OpenShift Container Platform 4.11

Red Hat build of OpenTelemetry

Red Hat build of OpenTelemetry

Last Updated: 2024-02-07

OpenShift Container Platform 4.11 Red Hat build of OpenTelemetry

Red Hat build of OpenTelemetry

Legal Notice

Abstract

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

Table of Contents

CHAPTER 1. RELEASE NOTES FOR RED HAT BUILD OF OPENTELEMETRY

1.1. 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 product 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 has a number of 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 designed to be customizable. It supports various processors, exporters, and receivers.

Auto-instrumentation features

Automatic instrumentation simplifies the process of adding observability to applications. Developers don't 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.

1.2. RED HAT BUILD OF OPENTELEMETRY 3.0

Red Hat build of OpenTelemetry 3.0 is based on OpenTelemetry 0.89.0.

1.2.1. New features and enhancements

This update introduces the following enhancements:

- The **OpenShift distributed tracing data collection Operator** is renamed as the **Red Hat build of OpenTelemetry Operator**.
- Support for the ARM architecture.

- Support for the Prometheus receiver for metrics collection.
- Support for the Kafka receiver and exporter for sending traces and metrics to Kafka.
- Support for cluster-wide proxy environments.
- The Red Hat build of OpenTelemetry Operator creates the Prometheus **ServiceMonitor** custom resource if the Prometheus exporter is enabled.
- The Operator enables the **Instrumentation** custom resource that allows injecting upstream OpenTelemetry auto-instrumentation libraries.

1.2.2. Removal notice

- In Red Hat build of OpenTelemetry 3.0, the Jaeger exporter has been removed. Bug fixes and support are provided only through the end of the 2.9 lifecycle. As an alternative to the Jaeger exporter for sending data to the Jaeger collector, you can use the OTLP exporter instead.

1.2.3. Bug fixes

This update introduces the following bug fixes:

- Fixed support for disconnected environments when using the **oc adm catalog mirror** CLI command.

1.2.4. Known issues

Curently, the cluster monitoring of the Red Hat build of OpenTelemetry Operator is disabled due to a bug (TRACING-3761). The bug is preventing the cluster monitoring from scraping metrics from the Red Hat build of OpenTelemetry Operator due to a missing label **openshift.io/cluster-monitoring=true** that is required for the cluster monitoring and service monitor object.

Workaround

You can enable the cluster monitoring as follows:

1. Add the following label in the Operator namespace: **oc label namespace openshift-opentelemetry-operator openshift.io/cluster-monitoring=true**
2. Create a service monitor, role, and role binding:

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: opentelemetry-operator-controller-manager-metrics-service
  namespace: openshift-opentelemetry-operator
spec:
  endpoints:
  - bearerTokenFile: /var/run/secrets/kubernetes.io/serviceaccount/token
    path: /metrics
    port: https
    scheme: https
    tlsConfig:
      insecureSkipVerify: true
  selector:
    matchLabels:
```

```
      app.kubernetes.io/name: opentelemetry-operator
      control-plane: controller-manager
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: otel-operator-prometheus
  namespace: openshift-opentelemetry-operator
  annotations:
    include.release.openshift.io/self-managed-high-availability: "true"
    include.release.openshift.io/single-node-developer: "true"
rules:
- apiGroups:
  - ""
  resources:
  - services
  - endpoints
  - pods
  verbs:
  - get
  - list
  - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: otel-operator-prometheus
  namespace: openshift-opentelemetry-operator
  annotations:
    include.release.openshift.io/self-managed-high-availability: "true"
    include.release.openshift.io/single-node-developer: "true"
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: otel-operator-prometheus
subjects:
- kind: ServiceAccount
  name: prometheus-k8s
  namespace: openshift-monitoring
```

1.3. 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 Hybrid Cloud Console. 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.

1.4. MAKING OPEN SOURCE MORE INCLUSIVE

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

CHAPTER 2. INSTALLING THE RED HAT BUILD OF OPENTELEMETRY

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.

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

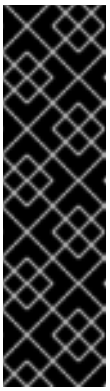
You can install the Red Hat build of OpenTelemetry from the **Administrator** view of the web console.

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-operators**
- **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 project of your choice for the **OpenTelemetry Collector** instance that you will create in the next step by going to **Home → Projects → Create Project**.
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) with the OTLP, Jaeger, Zipkin receivers and the debug exporter.

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: <project_of_opentelemetry_collector_instance>
spec:
  mode: deployment
  config: |
    receivers:
      otlp:
        protocols:
          grpc:
          http:
      jaeger:
        protocols:
          grpc:
          thrift_binary:
          thrift_compact:
          thrift_http:
      zipkin:
    processors:
      batch:
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
    exporters:
      debug:
    service:
      pipelines:
        traces:
          receivers: [otlp,jaeger,zipkin]
          processors: [memory_limiter,batch]
          exporters: [debug]
```

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.

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

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

```
$ oc new-project <project_of_opentelemetry_collector_instance>
```

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

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

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 **OpenTelemetry Collector** custom resource (CR) with the OTLP, Jaeger, and Zipkin receivers and the debug exporter:

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: <project_of_opentelemetry_collector_instance>
spec:
  mode: deployment
  config: |
    receivers:
      otlp:
        protocols:
          grpc:
          http:
      jaeger:
        protocols:
```

```
        grpc:
        thrift_binary:
        thrift_compact:
        thrift_http:
    zipkin:
  processors:
    batch:
    memory_limiter:
      check_interval: 1s
      limit_percentage: 50
      spike_limit_percentage: 30
  exporters:
    debug:
  service:
    pipelines:
      traces:
        receivers: [otlp,jaeger,zipkin]
        processors: [memory_limiter,batch]
        exporters: [debug]
```

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>
```

2.3. 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 3. CONFIGURING AND DEPLOYING THE RED HAT BUILD OF OPENTELEMETRY

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.

3.1. OPENTELEMETRY COLLECTOR CONFIGURATION OPTIONS

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

Receivers

A receiver, which can be push or pull based, is how data gets into the Collector. 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.

Processors

Optional. Processors process the data between it is received and exported. 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.

Exporters

An exporter, which can be push or pull based, is how you send data to one or more back ends or destinations. 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.

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.

Extensions

An extension adds capabilities to the Collector. For example, authentication can be added to the receivers and exporters automatically.

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/v1alpha1
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: jaeger-production-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: [jaeger]
      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 3.1. Parameters used by the Operator to define the OpenTelemetry Collector

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 data between it is received and exported. By default, no processors are enabled.	**batch**, **memory_limiter**, **resourcedetection**, **attributes**, **span**, **k8sattributes**, **filter**, **routing**	None

Parameter	Description	Values	Default
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, that is by consuming data as end-of-pipeline exporters and emitting data as start-of-pipeline receivers, and can be used to summarize, replicate, or route consumed data.	**spanmetrics**	None
extensions:	Optional components for tasks that do not involve processing telemetry data.	**bearertokenauth**, **oauth2client**, **jaegerremotesampli n**, **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.tra ces**.		None

Parameter	Description	Values	Default

`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

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

3.1.1. OpenTelemetry Collector components

3.1.1.1. Receivers

Receivers get data into the Collector.

3.1.1.1.1. OTLP Receiver

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, 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** 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.

3.1.1.1.2. 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.

3.1.1.1.3. Prometheus Receiver

The Prometheus receiver is currently a Technology Preview feature only.

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.

3.1.1.1.4. 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.

3.1.1.1.5. Kafka Receiver

The Kafka receiver is currently a Technology Preview feature only.

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

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 plaintext authentication configuration. If omitted, plaintext 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.

3.1.1.1.6. 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.

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 *.

3.1.1.2. Processors

Processors run through the data between it is received and exported.

3.1.1.2.1. 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: |
  processor:
    batch:
      timeout: 5s
      send_batch_max_size: 10000
  service:
```

```
    pipelines:
      traces:
        processors: [batch]
      metrics:
        processors: [batch]
```

Table 3.2. 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**

3.1.1.2.2. 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: |
    processor:
      memory_limiter:
        check_interval: 1s
        limit_mib: 4000
        spike_limit_mib: 800
    service:
```

```
    pipelines:
      traces:
        processors: [batch]
      metrics:
        processors: [batch]
```

Table 3.3. 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**

3.1.1.2.3. Resource Detection processor

The Resource Detection processor is currently a Technology Preview feature only.

The Resource Detection processor identifies host resource details in alignment with OpenTelemetry's resource semantic standards. Using the detected information, it can add or replace the resource values in telemetry data. This processor supports traces, metrics, and can be used with multiple detectors such

as the Docket metadata detector or the **OTEL_RESOURCE_ATTRIBUTES** environment variable detector.

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: |
  processor:
    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.

3.1.1.2.4. Attributes processor

The Attributes processor is currently a Technology Preview feature only.

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.

The 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 will be 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
```

3.1.1.2.5. Resource processor

The Resource processor is currently a Technology Preview feature only.

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

OpenTelemetry Collector using the Resource Detection processor

```
config: |
  processor:
```

```
    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.

3.1.1.2.6. Span processor

The Span processor is currently a Technology Preview feature only.

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

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

OpenTelemetry Collector using the Span processor for renaming a span

```
config: |
  processor:
    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 the processor to extract attributes from the span name.

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

```
config: |
  processor:
    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: |
  processor:
    span/set_status:
      status:
        code: Error
        description: "<error_description>"
```

3.1.1.2.7. Kubernetes Attributes processor

The Kubernetes Attributes processor is currently a Technology Preview feature only.

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']
```

OpenTelemetry Collector using the Kubernetes Attributes processor

```
config: |
  processors:
      k8sattributes:
          filter:
              node_from_env_var: KUBE_NODE_NAME
```

3.1.1.3. Filter processor

The Filter processor is currently a Technology Preview feature only.

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. The conditions can be combined by using the logical OR operator. This processor supports traces, metrics, and logs.

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.

3.1.1.4. Routing processor

The Routing processor is currently a Technology Preview feature only.

The Routing processor routes logs, metrics, or traces to specific exporters. This processor can read a header from an incoming HTTP request (gRPC or plain HTTP) or can read a resource attribute, and then directs the trace information to relevant exporters according to the read value.

OpenTelemetry Collector custom resource with an enabled OTLP exporter

```
config: |
  processors:
    routing:
      from_attribute: X-Tenant (1)
      default_exporters: (2)
      - jaeger
      table: (3)
      - value: acme
        exporters: [jaeger/acme]
  exporters:
    jaeger:
      endpoint: localhost:14250
    jaeger/acme:
      endpoint: localhost:24250
```

1. The HTTP header name for the lookup value when performing the route.
2. The default exporter when the attribute value is not present in the table in the next section.
3. The table that defines which values are to be routed to which exporters.

You can optionally create an **attribute_source** configuratiion, which defines where to look for the attribute in **from_attribute**. The allowed value is **context** to search the context, which includes the HTTP headers, or **resource** to search the resource attributes.

3.1.1.5. Exporters

Exporters send data to one or more back ends or destinations.

3.1.1.5.1. OTLP exporter

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

OpenTelemetry Collector custom resource with an 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.

3.1.1.5.2. OTLP HTTP exporter

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

OpenTelemetry Collector custom resource with an 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.

3.1.1.5.3. Debug exporter

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

OpenTelemetry Collector custom resource with an enabled Debug exporter

```
config: |
  exporters:
    debug:
      verbosity: detailed (1)
  service:
    pipelines:
      traces:
        exporters: [logging]
      metrics:
        exporters: [logging]
```

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

3.1.1.5.4. Prometheus exporter

The Prometheus exporter is currently a Technology Preview feature only.

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

OpenTelemetry Collector custom resource with an enabled Prometheus exporter

```
ports:
- name: promexporter (1)
  port: 8889
  protocol: TCP
config: |
  exporters:
    prometheus:
      endpoint: 0.0.0.0:8889 (2)
      tls: (3)
        ca_file: ca.pem
        cert_file: cert.pem
        key_file: key.pem
      namespace: prefix (4)
      const_labels: (5)
        label1: value1
      enable_open_metrics: true (6)
      resource_to_telemetry_conversion: (7)
        enabled: true
      metric_expiration: 180m (8)
      add_metric_suffixes: false (9)
  service:
    pipelines:
      metrics:
        exporters: [prometheus]
```

(1) Exposes the Prometheus port from the Collector pod and service. You can enable scraping of metrics by Prometheus by using the port name in **ServiceMonitor** or **PodMonitor** custom resource.

(2) The network endpoint where the metrics are exposed.

(3) The server-side TLS configuration. Defines paths to TLS certificates.

(4) If set, exports metrics under the provided value. No default.

(5) Key-value pair labels that are applied for every exported metric. No default.

(6) If **true**, metrics are exported 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.

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

(8) Defines how long metrics are exposed without updates. The default is **5m**.

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

3.1.1.5.5. Kafka exporter

The Kafka exporter is currently a Technology Preview feature only.

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

OpenTelemetry Collector custom resource with an 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 plaintext authentication configuration. If omitted, plaintext 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.

3.1.1.6. Connectors

Connectors connect two pipelines.

3.1.1.6.1. Spanmetrics connector

The Spanmetrics connector is currently a Technology Preview feature only.

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**.

3.1.1.7. Extensions

Extensions add capabilities to the Collector.

3.1.1.7.1. BearerTokenAuth extension

The BearerTokenAuth extension is currently a Technology Preview feature only.

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.

3.1.1.7.2. OAuth2Client extension

The OAuth2Client extension is currently a Technology Preview feature only.

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.

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 of the token.
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.

3.1.1.7.3. Jaeger Remote Sampling extension

The Jaeger Remote Sampling extension is currently a Technology Preview feature only.

The Jaeger Remote Sampling extension allows 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.

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:
    otlp:

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

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
      }
    ]
  }
}
```

3.1.1.7.4. Performance Profiler extension

The Performance Profiler extension is currently a Technology Preview feature only.

The Performance Profiler extension enables the Go **net/http/pprof** endpoint. This is typically used by developers to collect performance profiles and investigate issues with the service.

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:
    otlp:

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

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.

3.1.1.7.5. Health Check extension

The Health Check extension is currently a Technology Preview feature only.

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.

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:
    otlp:

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

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

3.1.1.7.6. Memory Ballast extension

The Memory Ballast extension is currently a Technology Preview feature only.

The Memory Ballast extension enables applications to configure memory ballast for the process.

OpenTelemetry Collector custom resource with the configured Memory Ballast extension

```
config: |
  extensions:
    memory_ballast:
      size_mib: 64 1
      size_in_percentage: 20 2

  receivers:
    otlp:
      protocols:
        http:

  exporters:
    otlp:

  service:
    extensions: [memory_ballast]
    pipelines:
      traces:
        receivers: [otlp]
        exporters: [otlp]
```

1 Sets the memory ballast size in MiB. Takes priority over the **size_in_percentage** if both are specified.

2 Sets the memory ballast as a percentage, **1**-**100**, of the total memory. Supports containerized and physical host environments.

3.1.1.7.7. zPages extension

The zPages extension is currently a Technology Preview feature only.

The zPages extension provides an HTTP endpoint for extensions that serve zPages. At the endpoint, this extension serves live data for debugging instrumented components. All core exporters and receivers provide some zPages instrumentation.

zPages are useful for in-process diagnostics without having to depend on a back end to examine traces or metrics.

OpenTelemetry Collector custom resource with the configured zPages extension

```
config: |
  extensions:
```

```
  zpages:
    endpoint: "localhost:55679" (1)

receivers:
  otlp:
    protocols:
      http:
exporters:
  otlp:

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

(1) Specifies the HTTP endpoint that serves zPages. Use **localhost:** to make it available only locally, or **":"** to make it available on all network interfaces. The default is **localhost:55679**.

3.2. GATHERING THE OBSERVABILITY DATA FROM DIFFERENT CLUSTERS WITH THE OPENTELEMETRY COLLECTOR

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:
      - Organization # <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/v1alpha1
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/v1alpha1
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:
      logging:
      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.

3.3. CONFIGURATION FOR SENDING METRICS TO THE MONITORING STACK

The OpenTelemetry Collector custom resource (CR) can be configured to create a Prometheus **ServiceMonitor** CR for scraping the Collector's pipeline metrics and the enabled Prometheus exporters.

Example of the OpenTelemetry Collector custom resource with the Prometheus exporter

```
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:
          address: ":8888"
      pipelines:
        metrics:
          receivers: [otlp]
          exporters: [prometheus]
```

1. Configures the operator to create the Prometheus **ServiceMonitor** CR to scrape the collector's internal metrics endpoint and Prometheus exporter metric endpoints. The metrics will be stored in the OpenShift monitoring stack.

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

Example of the PodMonitor custom resource 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 custom resource.
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.

3.4. SETTING UP MONITORING FOR THE RED HAT BUILD OF OPENTELEMETRY

The Red Hat build of OpenTelemetry Operator supports monitoring and alerting of each OpenTelemtry Collector instance and exposes upgrade and operational metrics about the Operator itself.

3.4.1. Configuring the OpenTelemetry Collector metrics

You can enable metrics and alerts of OpenTelemetry Collector instances.

Prerequisites

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

Procedure

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

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: <name>
spec:
  observability:
    metrics:
      enableMetrics: true
```

Verification

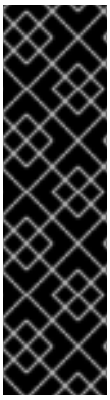
You can use the **Administrator** view of the web console to verify successful configuration:

- Go to **Observe → Targets**, filter by **Source: User**, and check that the **ServiceMonitors** in the **opentelemetry-collector-<instance_name>** format have the **Up** status.

3.5. ADDITIONAL RESOURCES

- Enabling monitoring for user-defined projects

CHAPTER 4. CONFIGURING AND DEPLOYING THE OPENTELEMETRY INSTRUMENTATION INJECTION



IMPORTANT

OpenTelemetry instrumentation injection 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.

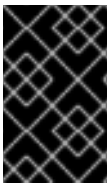
For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

The Red Hat build of OpenTelemetry Operator uses a custom resource definition (CRD) file that defines the configuration of the instrumentation.

4.1. OPENTELEMETRY INSTRUMENTATION CONFIGURATION OPTIONS

The Red Hat build of OpenTelemetry can inject and configure the OpenTelemetry auto-instrumentation libraries into your workloads. Currently, the project supports injection of the instrumentation libraries from Go, Java, Node.js, Python, .NET, and the Apache HTTP Server (**httpd**).

Auto-instrumentation in OpenTelemetry refers to the capability where the framework automatically instruments an application without manual code changes. This enables developers and administrators to get observability into their applications with minimal effort and changes to the existing codebase.



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.

4.1.1. Instrumentation options

Instrumentation options are specified in the **OpenTelemetryCollector** custom resource.

Sample OpenTelemetryCollector custom resource file

```
apiVersion: opentelemetry.io/v1alpha1
kind: Instrumentation
metadata:
  name: java-instrumentation
spec:
  env:
    - name: OTEL_EXPORTER_OTLP_TIMEOUT
      value: "20"
  exporter:
    endpoint: http://production-collector.observability.svc.cluster.local:4317
  propagators:
```

```
    - w3c
  sampler:
    type: parentbased_traceidratio
    argument: "0.25"
  java:
    env:
    - name: OTEL_JAVAAGENT_DEBUG
      value: "true"
```

Table 4.1. Parameters used by the Operator to define the Instrumentation

Parameter	Description	Values
env	Common environment variables to define across all the instrumentations.	
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.	

4.1.2. 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.

4.1.2.1. Configuration of the Apache HTTP Server auto-instrumentation

Table 4.2. Prameters for the .spec.apacheHttpd field

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"
```

4.1.2.2. Configuration of the .NET auto-instrumentation

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

4.1.2.3. Configuration of the Go auto-instrumentation

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"
```

Additional 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>
```

4.1.2.4. Configuration of the Java auto-instrumentation

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

The PodSpec annotation to enable injection

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

4.1.2.5. Configuration of the Node.js auto-instrumentation

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"
instrumentation.opentelemetry.io/otel-go-auto-target-exe: "/path/to/container/executable"
```

The **instrumentation.opentelemetry.io/otel-go-auto-target-exe** annotation sets the value for the required **OTEL_GO_AUTO_TARGET_EXE** environment variable.

4.1.2.6. Configuration of the Python auto-instrumentation

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"
```

4.1.2.7. Configuration of the OpenTelemetry SDK variables

The OpenTelemetry SDK variables in your pod are configurable by using the following annotation:

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

Note that all the annotations accept the following values:

true
: Injects the **Instrumentation** resource from the namespace.

false
: Does not inject any instrumentation.

instrumentation-name
: The name of the instrumentation resource to inject from the current namespace.

other-namespace/instrumentation-name
: The name of the instrumentation resource to inject from another namespace.

4.1.2.8. Multi-container pods

The instrumentation is run on the first container that is available by default according to the pod specification. In some cases, you can also specify target containers for injection.

Pod annotation

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

NOTE

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

CHAPTER 5. USING THE RED HAT BUILD OF OPENTELEMETRY

You can set up and use the Red Hat build of OpenTelemetry to send traces to the OpenTelemetry Collector or the TempoStack.

5.1. FORWARDING TRACES TO A TEMPOSTACK BY USING THE OPENTELEMETRY COLLECTOR

To configure forwarding traces to a TempoStack, 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 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:
     1
     2
   - apiGroups: ["", "config.openshift.io"]
     resources: ["pods", "namespaces", "infrastructures", "infrastructures/status"]
     verbs: ["get", "watch", "list"]
   ```

   1 The **k8sattributesprocessor** requires permissions for pods and namespaces resources.

   2 The **resourcedetectionprocessor** requires permissions for infrastructures and status.

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/v1alpha1
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 **tracegen** as a test:

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

Additional resources

- OpenTelemetry Collector documentation
- Deployment examples on GitHub

5.2. SENDING TRACES AND METRICS TO THE OPENTELEMETRY COLLECTOR

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

5.2.1. 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 (Tempo) 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/v1alpha1
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: [jaeger]
          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.

5.2.2. 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 (Tempo) 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
  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/v1alpha1
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 6. TROUBLESHOOTING THE RED HAT BUILD OF OPENTELEMETRY

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

6.1. 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.

6.2. EXPOSING THE METRICS

The OpenTelemetry Collector exposes the metrics about the data volumes it has processed. The following metrics are for spans, although similar metrics are exposed for metrics and logs signals:

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.

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:

```
config: |
  service:
    telemetry:
```

```
    metrics:
      address: ":8888" (1)
```

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

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

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

2. Access the metrics endpoint at **http://localhost:8888/metrics**.

6.3. 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.

CHAPTER 7. MIGRATING FROM THE DISTRIBUTED TRACING PLATFORM (JAEGER) TO THE RED HAT BUILD OF OPENTELEMETRY

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.

7.1. MIGRATING FROM THE DISTRIBUTED TRACING PLATFORM (JAEGER) TO THE RED HAT BUILD OF OPENTELEMETRY 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/v1alpha1
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.

7.2. MIGRATING FROM THE DISTRIBUTED TRACING PLATFORM (JAEGER) TO THE RED HAT BUILD OF OPENTELEMETRY 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/v1alpha1
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 8. UPDATING THE RED HAT BUILD OF OPENTELEMETRY

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.

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.

8.1. ADDITIONAL RESOURCES

- Operator Lifecycle Manager concepts and resources
- Updating installed Operators

CHAPTER 9. REMOVING THE RED HAT BUILD OF OPENTELEMETRY

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.

9.1. 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.

9.2. 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>
```

9.3. ADDITIONAL RESOURCES

- Deleting Operators from a cluster
- Getting started with the OpenShift CLI