauth] 2
    pipelines:
      logs:
        receivers: [otlp]
        processors: [k8sattributes, transform, resource]
        exporters: [otlphttp] 3
      logs/test:
        receivers: [otlp]
        processors: []
        exporters: [debug]
```

1 Provides the following resource attributes to be used by the web console: **kubernetes.namespace_name**, **kubernetes.pod_name**, **kubernetes.container_name**, and **log_type**.

2 Enables the BearerTokenAuth Extension that is required by the OTLP HTTP Exporter.

3 Enables the OTLP HTTP Exporter to export logs from the Collector.

TIP

You can deploy **telemetrygen** as a test:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: telemetrygen
spec:
  template:
    spec:
      containers:
        - name: telemetrygen
          image: ghcr.io/open-telemetry/opentelemetry-collector-contrib/telemetrygen:v0.106.1
          args:
            - logs
            - --otlp-endpoint=otel-collector.openshift-logging.svc.cluster.local:4317
            - --otlp-insecure
            - --duration=180s
            - --workers=1
            - --logs=10
            - --otlp-attributes=k8s.container.name="telemetrygen"
      restartPolicy: Never
  backoffLimit: 4
```

8.3. FORWARDING TELEMETRY DATA TO THIRD-PARTY SYSTEMS

The OpenTelemetry Collector exports telemetry data by using the OTLP exporter via the OpenTelemetry Protocol (OTLP) that is implemented over the gRPC or HTTP transports. If you need to forward telemetry data to your third-party system and it does not support the OTLP or other supported protocol in the Red Hat build of OpenTelemetry, then you can deploy an unsupported custom OpenTelemetry Collector that can receive telemetry data via the OTLP and export it to your third-party system by using a custom exporter.

WARNING

Red Hat does not support custom deployments.

Prerequisites

- You have developed your own unsupported custom exporter that can export telemetry data via the OTLP to your third-party system.

Procedure

- Deploy a custom Collector either through the OperatorHub or manually:
  - If your third-party system supports it, deploy the custom Collector by using the OperatorHub.
  - Deploy the custom Collector manually by using a config map, deployment, and service.

    Example of a custom Collector deployment

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: custom-otel-collector-config
data:
  otel-collector-config.yaml: |
    receivers:
      otlp:
        protocols:
          grpc:
    exporters:
      debug: {}
      prometheus:
    service:
      pipelines:
        traces:
          receivers: [otlp]
          exporters: [debug] (1)
---
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: custom-otel-collector-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      component: otel-collector
  template:
    metadata:
      labels:
        component: otel-collector
    spec:
      containers:
      - name: opentelemetry-collector
        image: ghcr.io/open-telemetry/opentelemetry-collector-releases/opentelemetry-
collector-contrib:latest (2)
        command:
        - "/otelcol-contrib"
        - "--config=/conf/otel-collector-config.yaml"
        ports:
        - name: otlp
          containerPort: 4317
          protocol: TCP
        volumeMounts:
        - name: otel-collector-config-vol
          mountPath: /conf
          readOnly: true
      volumes:
      - name: otel-collector-config-vol
        configMap:
          name: custom-otel-collector-config
---
apiVersion: v1
kind: Service
metadata:
  name: custom-otel-collector-service (3)
  labels:
    component: otel-collector
spec:
  type: ClusterIP
  ports:
  - name: otlp-grpc
    port: 4317
    targetPort: 4317
  selector:
    component: otel-collector
```

1. Replace **debug** with the required exporter for your third-party system.
2. Replace the image with the required version of the OpenTelemetry Collector that has the required exporter for your third-party system.
3. The service name is used in the Red Hat build of OpenTelemetry Collector CR to configure the OTLP exporter.

8.4. ADDITIONAL RESOURCES

- OpenTelemetry Protocol (OTLP)

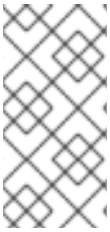
CHAPTER 9. CONFIGURING THE OPENTELEMETRY COLLECTOR METRICS

The following list shows some of these metrics:

- Collector memory usage
- CPU utilization
- Number of active traces and spans processed
- Dropped spans, logs, or metrics
- Exporter and receiver statistics

The Red Hat build of OpenTelemetry Operator automatically creates a service named **<instance_name>-collector-monitoring** that exposes the Collector's internal metrics. This service listens on port **8888** by default.

You can use these metrics for monitoring the Collector's performance, resource consumption, and other internal behaviors. You can also use a Prometheus instance or another monitoring tool to scrape these metrics from the mentioned **<instance_name>-collector-monitoring** service.



NOTE

When the **spec.observability.metrics.enableMetrics** field in the **OpenTelemetryCollector** custom resource (CR) is set to **true**, the **OpenTelemetryCollector** CR automatically creates a Prometheus **ServiceMonitor** or **PodMonitor** CR to enable Prometheus to scrape your metrics.

Prerequisites

- Monitoring for user-defined projects is enabled in the cluster.

Procedure

- To enable metrics of an OpenTelemetry Collector instance, set the **spec.observability.metrics.enableMetrics** field to **true**:

```
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
metadata:
  name: <name>
spec:
  observability:
    metrics:
      enableMetrics: true
```

Verification

You can use the **Administrator** view of the web console to verify successful configuration:

1. Go to **Observe → Targets**.

2. Filter by **Source: User**.
3. Check that the **ServiceMonitors** or **PodMonitors** in the **opentelemetry-collector-<instance_name>** format have the **Up** status.

Additional resources

- Enabling monitoring for user-defined projects

CHAPTER 10. GATHERING THE OBSERVABILITY DATA FROM MULTIPLE CLUSTERS

For a multicluster configuration, you can create one OpenTelemetry Collector instance in each one of the remote clusters and then forward all the telemetry data to one OpenTelemetry Collector instance.

Prerequisites

- The Red Hat build of OpenTelemetry Operator is installed.
- The Tempo Operator is installed.
- A TempoStack instance is deployed on the cluster.
- The following mounted certificates: Issuer, self-signed certificate, CA issuer, client and server certificates. To create any of these certificates, see step 1.

Procedure

1. Mount the following certificates in the OpenTelemetry Collector instance, skipping already mounted certificates.
   a. An Issuer to generate the certificates by using the cert-manager Operator for Red Hat OpenShift.

```
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
spec:
  selfSigned: {}
```

   b. A self-signed certificate.

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: ca
spec:
  isCA: true
  commonName: ca
  subject:
    organizations:
      - <your_organization_name>
    organizationalUnits:
      - Widgets
  secretName: ca-secret
  privateKey:
    algorithm: ECDSA
    size: 256
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
```

c. A CA issuer.

```
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: test-ca-issuer
spec:
  ca:
    secretName: ca-secret
```

d. The client and server certificates.

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: server
spec:
  secretName: server-tls
  isCA: false
  usages:
    - server auth
    - client auth
  dnsNames:
  - "otel.observability.svc.cluster.local" (1)
  issuerRef:
    name: ca-issuer
---
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: client
spec:
  secretName: client-tls
  isCA: false
  usages:
    - server auth
    - client auth
  dnsNames:
  - "otel.observability.svc.cluster.local" (2)
  issuerRef:
    name: ca-issuer
```

(1) List of exact DNS names to be mapped to a solver in the server OpenTelemetry Collector instance.

(2) List of exact DNS names to be mapped to a solver in the client OpenTelemetry Collector instance.

2. Create a service account for the OpenTelemetry Collector instance.

Example ServiceAccount

```
apiVersion: v1
kind: ServiceAccount
```

```
  metadata:
    name: otel-collector-deployment
```

3. Create a cluster role for the service account.

Example ClusterRole

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otel-collector
rules:
  1
  2
- apiGroups: ["", "config.openshift.io"]
  resources: ["pods", "namespaces", "infrastructures", "infrastructures/status"]
  verbs: ["get", "watch", "list"]
```

1. The **k8sattributesprocessor** requires permissions for pods and namespace resources.
2. The **resourcedetectionprocessor** requires permissions for infrastructures and status.

4. Bind the cluster role to the service account.

Example ClusterRoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-collector
subjects:
- kind: ServiceAccount
  name: otel-collector-deployment
  namespace: otel-collector-<example>
roleRef:
  kind: ClusterRole
  name: otel-collector
  apiGroup: rbac.authorization.k8s.io
```

5. Create the YAML file to define the **OpenTelemetryCollector** custom resource (CR) in the edge clusters.

Example OpenTelemetryCollector custom resource for the edge clusters

```
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: otel-collector-<example>
spec:
  mode: daemonset
  serviceAccount: otel-collector-deployment
  config:
```

```
    receivers:
      jaeger:
        protocols:
          grpc: {}
          thrift_binary: {}
          thrift_compact: {}
          thrift_http: {}
      opencensus:
      otlp:
        protocols:
          grpc: {}
          http: {}
      zipkin: {}
    processors:
      batch: {}
      k8sattributes: {}
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
      resourcedetection:
        detectors: [openshift]
    exporters:
      otlphttp:
        endpoint: https://observability-cluster.com:443 (1)
        tls:
          insecure: false
          cert_file: /certs/server.crt
          key_file: /certs/server.key
          ca_file: /certs/ca.crt
    service:
      pipelines:
        traces:
          receivers: [jaeger, opencensus, otlp, zipkin]
          processors: [memory_limiter, k8sattributes, resourcedetection, batch]
          exporters: [otlp]
  volumes:
    - name: otel-certs
      secret:
        name: otel-certs
  volumeMounts:
    - name: otel-certs
      mountPath: /certs
```

(1) The Collector exporter is configured to export OTLP HTTP and points to the OpenTelemetry Collector from the central cluster.

6. Create the YAML file to define the **OpenTelemetryCollector** custom resource (CR) in the central cluster.

Example OpenTelemetryCollector custom resource for the central cluster

```
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
metadata:
```

```
  name: otlp-receiver
  namespace: observability
spec:
  mode: "deployment"
  ingress:
    type: route
    route:
      termination: "passthrough"
  config:
    receivers:
      otlp:
        protocols:
          http:
            tls: (1)
              cert_file: /certs/server.crt
              key_file: /certs/server.key
              client_ca_file: /certs/ca.crt
    exporters:
      otlp:
        endpoint: "tempo-<simplest>-distributor:4317" (2)
        tls:
          insecure: true
    service:
      pipelines:
        traces:
          receivers: [otlp]
          processors: []
          exporters: [otlp]
  volumes:
    - name: otel-certs
      secret:
        name: otel-certs
  volumeMounts:
    - name: otel-certs
      mountPath: /certs
```

1. The Collector receiver requires the certificates listed in the first step.
2. The Collector exporter is configured to export OTLP and points to the Tempo distributor endpoint, which in this example is **"tempo-simplest-distributor:4317"** and already created.

CHAPTER 11. TROUBLESHOOTING

The OpenTelemetry Collector offers multiple ways to measure its health as well as investigate data ingestion issues.

11.1. COLLECTING DIAGNOSTIC DATA FROM THE COMMAND LINE

When submitting a support case, it is helpful to include diagnostic information about your cluster to Red Hat Support. You can use the **oc adm must-gather** tool to gather diagnostic data for resources of various types, such as **OpenTelemetryCollector**, **Instrumentation**, and the created resources like **Deployment**, **Pod**, or **ConfigMap**. The **oc adm must-gather** tool creates a new pod that collects this data.

Procedure

- From the directory where you want to save the collected data, run the **oc adm must-gather** command to collect the data:

```
$ oc adm must-gather --image=ghcr.io/open-telemetry/opentelemetry-operator/must-gather --
\
/usr/bin/must-gather --operator-namespace <operator_namespace> 1
```

1 The default namespace where the Operator is installed is **openshift-opentelemetry-operator**.

Verification

- Verify that the new directory is created and contains the collected data.

11.2. GETTING THE OPENTELEMETRY COLLECTOR LOGS

You can get the logs for the OpenTelemetry Collector as follows.

Procedure

1. Set the relevant log level in the **OpenTelemetryCollector** custom resource (CR):

```
config:
  service:
    telemetry:
      logs:
        level: debug 1
```

1 Collector's log level. Supported values include **info**, **warn**, **error**, or **debug**. Defaults to **info**.

2. Use the **oc logs** command or the web console to retrieve the logs.

11.3. EXPOSING THE METRICS

The OpenTelemetry Collector exposes the following metrics about the data volumes it has processed:

otelcol_receiver_accepted_spans

The number of spans successfully pushed into the pipeline.

otelcol_receiver_refused_spans

The number of spans that could not be pushed into the pipeline.

otelcol_exporter_sent_spans

The number of spans successfully sent to the destination.

otelcol_exporter_enqueue_failed_spans

The number of spans failed to be added to the sending queue.

otelcol_receiver_accepted_logs

The number of logs successfully pushed into the pipeline.

otelcol_receiver_refused_logs

The number of logs that could not be pushed into the pipeline.

otelcol_exporter_sent_logs

The number of logs successfully sent to the destination.

otelcol_exporter_enqueue_failed_logs

The number of logs failed to be added to the sending queue.

otelcol_receiver_accepted_metrics

The number of metrics successfully pushed into the pipeline.

otelcol_receiver_refused_metrics

The number of metrics that could not be pushed into the pipeline.

otelcol_exporter_sent_metrics

The number of metrics successfully sent to the destination.

otelcol_exporter_enqueue_failed_metrics

The number of metrics failed to be added to the sending queue.

You can use these metrics to troubleshoot issues with your Collector. For example, if the **otelcol_receiver_refused_spans** metric has a high value, it indicates that the Collector is not able to process incoming spans.

The Operator creates a **<cr_name>-collector-monitoring** telemetry service that you can use to scrape the metrics endpoint.

Procedure

1. Enable the telemetry service by adding the following lines in the **OpenTelemetryCollector** custom resource (CR):

```
# ...
  config:
    service:
      telemetry:
        metrics:
          readers:
          - pull:
              exporter:
```

```
        prometheus:
          host: 0.0.0.0
          port: 8888 (1)
# ...
```

(1) The port at which the internal collector metrics are exposed. Defaults to **:8888**.

2. Retrieve the metrics by running the following command, which uses the port-forwarding Collector pod:

```
$ oc port-forward <collector_pod>
```

3. In the **OpenTelemetryCollector** CR, set the **enableMetrics** field to **true** to scrape internal metrics:

```
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
spec:
# ...
  mode: deployment
  observability:
    metrics:
      enableMetrics: true
# ...
```

Depending on the deployment mode of the OpenTelemetry Collector, the internal metrics are scraped by using **PodMonitors** or **ServiceMonitors**.

NOTE

Alternatively, if you do not set the **enableMetrics** field to **true**, you can access the metrics endpoint at **http://localhost:8888/metrics**.

4. Optional: If the **User Workload Monitoring** feature is enabled in the web console, go to **Observe → Dashboards** in the web console, and then select the **OpenTelemetry Collector** dashboard from the drop-down list to view it. For more information about the **User Workload Monitoring** feature, see "Enabling monitoring for user-defined projects" in *Monitoring*.

TIP

You can filter the visualized data such as spans or metrics by the Collector instance, namespace, or OpenTelemetry components such as processors, receivers, or exporters.

Additional resources

- Enabling monitoring for user-defined projects

11.4. DEBUG EXPORTER

You can configure the Debug Exporter to export the collected data to the standard output.

Procedure

1. Configure the **OpenTelemetryCollector** custom resource as follows:

```
config:
  exporters:
    debug:
      verbosity: detailed
  service:
    pipelines:
      traces:
        exporters: [debug]
      metrics:
        exporters: [debug]
      logs:
        exporters: [debug]
```

2. Use the **oc logs** command or the web console to export the logs to the standard output.

11.5. DISABLING NETWORK POLICIES

The Red Hat build of OpenTelemetry Operator creates network policies to control the traffic for the Operator and operands to improve security. By default, the network policies are enabled and configured to allow traffic to all the required components. No additional configuration is needed.

If you are experiencing traffic issues for the OpenTelemetry Collector or its Target Allocator component, the problem might be caused by the default network policy configuration. You can disable network policies for the OpenTelemetry Collector to troubleshoot the issue.

Prerequisites

- You have access to the cluster as a cluster administrator with the **cluster-admin** role.

Procedure

- Disable the network policy for the OpenTelemetry Collector by configuring the **OpenTelemetryCollector** custom resource (CR):

```
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: observability
spec:
  networkPolicy:
    enabled: false (1)
  # ...
```

(1) Specify whether to enable network policies by setting **networkPolicy.enabled** to **true** (default) or **false**. Setting it to **false** disables the creation of network policies.

11.6. USING THE NETWORK OBSERVABILITY OPERATOR FOR TROUBLESHOOTING

You can debug the traffic between your observability components by visualizing it with the Network Observability Operator.

Prerequisites

- You have installed the Network Observability Operator as explained in "Installing the Network Observability Operator".

Procedure

1. In the OpenShift Container Platform web console, go to **Observe → Network Traffic → Topology**.
2. Select **Namespace** to filter the workloads by the namespace in which your OpenTelemetry Collector is deployed.
3. Use the network traffic visuals to troubleshoot possible issues. See "Observing the network traffic from the Topology view" for more details.

Additional resources

- Installing the Network Observability Operator
- Observing the network traffic from the Topology view

11.7. TROUBLESHOOTING THE INSTRUMENTATION

To troubleshoot the instrumentation, look for any of the following issues:

- Issues with instrumentation injection into your workload
- Issues with data generation by the instrumentation libraries

11.7.1. Troubleshooting instrumentation injection into your workload

To troubleshoot instrumentation injection, you can perform the following activities:

- Checking if the **Instrumentation** object was created
- Checking if the init-container started
- Checking if the resources were deployed in the correct order
- Searching for errors in the Operator logs
- Double-checking the pod annotations

Procedure

1. Run the following command to verify that the **Instrumentation** object was successfully created:

```
$ oc get instrumentation -n <workload_project> 1
```

1 The namespace where the instrumentation was created.

2. Run the following command to verify that the **opentelemetry-auto-instrumentation** init-container successfully started, which is a prerequisite for instrumentation injection into workloads:

```
$ oc get events -n <workload_project> 1
```

1 The namespace where the instrumentation is injected for workloads.

Example output

```
... Created container opentelemetry-auto-instrumentation
... Started container opentelemetry-auto-instrumentation
```

3. Verify that the resources were deployed in the correct order for the auto-instrumentation to work correctly. The correct order is to deploy the **Instrumentation** custom resource (CR) before the application. For information about the **Instrumentation** CR, see the section "Configuring the instrumentation".

NOTE

When the pod starts, the Red Hat build of OpenTelemetry Operator checks the **Instrumentation** CR for annotations containing instructions for injecting auto-instrumentation. Generally, the Operator then adds an init-container to the application's pod that injects the auto-instrumentation and environment variables into the application's container. If the **Instrumentation** CR is not available to the Operator when the application is deployed, the Operator is unable to inject the auto-instrumentation.

Fixing the order of deployment requires the following steps:

a. Update the instrumentation settings.

b. Delete the instrumentation object.

c. Redeploy the application.

4. Run the following command to inspect the Operator logs for instrumentation errors:

```
$ oc logs -l app.kubernetes.io/name=opentelemetry-operator --container manager -n
openshift-opentelemetry-operator --follow
```

5. Troubleshoot pod annotations for the instrumentations for a specific programming language. See the required annotation fields and values in "Configuring the instrumentation".

a. Verify that the application pods that you are instrumenting are labeled with correct annotations and the appropriate auto-instrumentation settings have been applied.

Example

```
instrumentation.opentelemetry.io/inject-python="true"
```

Example command to get pod annotations for an instrumented Python application

```
$ oc get pods -n <workload_project> -o jsonpath='{range .items[?
(@.metadata.annotations["instrumentation.opentelemetry.io/inject-python"]=="true")]}
{.metadata.name}{"\n"}{end}'
```

b. Verify that the annotation applied to the instrumentation object is correct for the programming language that you are instrumenting.

c. If there are multiple instrumentations in the same namespace, specify the name of the **Instrumentation** object in their annotations.

Example

```
instrumentation.opentelemetry.io/inject-nodejs: "<instrumentation_object>"
```

d. If the **Instrumentation** object is in a different namespace, specify the namespace in the annotation.

Example

```
instrumentation.opentelemetry.io/inject-nodejs: "
<other_namespace>/<instrumentation_object>"
```

e. Verify that the **OpenTelemetryCollector** custom resource specifies the auto-instrumentation annotations under **spec.template.metadata.annotations**. If the auto-instrumentation annotations are in **spec.metadata.annotations** instead, move them into **spec.template.metadata.annotations**.

11.7.2. Troubleshooting telemetry data generation by the instrumentation libraries

You can troubleshoot telemetry data generation by the instrumentation libraries by checking the endpoint, looking for errors in your application logs, and verifying that the Collector is receiving the telemetry data.

Procedure

1. Verify that the instrumentation is transmitting data to the correct endpoint:

```
$ oc get instrumentation <instrumentation_name> -n <workload_project> -o
jsonpath='{.spec.endpoint}'
```

The default endpoint **http://localhost:4317** for the **Instrumentation** object is only applicable to a Collector instance that is deployed as a sidecar in your application pod. If you are using an incorrect endpoint, correct it by editing the **Instrumentation** object and redeploying your application.

2. Inspect your application logs for error messages that might indicate that the instrumentation is malfunctioning:

```
$ oc logs <application_pod> -n <workload_project>
```

3. If the application logs contain error messages that indicate that the instrumentation might be malfunctioning, install the OpenTelemetry SDK and libraries locally. Then run your application locally and troubleshoot for issues between the instrumentation libraries and your application without OpenShift Container Platform.

4. Use the Debug Exporter to verify that the telemetry data is reaching the destination OpenTelemetry Collector instance. For more information, see "Debug Exporter".

CHAPTER 12. MIGRATING

WARNING

The deprecated Red Hat OpenShift Distributed Tracing Platform (Jaeger) 3.5 was the last release of the Red Hat OpenShift Distributed Tracing Platform (Jaeger) that Red Hat supports.

Support for the deprecated Red Hat OpenShift Distributed Tracing Platform (Jaeger) ends on November 3, 2025.

The Red Hat OpenShift Distributed Tracing Platform Operator (Jaeger) will be removed from the **redhat-operators** catalog on November 3, 2025. For more information, see the Red Hat Knowledgebase solution Jaeger Deprecation and Removal in OpenShift.

You must migrate to the Red Hat build of OpenTelemetry Operator and the Tempo Operator for distributed tracing collection and storage. For more information, see "Migrating" in the Red Hat build of OpenTelemetry documentation, "Installing" in the Red Hat build of OpenTelemetry documentation, and "Installing" in the Distributed Tracing Platform documentation.

If you are already using the Red Hat OpenShift Distributed Tracing Platform (Jaeger) for your applications, you can migrate to the Red Hat build of OpenTelemetry, which is based on the OpenTelemetry open-source project.

The Red Hat build of OpenTelemetry provides a set of APIs, libraries, agents, and instrumentation to facilitate observability in distributed systems. The OpenTelemetry Collector in the Red Hat build of OpenTelemetry can ingest the Jaeger protocol, so you do not need to change the SDKs in your applications.

Migration from the Distributed Tracing Platform (Jaeger) to the Red Hat build of OpenTelemetry requires configuring the OpenTelemetry Collector and your applications to report traces seamlessly. You can migrate sidecar and sidecarless deployments.

12.1. MIGRATING WITH SIDECARS

The Red Hat build of OpenTelemetry Operator supports sidecar injection into deployment workloads, so you can migrate from a Distributed Tracing Platform (Jaeger) sidecar to a Red Hat build of OpenTelemetry sidecar.

Prerequisites

- The Red Hat OpenShift Distributed Tracing Platform (Jaeger) is used on the cluster.
- The Red Hat build of OpenTelemetry is installed.

Procedure

1. Configure the OpenTelemetry Collector as a sidecar.

```
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: <otel-collector-namespace>
spec:
  mode: sidecar
  config:
    receivers:
      jaeger:
        protocols:
          grpc: {}
          thrift_binary: {}
          thrift_compact: {}
          thrift_http: {}
    processors:
      batch: {}
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
      resourcedetection:
        detectors: [openshift]
        timeout: 2s
    exporters:
      otlp:
        endpoint: "tempo-<example>-gateway:8090" (1)
        tls:
          insecure: true
    service:
      pipelines:
        traces:
          receivers: [jaeger]
          processors: [memory_limiter, resourcedetection, batch]
          exporters: [otlp]
```

(1) This endpoint points to the Gateway of a TempoStack instance deployed by using the **<example>** Tempo Operator.

2. Create a service account for running your application.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: otel-collector-sidecar
```

3. Create a cluster role for the permissions needed by some processors.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otel-collector-sidecar
rules:
  (1)
```

```
- apiGroups: ["config.openshift.io"]
  resources: ["infrastructures", "infrastructures/status"]
  verbs: ["get", "watch", "list"]
```

1 The **resourcedetectionprocessor** requires permissions for infrastructures and infrastructures/status.

4. Create a **ClusterRoleBinding** to set the permissions for the service account.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-collector-sidecar
subjects:
- kind: ServiceAccount
  name: otel-collector-deployment
  namespace: otel-collector-example
roleRef:
  kind: ClusterRole
  name: otel-collector
  apiGroup: rbac.authorization.k8s.io
```

5. Deploy the OpenTelemetry Collector as a sidecar.

6. Remove the injected Jaeger Agent from your application by removing the **"sidecar.jaegertracing.io/inject": "true"** annotation from your **Deployment** object.

7. Enable automatic injection of the OpenTelemetry sidecar by adding the **sidecar.opentelemetry.io/inject: "true"** annotation to the **.spec.template.metadata.annotations** field of your **Deployment** object.

8. Use the created service account for the deployment of your application to allow the processors to get the correct information and add it to your traces.

12.2. MIGRATING WITHOUT SIDECARS

You can migrate from the Distributed Tracing Platform (Jaeger) to the Red Hat build of OpenTelemetry without sidecar deployment.

Prerequisites

- The Red Hat OpenShift Distributed Tracing Platform (Jaeger) is used on the cluster.
- The Red Hat build of OpenTelemetry is installed.

Procedure

1. Configure OpenTelemetry Collector deployment.

2. Create the project where the OpenTelemetry Collector will be deployed.

```
apiVersion: project.openshift.io/v1
kind: Project
metadata:
```

```
  name: observability
```

3. Create a service account for running the OpenTelemetry Collector instance.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: otel-collector-deployment
  namespace: observability
```

4. Create a cluster role for setting the required permissions for the processors.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otel-collector
rules:
  1
  2
- apiGroups: ["", "config.openshift.io"]
  resources: ["pods", "namespaces", "infrastructures", "infrastructures/status"]
  verbs: ["get", "watch", "list"]
```

1 Permissions for the **pods** and **namespaces** resources are required for the **k8sattributesprocessor**.

2 Permissions for **infrastructures** and **infrastructures/status** are required for **resourcedetectionprocessor**.

5. Create a ClusterRoleBinding to set the permissions for the service account.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-collector
subjects:
- kind: ServiceAccount
  name: otel-collector-deployment
  namespace: observability
roleRef:
  kind: ClusterRole
  name: otel-collector
  apiGroup: rbac.authorization.k8s.io
```

6. Create the OpenTelemetry Collector instance.

NOTE

This collector will export traces to a TempoStack instance. You must create your TempoStack instance by using the Red Hat Tempo Operator and place here the correct endpoint.

```
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: observability
spec:
  mode: deployment
  serviceAccount: otel-collector-deployment
  config:
    receivers:
      jaeger:
        protocols:
          grpc: {}
          thrift_binary: {}
          thrift_compact: {}
          thrift_http: {}
    processors:
      batch: {}
      k8sattributes: {}
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
      resourcedetection:
        detectors: [openshift]
    exporters:
      otlp:
        endpoint: "tempo-example-gateway:8090"
        tls:
          insecure: true
    service:
      pipelines:
        traces:
          receivers: [jaeger]
          processors: [memory_limiter, k8sattributes, resourcedetection, batch]
          exporters: [otlp]
```

7. Point your tracing endpoint to the OpenTelemetry Operator.

8. If you are exporting your traces directly from your application to Jaeger, change the API endpoint from the Jaeger endpoint to the OpenTelemetry Collector endpoint.

Example of exporting traces by using the `jaegerexporter` with Golang

```
exp, err := jaeger.New(jaeger.WithCollectorEndpoint(jaeger.WithEndpoint(url))) (1)
```

(1) The URL points to the OpenTelemetry Collector API endpoint.

CHAPTER 13. UPGRADING

For version upgrades, the Red Hat build of OpenTelemetry Operator uses the Operator Lifecycle Manager (OLM), which controls installation, upgrade, and role-based access control (RBAC) of Operators in a cluster.

The OLM runs in the OpenShift Container Platform by default. The OLM queries for available Operators as well as upgrades for installed Operators.

The Red Hat build of OpenTelemetry Operator automatically upgrades all **OpenTelemetryCollector** custom resources during its startup. The Operator reconciles all managed instances during its startup. If there is an error, the Operator retries the upgrade at exponential backoff. If an upgrade fails, the Operator will retry the upgrade again when it restarts.

When the Red Hat build of OpenTelemetry Operator is upgraded to the new version, it scans for running OpenTelemetry Collector instances that it manages and upgrades them to the version corresponding to the Operator's new version.

13.1. ADDITIONAL RESOURCES

- Operator Lifecycle Manager concepts and resources
- Updating installed Operators

CHAPTER 14. REMOVING

The steps for removing the Red Hat build of OpenTelemetry from an OpenShift Container Platform cluster are as follows:

1. Shut down all Red Hat build of OpenTelemetry pods.
2. Remove any OpenTelemetryCollector instances.
3. Remove the Red Hat build of OpenTelemetry Operator.

14.1. REMOVING AN OPENTELEMETRY COLLECTOR INSTANCE BY USING THE WEB CONSOLE

You can remove an OpenTelemetry Collector instance in the **Administrator** view of the web console.

Prerequisites

- You are logged in to the web console as a cluster administrator with the **cluster-admin** role.
- For Red Hat OpenShift Dedicated, you must be logged in using an account with the **dedicated-admin** role.

Procedure

1. Go to **Operators → Installed Operators → Red Hat build of OpenTelemetry Operator→ OpenTelemetryInstrumentation** or **OpenTelemetryCollector**.
2. To remove the relevant instance, select  **→ Delete** ... **→ Delete**.
3. Optional: Remove the Red Hat build of OpenTelemetry Operator.

14.2. REMOVING AN OPENTELEMETRY COLLECTOR INSTANCE BY USING THE CLI

You can remove an OpenTelemetry Collector instance on the command line.

Prerequisites

- An active OpenShift CLI (**oc**) session by a cluster administrator with the **cluster-admin** role.

  TIP

  - Ensure that your OpenShift CLI (**oc**) version is up to date and matches your OpenShift Container Platform version.
  - Run **oc login**:

    ```
    $ oc login --username=<your_username>
    ```

Procedure

1. Get the name of the OpenTelemetry Collector instance by running the following command:

```
$ oc get deployments -n <project_of_opentelemetry_instance>
```

2. Remove the OpenTelemetry Collector instance by running the following command:

```
$ oc delete opentelemetrycollectors <opentelemetry_instance_name> -n
<project_of_opentelemetry_instance>
```

3. Optional: Remove the Red Hat build of OpenTelemetry Operator.

Verification

- To verify successful removal of the OpenTelemetry Collector instance, run **oc get deployments** again:

```
$ oc get deployments -n <project_of_opentelemetry_instance>
```

14.3. ADDITIONAL RESOURCES

- Deleting Operators from a cluster
- Getting started with the OpenShift CLI