



# OpenShift Container Platform 4.13

## Distributed tracing

Distributed tracing installation, usage, and release notes



## OpenShift Container Platform 4.13 Distributed tracing

---

Distributed tracing installation, usage, and release notes

## Legal Notice

Copyright © 2023 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux<sup>®</sup> is the registered trademark of Linus Torvalds in the United States and other countries.

Java<sup>®</sup> is a registered trademark of Oracle and/or its affiliates.

XFS<sup>®</sup> is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL<sup>®</sup> is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js<sup>®</sup> is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack<sup>®</sup> Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

## Abstract

This document provides information on how to use distributed tracing in OpenShift Container Platform.

# Table of Contents

<b>CHAPTER 1. DISTRIBUTED TRACING RELEASE NOTES .....</b>	<b>6</b>
1.1. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.9	6
1.1.1. Distributed tracing overview	6
1.1.2. Component versions in the Red Hat OpenShift distributed tracing platform 2.9	6
1.1.3. Red Hat OpenShift distributed tracing platform (Jaeger)	6
1.1.3.1. New features and enhancements	6
1.1.3.2. Bug fixes	7
1.1.3.3. Known issues	7
1.1.4. Red Hat OpenShift distributed tracing platform (Tempo)	7
1.1.4.1. New features and enhancements	7
1.1.4.2. Bug fixes	8
1.1.4.3. Known issues	8
1.1.5. Red Hat OpenShift distributed tracing data collection	9
1.1.5.1. New features and enhancements	9
1.1.5.2. Bug fixes	10
1.1.5.3. Known issues	10
1.1.6. Getting support	10
1.1.7. Making open source more inclusive	10
1.2. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.8	10
1.2.1. Distributed tracing overview	10
1.2.2. Component versions in the Red Hat OpenShift distributed tracing platform 2.8	11
1.2.3. Technology Preview features	11
1.2.4. Bug fixes	12
1.2.5. Getting support	12
1.2.6. Making open source more inclusive	12
1.3. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.7	12
1.3.1. Distributed tracing overview	13
1.3.2. Component versions in the Red Hat OpenShift distributed tracing platform 2.7	13
1.3.3. Bug fixes	13
1.3.4. Getting support	13
1.3.5. Making open source more inclusive	14
1.4. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.6	14
1.4.1. Distributed tracing overview	14
1.4.2. Component versions in the Red Hat OpenShift distributed tracing platform 2.6	14
1.4.3. Bug fixes	14
1.4.4. Getting support	15
1.4.5. Making open source more inclusive	15
1.5. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.5	15
1.5.1. Distributed tracing overview	15
1.5.2. Component versions in the Red Hat OpenShift distributed tracing platform 2.5	16
1.5.3. New features and enhancements	16
1.5.4. Bug fixes	16
1.5.5. Getting support	16
1.5.6. Making open source more inclusive	16
1.6. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.4	17
1.6.1. Distributed tracing overview	17
1.6.2. Component versions in the Red Hat OpenShift distributed tracing platform 2.4	17
1.6.3. New features and enhancements	17
1.6.4. Technology Preview features	18
1.6.5. Bug fixes	18
1.6.6. Getting support	18

1.6.7. Making open source more inclusive	18
1.7. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.3	18
1.7.1. Distributed tracing overview	18
1.7.2. Component versions in the Red Hat OpenShift distributed tracing platform 2.3.0	19
1.7.3. Component versions in the Red Hat OpenShift distributed tracing platform 2.3.1	19
1.7.4. New features and enhancements	19
1.7.5. Bug fixes	19
1.7.6. Getting support	19
1.7.7. Making open source more inclusive	20
1.8. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.2	20
1.8.1. Distributed tracing overview	20
1.8.2. Technology Preview features	20
1.8.3. Bug fixes	21
1.8.4. Getting support	21
1.8.5. Making open source more inclusive	21
1.9. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.1	21
1.9.1. Distributed tracing overview	21
1.9.2. Component versions in the Red Hat OpenShift distributed tracing platform 2.1.0	22
1.9.3. Technology Preview features	22
1.9.4. Bug fixes	22
1.9.5. Getting support	23
1.9.6. Making open source more inclusive	23
1.10. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.0	23
1.10.1. Distributed tracing overview	23
1.10.2. Component versions in the Red Hat OpenShift distributed tracing platform 2.0.0	24
1.10.3. New features and enhancements	24
1.10.4. Technology Preview features	24
1.10.5. Bug fixes	24
1.10.6. Getting support	24
1.10.7. Making open source more inclusive	25
<b>CHAPTER 2. DISTRIBUTED TRACING ARCHITECTURE</b>	<b>26</b>
2.1. DISTRIBUTED TRACING ARCHITECTURE	26
2.1.1. Distributed tracing overview	26
2.1.2. Red Hat OpenShift distributed tracing platform features	26
2.1.3. Red Hat OpenShift distributed tracing platform architecture	27
<b>CHAPTER 3. DISTRIBUTED TRACING PLATFORM (JAEGER)</b>	<b>29</b>
3.1. INSTALLING THE DISTRIBUTED TRACING PLATFORM JAEGER	29
3.1.1. Prerequisites	29
3.1.2. Red Hat OpenShift distributed tracing platform installation overview	29
3.1.3. Installing the OpenShift Elasticsearch Operator	30
3.1.4. Installing the Red Hat OpenShift distributed tracing platform (Jaeger) Operator	31
3.2. CONFIGURING AND DEPLOYING THE DISTRIBUTED TRACING PLATFORM JAEGER	32
3.2.1. Supported deployment strategies	33
3.2.2. Deploying the distributed tracing platform default strategy from the web console	34
3.2.2.1. Deploying the distributed tracing platform default strategy from the CLI	35
3.2.3. Deploying the distributed tracing platform production strategy from the web console	36
3.2.3.1. Deploying the distributed tracing platform production strategy from the CLI	37
3.2.4. Deploying the distributed tracing platform streaming strategy from the web console	38
3.2.4.1. Deploying the distributed tracing platform streaming strategy from the CLI	40
3.2.5. Validating your deployment	41
3.2.5.1. Accessing the Jaeger console	41

3.2.6. Customizing your deployment	42
3.2.6.1. Deployment best practices	42
3.2.6.2. Distributed tracing default configuration options	42
3.2.6.3. Jaeger Collector configuration options	45
3.2.6.4. Distributed tracing sampling configuration options	46
3.2.6.5. Distributed tracing storage configuration options	48
3.2.6.5.1. Auto-provisioning an Elasticsearch instance	50
3.2.6.5.2. Connecting to an existing Elasticsearch instance	53
3.2.6.6. Managing certificates with Elasticsearch	62
3.2.6.7. Query configuration options	64
3.2.6.8. Ingester configuration options	65
3.2.7. Injecting sidecars	66
3.2.7.1. Automatically injecting sidecars	67
3.2.7.2. Manually injecting sidecars	67
3.3. UPDATING THE DISTRIBUTED TRACING PLATFORM JAEGER	68
3.3.1. Additional resources	69
3.4. REMOVING THE DISTRIBUTED TRACING PLATFORM JAEGER	69
3.4.1. Removing a distributed tracing platform (Jaeger) instance by using the web console	69
3.4.2. Removing a distributed tracing platform (Jaeger) instance by using the CLI	70
3.4.3. Removing the Red Hat OpenShift distributed tracing platform Operators	71
<b>CHAPTER 4. DISTRIBUTED TRACING PLATFORM (TEMPO)</b>	<b>72</b>
4.1. INSTALLING THE DISTRIBUTED TRACING PLATFORM (TEMPO)	72
4.1.1. Installing the distributed tracing platform (Tempo) from the web console	72
4.1.2. Installing the distributed tracing platform (Tempo) by using the CLI	76
4.1.3. Additional resources	81
4.2. CONFIGURING AND DEPLOYING THE DISTRIBUTED TRACING PLATFORM (TEMPO)	82
4.2.1. Customizing your deployment	82
4.2.1.1. Distributed tracing default configuration options	82
4.2.1.2. The distributed tracing platform (Tempo) storage configuration	85
4.2.1.3. Query configuration options	87
4.2.2. Setting up monitoring for the distributed tracing platform (Tempo)	88
4.2.2.1. Configuring TempoStack metrics and alerts	88
4.2.2.2. Configuring Tempo Operator metrics and alerts	89
4.3. UPDATING THE DISTRIBUTED TRACING PLATFORM (TEMPO)	89
4.3.1. Automatic updates of the distributed tracing platform (Tempo)	89
4.3.2. Additional resources	89
4.4. REMOVING THE RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM (TEMPO)	89
4.4.1. Removing a TempoStack instance by using the web console	90
4.4.2. Removing a TempoStack instance by using the CLI	90
4.4.3. Additional resources	91
<b>CHAPTER 5. DISTRIBUTED TRACING DATA COLLECTION (OPENTELEMETRY)</b>	<b>92</b>
5.1. INSTALLING THE DISTRIBUTED TRACING DATA COLLECTION	92
5.1.1. Installing the distributed tracing data collection from the web console	92
5.1.2. Additional resources	94
5.2. CONFIGURING AND DEPLOYING THE DISTRIBUTED TRACING DATA COLLECTION	94
5.2.1. OpenTelemetry Collector configuration options	94
5.2.1.1. OpenTelemetry Collector components	97
5.2.1.1.1. Receivers	97
5.2.1.1.1.1. OTLP Receiver	97
5.2.1.1.1.2. Jaeger Receiver	98
5.2.1.1.1.3. Zipkin Receiver	99

5.2.1.1.2. Processors	100
5.2.1.1.2.1. Resource Detection processor	100
5.2.1.1.3. Exporters	100
5.2.1.1.3.1. OTLP exporter	100
5.2.1.1.3.2. OTLP HTTP exporter	101
5.2.1.1.3.3. Jaeger exporter	102
5.2.1.1.3.4. Logging exporter	102
5.2.1.1.3.5. Prometheus exporter	103
5.2.2. Sending metrics to the monitoring stack	104
5.2.3. Additional resources	104
5.3. USING THE DISTRIBUTED TRACING DATA COLLECTION	104
5.3.1. Forwarding traces to a TempoStack by using the OpenTelemetry Collector	104
5.3.2. Sending traces and metrics to the OpenTelemetry Collector	107
5.3.2.1. Sending traces and metrics to the OpenTelemetry Collector with sidecar injection	107
5.3.2.2. Sending traces and metrics to the OpenTelemetry Collector without sidecar injection	109
5.4. TROUBLESHOOTING THE DISTRIBUTED TRACING DATA COLLECTION	112
5.4.1. Getting the OpenTelemetry Collector logs	112
5.4.2. Exposing the metrics	113
5.4.3. Logging exporter	113
5.5. MIGRATING FROM THE DISTRIBUTED TRACING PLATFORM (JAEGER) TO THE DISTRIBUTED TRACING DATA COLLECTION	114
5.5.1. Migrating from the distributed tracing platform (Jaeger) to the distributed tracing data collection with sidecars	114
5.5.2. Migrating from the distributed tracing platform (Jaeger) to the distributed tracing data collection without sidecars	116
5.6. REMOVING THE DISTRIBUTED TRACING DATA COLLECTION	118
5.6.1. Removing a distributed tracing data collection instance by using the web console	118
5.6.2. Removing a distributed tracing data collection instance by using the CLI	119
5.6.3. Additional resources	120





# CHAPTER 1. DISTRIBUTED TRACING RELEASE NOTES

## 1.1. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.9

### 1.1.1. Distributed tracing overview

As a service owner, you can use distributed tracing to instrument your services to gather insights into your service architecture. You can use the Red Hat OpenShift distributed tracing platform for monitoring, network profiling, and troubleshooting the interaction between components in modern, cloud-native, microservices-based applications.

With the distributed tracing platform, you can perform the following functions:

- Monitor distributed transactions
- Optimize performance and latency
- Perform root cause analysis

The distributed tracing platform consists of three components:

- **Red Hat OpenShift distributed tracing platform (Jaeger)**, which is based on the open source [Jaeger project](#).
- **Red Hat OpenShift distributed tracing platform (Tempo)** which is based on the open source [Grafana Tempo project](#).
- **Red Hat OpenShift distributed tracing data collection**, which is based on the open source [OpenTelemetry project](#).

### 1.1.2. Component versions in the Red Hat OpenShift distributed tracing platform 2.9

Operator	Component	Version
Red Hat OpenShift distributed tracing platform (Jaeger)	Jaeger	1.47.0
Red Hat OpenShift distributed tracing data collection	OpenTelemetry	0.81.0
Red Hat OpenShift distributed tracing platform (Tempo)	Tempo	2.1.1

### 1.1.3. Red Hat OpenShift distributed tracing platform (Jaeger)

#### 1.1.3.1. New features and enhancements

- None.

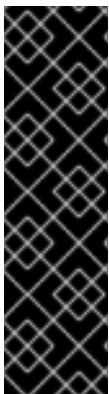
### 1.1.3.2. Bug fixes

- Before this update, connection was refused due to a missing gRPC port on the **jaeger-query** deployment. This issue resulted in **transport: Error while dialing: dial tcp :16685: connect: connection refused** error message. With this update, the Jaeger Query gRPC port (16685) is successfully exposed on the Jaeger Query service. ([TRACING-3322](#))
- Before this update, the wrong port was exposed for **jaeger-production-query**, resulting in refused connection. With this update, the issue is fixed by exposing the Jaeger Query gRPC port (16685) on the Jaeger Query deployment. ([TRACING-2968](#))
- Before this update, when deploying Service Mesh on single-node OpenShift clusters in disconnected environments, the Jaeger pod frequently went into the **Pending** state. With this update, the issue is fixed. ([TRACING-3312](#))
- Before this update, the Jaeger Operator pod restarted with the default memory value due to the **reason: OOMKilled** error message. With this update, this issue is fixed by removing the resource limits. ([TRACING-3173](#))

### 1.1.3.3. Known issues

- Apache Spark is not supported.
- The streaming deployment via AMQ/Kafka is unsupported on IBM Z and IBM Power Systems.

### 1.1.4. Red Hat OpenShift distributed tracing platform (Tempo)



#### IMPORTANT

The Red Hat OpenShift distributed tracing platform (Tempo) 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](#).

#### 1.1.4.1. New features and enhancements

This release introduces the following enhancements for the distributed tracing platform (Tempo):

- Support the [operator maturity](#) Level IV, Deep Insights, which enables upgrading, monitoring, and alerting of **TempoStack** instances and the Tempo Operator.
- Add Ingress and Route configuration for the Gateway.
- Support the **managed** and **unmanaged** states in the **TempoStack** custom resource.
- Expose the following additional ingestion protocols in the Distributor service: Jaeger Thrift binary, Jaeger Thrift compact, Jaeger gRPC, and Zipkin. When the Gateway is enabled, only the OpenTelemetry protocol (OTLP) gRPC is enabled.
- Expose the Jaeger Query gRPC endpoint on the Query Frontend service.

- Support multitenancy without Gateway authentication and authorization.

#### 1.1.4.2. Bug fixes

- Before this update, the Tempo Operator was not compatible with disconnected environments. With this update, the Tempo Operator supports disconnected environments. ([TRACING-3145](#))
- Before this update, the Tempo Operator with TLS failed to start on OpenShift Container Platform. With this update, the mTLS communication is enabled between Tempo components, the Operand starts successfully, and the Jaeger UI is accessible. ([TRACING-3091](#))
- Before this update, the resource limits from the Tempo Operator caused error messages such as **reason: OOMKilled**. With this update, the resource limits for the Tempo Operator are removed to avoid such errors. ([TRACING-3204](#))

#### 1.1.4.3. Known issues

- Currently, the custom TLS CA option is not implemented for connecting to object storage. ([TRACING-3462](#))
- Currently, when used with the Tempo Operator, the Jaeger UI only displays services that have sent traces in the last 15 minutes. For services that did not send traces in the last 15 minutes, traces are still stored but not displayed in the Jaeger UI. ([TRACING-3139](#))
- Currently, the distributed tracing platform (Tempo) fails on the IBM Z (**s390x**) architecture. ([TRACING-3545](#))
- Currently, the Tempo query frontend service must not use internal mTLS when Gateway is not deployed. This issue does not affect the Jaeger Query API. The workaround is to disable mTLS. ([TRACING-3510](#))

#### Workaround

Disable mTLS as follows:

1. Open the Tempo Operator ConfigMap for editing by running the following command:

```
$ oc edit configmap tempo-operator-manager-config -n openshift-tempo-operator 1
```

- 1** The project where the Tempo Operator is installed.

2. Disable the mTLS in the operator configuration by updating the YAML file:

```
data:
  controller_manager_config.yaml: |
    featureGates:
      httpEncryption: false
      grpcEncryption: false
      builtInCertManagement:
        enabled: false
```

3. Restart the Tempo Operator pod by running the following command:

```
$ oc rollout restart deployment.apps/tempo-operator-controller -n openshift-tempo-operator
```

- Missing images for running the Tempo Operator in restricted environments. The Red Hat OpenShift distributed tracing platform (Tempo) CSV is missing references to the operand images. ([TRACING-3523](#))

## Workaround

Add the Tempo Operator related images in the mirroring tool to mirror the images to the registry:

```
kind: ImageSetConfiguration
apiVersion: mirror.openshift.io/v1alpha2
archiveSize: 20
storageConfig:
  local:
    path: /home/user/images
mirror:
  operators:
    - catalog: registry.redhat.io/redhat/redhat-operator-index:v4.13
    packages:
      - name: tempo-product
      channels:
        - name: stable
  additionalImages:
    - name: registry.redhat.io/rhosdt/tempo-
      rhel8@sha256:e4295f837066efb05bcc5897f31eb2bdbd81684a8c59d6f9498dd3590c62c12a
    - name: registry.redhat.io/rhosdt/tempo-gateway-
      rhel8@sha256:b62f5cedfeb5907b638f14ca6aaeea50f41642980a8a6f87b7061e88d90fac23
    - name: registry.redhat.io/rhosdt/tempo-gateway-opa-
      rhel8@sha256:8cd134deca47d6817b26566e272e6c3f75367653d589f5c90855c59b2fab01e9
    - name: registry.redhat.io/rhosdt/tempo-query-
      rhel8@sha256:0da43034f440b8258a48a0697ba643b5643d48b615cdb882ac7f4f1f80aad08e
```

## 1.1.5. Red Hat OpenShift distributed tracing data collection



### IMPORTANT

The Red Hat OpenShift distributed tracing data collection 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](#).

### 1.1.5.1. New features and enhancements

This release introduces the following enhancements for the distributed tracing data collection:

- Support OTLP metrics ingestion. The metrics can be forwarded and stored in the **user-workload-monitoring** via the Prometheus exporter.

- Support the [operator maturity](#) Level IV, Deep Insights, which enables upgrading and monitoring of **OpenTelemetry Collector** instances and the Red Hat OpenShift distributed tracing data collection Operator.
- Report traces and metrics from remote clusters using OTLP or HTTP and HTTPS.
- Collect OpenShift Container Platform resource attributes via the **resourcedetection** processor.
- Support the **managed** and **unmanaged** states in the **OpenTelemetryCollector** custom resource.

#### 1.1.5.2. Bug fixes

None.

#### 1.1.5.3. Known issues

- Currently, you must manually set [operator maturity](#) to Level IV, Deep Insights. ( [TRACING-3431](#) )

### 1.1.6. Getting support

If you experience difficulty with a procedure described in this documentation, or with OpenShift Container Platform in general, visit the [Red Hat Customer Portal](#) . From the Customer Portal, you can:

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

To identify issues with your cluster, you can use Insights in [OpenShift Cluster Manager 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.

#### 1.1.7. 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](#) .

## 1.2. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.8

### 1.2.1. Distributed tracing overview

As a service owner, you can use distributed tracing to instrument your services to gather insights into your service architecture. You can use the Red Hat OpenShift distributed tracing platform for monitoring, network profiling, and troubleshooting the interaction between components in modern,

cloud-native, microservices-based applications.

With the distributed tracing platform, you can perform the following functions:

- Monitor distributed transactions
- Optimize performance and latency
- Perform root cause analysis

The distributed tracing platform consists of three components:

- **Red Hat OpenShift distributed tracing platform (Jaeger)**, which is based on the open source [Jaeger project](#).
- **Red Hat OpenShift distributed tracing platform (Tempo)** which is based on the open source [Grafana Tempo project](#).
- **Red Hat OpenShift distributed tracing data collection**, which is based on the open source [OpenTelemetry project](#).

### 1.2.2. Component versions in the Red Hat OpenShift distributed tracing platform 2.8

Operator	Component	Version
Red Hat OpenShift distributed tracing platform (Jaeger)	Jaeger	1.42
Red Hat OpenShift distributed tracing data collection	OpenTelemetry	0.74.0
Red Hat OpenShift distributed tracing platform (Tempo)	Tempo	0.1.0

### 1.2.3. Technology Preview features

This release introduces support for the Red Hat OpenShift distributed tracing platform (Tempo) as a [Technology Preview](#) feature for Red Hat OpenShift distributed tracing platform.



#### IMPORTANT

The Red Hat OpenShift distributed tracing platform (Tempo) 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 feature uses version 0.1.0 of the Red Hat OpenShift distributed tracing platform (Tempo) and version 2.0.1 of the upstream distributed tracing platform (Tempo) components.

You can use the distributed tracing platform (Tempo) to replace Jaeger so that you can use S3-compatible storage instead of Elasticsearch. Most users who use the distributed tracing platform (Tempo) instead of Jaeger will not notice any difference in functionality because the distributed tracing platform (Tempo) supports the same ingestion and query protocols as Jaeger and uses the same user interface.

If you enable this Technology Preview feature, note the following limitations of the current implementation:

- The distributed tracing platform (Tempo) currently does not support disconnected installations. ([TRACING-3145](#))
- When you use the Jaeger user interface (UI) with the distributed tracing platform (Tempo), the Jaeger UI lists only services that have sent traces within the last 15 minutes. For services that have not sent traces within the last 15 minutes, those traces are still stored even though they are not visible in the Jaeger UI. ([TRACING-3139](#))

Expanded support for the Tempo Operator is planned for future releases of the Red Hat OpenShift distributed tracing platform. Possible additional features might include support for TLS authentication, multitenancy, and multiple clusters. For more information about the Tempo Operator, see the [Tempo community documentation](#).

#### 1.2.4. Bug fixes

This release addresses Common Vulnerabilities and Exposures (CVEs) and bug fixes.

#### 1.2.5. 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.

#### 1.2.6. 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](#).

## 1.3. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.7



### 1.3.1. Distributed tracing overview

As a service owner, you can use distributed tracing to instrument your services to gather insights into your service architecture. You can use the Red Hat OpenShift distributed tracing platform for monitoring, network profiling, and troubleshooting the interaction between components in modern, cloud-native, microservices-based applications.

With the distributed tracing platform, you can perform the following functions:

- Monitor distributed transactions
- Optimize performance and latency
- Perform root cause analysis

The distributed tracing platform consists of three components:

- **Red Hat OpenShift distributed tracing platform (Jaeger)**, which is based on the open source [Jaeger project](#).
- **Red Hat OpenShift distributed tracing platform (Tempo)** which is based on the open source [Grafana Tempo project](#).
- **Red Hat OpenShift distributed tracing data collection**, which is based on the open source [OpenTelemetry project](#).

### 1.3.2. Component versions in the Red Hat OpenShift distributed tracing platform 2.7

Operator	Component	Version
Red Hat OpenShift distributed tracing platform (Jaeger)	Jaeger	1.39
Red Hat OpenShift distributed tracing data collection	OpenTelemetry	0.63.1

### 1.3.3. Bug fixes

This release addresses Common Vulnerabilities and Exposures (CVEs) and bug fixes.

### 1.3.4. 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.

### 1.3.5. 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](#).

## 1.4. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.6

### 1.4.1. Distributed tracing overview

As a service owner, you can use distributed tracing to instrument your services to gather insights into your service architecture. You can use the Red Hat OpenShift distributed tracing platform for monitoring, network profiling, and troubleshooting the interaction between components in modern, cloud-native, microservices-based applications.

With the distributed tracing platform, you can perform the following functions:

- Monitor distributed transactions
- Optimize performance and latency
- Perform root cause analysis

The distributed tracing platform consists of three components:

- **Red Hat OpenShift distributed tracing platform (Jaeger)**, which is based on the open source [Jaeger project](#).
- **Red Hat OpenShift distributed tracing platform (Tempo)** which is based on the open source [Grafana Tempo project](#).
- **Red Hat OpenShift distributed tracing data collection**, which is based on the open source [OpenTelemetry project](#).

### 1.4.2. Component versions in the Red Hat OpenShift distributed tracing platform 2.6

Operator	Component	Version
Red Hat OpenShift distributed tracing platform (Jaeger)	Jaeger	1.38
Red Hat OpenShift distributed tracing data collection	OpenTelemetry	0.60

### 1.4.3. Bug fixes

This release addresses Common Vulnerabilities and Exposures (CVEs) and bug fixes.

#### 1.4.4. 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.

#### 1.4.5. 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](#).

## 1.5. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.5

### 1.5.1. Distributed tracing overview

As a service owner, you can use distributed tracing to instrument your services to gather insights into your service architecture. You can use the Red Hat OpenShift distributed tracing platform for monitoring, network profiling, and troubleshooting the interaction between components in modern, cloud-native, microservices-based applications.

With the distributed tracing platform, you can perform the following functions:

- Monitor distributed transactions
- Optimize performance and latency
- Perform root cause analysis

The distributed tracing platform consists of three components:

- **Red Hat OpenShift distributed tracing platform (Jaeger)**, which is based on the open source [Jaeger project](#).
- **Red Hat OpenShift distributed tracing platform (Tempo)** which is based on the open source [Grafana Tempo project](#).

- **Red Hat OpenShift distributed tracing data collection**, which is based on the open source [OpenTelemetry project](#).

### 1.5.2. Component versions in the Red Hat OpenShift distributed tracing platform 2.5

Operator	Component	Version
Red Hat OpenShift distributed tracing platform (Jaeger)	Jaeger	1.36
Red Hat OpenShift distributed tracing data collection	OpenTelemetry	0.56

### 1.5.3. New features and enhancements

This release introduces support for ingesting OpenTelemetry protocol (OTLP) to the Red Hat OpenShift distributed tracing platform (Jaeger) Operator. The Operator now automatically enables the OTLP ports:

- Port 4317 for the OTLP gRPC protocol.
- Port 4318 for the OTLP HTTP protocol.

This release also adds support for collecting Kubernetes resource attributes to the Red Hat OpenShift distributed tracing data collection Operator.

### 1.5.4. Bug fixes

This release addresses Common Vulnerabilities and Exposures (CVEs) and bug fixes.

### 1.5.5. 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.

### 1.5.6. 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](#).

## 1.6. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.4

### 1.6.1. Distributed tracing overview

As a service owner, you can use distributed tracing to instrument your services to gather insights into your service architecture. You can use the Red Hat OpenShift distributed tracing platform for monitoring, network profiling, and troubleshooting the interaction between components in modern, cloud-native, microservices-based applications.

With the distributed tracing platform, you can perform the following functions:

- Monitor distributed transactions
- Optimize performance and latency
- Perform root cause analysis

The distributed tracing platform consists of three components:

- **Red Hat OpenShift distributed tracing platform (Jaeger)**, which is based on the open source [Jaeger project](#).
- **Red Hat OpenShift distributed tracing platform (Tempo)** which is based on the open source [Grafana Tempo project](#).
- **Red Hat OpenShift distributed tracing data collection**, which is based on the open source [OpenTelemetry project](#).

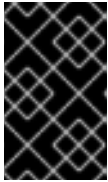
### 1.6.2. Component versions in the Red Hat OpenShift distributed tracing platform 2.4

Operator	Component	Version
Red Hat OpenShift distributed tracing platform (Jaeger)	Jaeger	1.34.1
Red Hat OpenShift distributed tracing data collection	OpenTelemetry	0.49

### 1.6.3. New features and enhancements

This release adds support for auto-provisioning certificates using the Red Hat Elasticsearch Operator.

- Self-provisioning by using the Red Hat OpenShift distributed tracing platform (Jaeger) Operator to call the Red Hat Elasticsearch Operator during installation.



## IMPORTANT

When upgrading to the Red Hat OpenShift distributed tracing platform 2.4, the operator recreates the Elasticsearch instance, which might take five to ten minutes. Distributed tracing will be down and unavailable for that period.

### 1.6.4. Technology Preview features

- Creating the Elasticsearch instance and certificates first and then configuring the distributed tracing platform (Jaeger) to use the certificate is a [Technology Preview](#) for this release.

### 1.6.5. Bug fixes

This release addresses Common Vulnerabilities and Exposures (CVEs) and bug fixes.

### 1.6.6. Getting support

If you experience difficulty with a procedure described in this documentation, or with OpenShift Container Platform in general, visit the [Red Hat Customer Portal](#). From the Customer Portal, you can:

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

To identify issues with your cluster, you can use Insights in [OpenShift Cluster Manager 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.

### 1.6.7. 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](#).

## 1.7. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.3

### 1.7.1. Distributed tracing overview

As a service owner, you can use distributed tracing to instrument your services to gather insights into your service architecture. You can use the Red Hat OpenShift distributed tracing platform for monitoring, network profiling, and troubleshooting the interaction between components in modern, cloud-native, microservices-based applications.

With the distributed tracing platform, you can perform the following functions:

- Monitor distributed transactions

- Optimize performance and latency
- Perform root cause analysis

The distributed tracing platform consists of three components:

- **Red Hat OpenShift distributed tracing platform (Jaeger)**, which is based on the open source [Jaeger project](#).
- **Red Hat OpenShift distributed tracing platform (Tempo)** which is based on the open source [Grafana Tempo project](#).
- **Red Hat OpenShift distributed tracing data collection**, which is based on the open source [OpenTelemetry project](#).

### 1.7.2. Component versions in the Red Hat OpenShift distributed tracing platform 2.3.0

Operator	Component	Version
Red Hat OpenShift distributed tracing platform (Jaeger)	Jaeger	1.30.1
Red Hat OpenShift distributed tracing data collection	OpenTelemetry	0.44.0

### 1.7.3. Component versions in the Red Hat OpenShift distributed tracing platform 2.3.1

Operator	Component	Version
Red Hat OpenShift distributed tracing platform (Jaeger)	Jaeger	1.30.2
Red Hat OpenShift distributed tracing data collection	OpenTelemetry	0.44.1-1

### 1.7.4. New features and enhancements

With this release, the Red Hat OpenShift distributed tracing platform (Jaeger) Operator is now installed to the **openshift-distributed-tracing** namespace by default. Before this update, the default installation had been in the **openshift-operators** namespace.

### 1.7.5. Bug fixes

This release addresses Common Vulnerabilities and Exposures (CVEs) and bug fixes.

### 1.7.6. Getting support

If you experience difficulty with a procedure described in this documentation, or with OpenShift Container Platform in general, visit the [Red Hat Customer Portal](#). From the Customer Portal, you can:

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

To identify issues with your cluster, you can use Insights in [OpenShift Cluster Manager 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.

### 1.7.7. 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](#).

## 1.8. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.2

### 1.8.1. Distributed tracing overview

As a service owner, you can use distributed tracing to instrument your services to gather insights into your service architecture. You can use the Red Hat OpenShift distributed tracing platform for monitoring, network profiling, and troubleshooting the interaction between components in modern, cloud-native, microservices-based applications.

With the distributed tracing platform, you can perform the following functions:

- Monitor distributed transactions
- Optimize performance and latency
- Perform root cause analysis

The distributed tracing platform consists of three components:

- **Red Hat OpenShift distributed tracing platform (Jaeger)**, which is based on the open source [Jaeger project](#).
- **Red Hat OpenShift distributed tracing platform (Tempo)** which is based on the open source [Grafana Tempo project](#).
- **Red Hat OpenShift distributed tracing data collection** which is based on the open source [OpenTelemetry project](#).

### 1.8.2. Technology Preview features



- The unsupported OpenTelemetry Collector components included in the 2.1 release are removed.

### 1.8.3. Bug fixes

This release of the Red Hat OpenShift distributed tracing platform addresses Common Vulnerabilities and Exposures (CVEs) and bug fixes.

### 1.8.4. 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.

### 1.8.5. 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](#).

## 1.9. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.1

### 1.9.1. Distributed tracing overview

As a service owner, you can use distributed tracing to instrument your services to gather insights into your service architecture. You can use the Red Hat OpenShift distributed tracing platform for monitoring, network profiling, and troubleshooting the interaction between components in modern, cloud-native, microservices-based applications.

With the distributed tracing platform, you can perform the following functions:

- Monitor distributed transactions
- Optimize performance and latency
- Perform root cause analysis

The distributed tracing platform consists of three components:

- **Red Hat OpenShift distributed tracing platform (Jaeger)**, which is based on the open source [Jaeger project](#).
- **Red Hat OpenShift distributed tracing platform (Tempo)** which is based on the open source [Grafana Tempo project](#).
- **Red Hat OpenShift distributed tracing data collection**, which is based on the open source [OpenTelemetry project](#).

### 1.9.2. Component versions in the Red Hat OpenShift distributed tracing platform 2.1.0

Operator	Component	Version
Red Hat OpenShift distributed tracing platform (Jaeger)	Jaeger	1.29.1
Red Hat OpenShift distributed tracing data collection	OpenTelemetry	0.41.1

### 1.9.3. Technology Preview features

- This release introduces a breaking change to how to configure certificates in the OpenTelemetry custom resource file. With this update, the **ca\_file** moves under **tls** in the custom resource, as shown in the following examples.

#### CA file configuration for OpenTelemetry version 0.33

```
spec:
  mode: deployment
  config: |
    exporters:
      jaeger:
        endpoint: jaeger-production-collector-headless.tracing-system.svc:14250
        ca_file: "/var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt"
```

#### CA file configuration for OpenTelemetry version 0.41.1

```
spec:
  mode: deployment
  config: |
    exporters:
      jaeger:
        endpoint: jaeger-production-collector-headless.tracing-system.svc:14250
        tls:
          ca_file: "/var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt"
```

### 1.9.4. Bug fixes

This release addresses Common Vulnerabilities and Exposures (CVEs) and bug fixes.

### 1.9.5. 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.

### 1.9.6. 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](#).

## 1.10. RELEASE NOTES FOR RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM 2.0

### 1.10.1. Distributed tracing overview

As a service owner, you can use distributed tracing to instrument your services to gather insights into your service architecture. You can use the Red Hat OpenShift distributed tracing platform for monitoring, network profiling, and troubleshooting the interaction between components in modern, cloud-native, microservices-based applications.

With the distributed tracing platform, you can perform the following functions:

- Monitor distributed transactions
- Optimize performance and latency
- Perform root cause analysis

The distributed tracing platform consists of three components:

- **Red Hat OpenShift distributed tracing platform (Jaeger)**, which is based on the open source [Jaeger project](#).
- **Red Hat OpenShift distributed tracing platform (Tempo)** which is based on the open source [Grafana Tempo project](#).
- **Red Hat OpenShift distributed tracing data collection**, which is based on the open source [OpenTelemetry project](#).

## 1.10.2. Component versions in the Red Hat OpenShift distributed tracing platform 2.0.0

Operator	Component	Version
Red Hat OpenShift distributed tracing platform (Jaeger)	Jaeger	1.28.0
Red Hat OpenShift distributed tracing data collection	OpenTelemetry	0.33.0

## 1.10.3. New features and enhancements

This release introduces the following new features and enhancements:

- Rebrands Red Hat OpenShift Jaeger as the Red Hat OpenShift distributed tracing platform.
- Updates Red Hat OpenShift distributed tracing platform (Jaeger) Operator to Jaeger 1.28. Going forward, the Red Hat OpenShift distributed tracing platform will only support the **stable** Operator channel. Channels for individual releases are no longer supported.
- Adds support for OpenTelemetry protocol (OTLP) to the Query service.
- Introduces a new distributed tracing icon that appears in the OperatorHub.
- Includes rolling updates to the documentation to support the name change and new features.

## 1.10.4. Technology Preview features

- This release adds the Red Hat OpenShift distributed tracing data collection as a [Technology Preview](#), which you install using the Red Hat OpenShift distributed tracing data collection Operator. Red Hat OpenShift distributed tracing data collection is based on the [OpenTelemetry](#) APIs and instrumentation. The Red Hat OpenShift distributed tracing data collection includes the OpenTelemetry Operator and Collector. You can use the Collector to receive traces in the OpenTelemetry or Jaeger protocol and send the trace data to the Red Hat OpenShift distributed tracing platform. Other capabilities of the Collector are not supported at this time. The OpenTelemetry Collector allows developers to instrument their code with vendor agnostic APIs, avoiding vendor lock-in and enabling a growing ecosystem of observability tooling.

## 1.10.5. Bug fixes

This release addresses Common Vulnerabilities and Exposures (CVEs) and bug fixes.

## 1.10.6. Getting support

If you experience difficulty with a procedure described in this documentation, or with OpenShift Container Platform in general, visit the [Red Hat Customer Portal](#). From the Customer Portal, you can:

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

- Access other product documentation.

To identify issues with your cluster, you can use Insights in [OpenShift Cluster Manager 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.

### 1.10.7. 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. DISTRIBUTED TRACING ARCHITECTURE

### 2.1. DISTRIBUTED TRACING ARCHITECTURE

Every time a user takes an action in an application, a request is executed by the architecture that may require dozens of different services to participate to produce a response. Red Hat OpenShift distributed tracing platform lets you perform distributed tracing, which records the path of a request through various microservices that make up an application.

*Distributed tracing* is a technique that is used to tie the information about different units of work together – usually executed in different processes or hosts – to understand a whole chain of events in a distributed transaction. Developers can visualize call flows in large microservice architectures with distributed tracing. It is valuable for understanding serialization, parallelism, and sources of latency.

Red Hat OpenShift distributed tracing platform records the execution of individual requests across the whole stack of microservices, and presents them as traces. A *trace* is a data/execution path through the system. An end-to-end trace is comprised of one or more spans.

A *span* represents a logical unit of work in Red Hat OpenShift distributed tracing platform that has an operation name, the start time of the operation, and the duration, as well as potentially tags and logs. Spans may be nested and ordered to model causal relationships.

#### 2.1.1. Distributed tracing overview

As a service owner, you can use distributed tracing to instrument your services to gather insights into your service architecture. You can use the Red Hat OpenShift distributed tracing platform for monitoring, network profiling, and troubleshooting the interaction between components in modern, cloud-native, microservices-based applications.

With the distributed tracing platform, you can perform the following functions:

- Monitor distributed transactions
- Optimize performance and latency
- Perform root cause analysis

The distributed tracing platform consists of three components:

- **Red Hat OpenShift distributed tracing platform (Jaeger)**, which is based on the open source [Jaeger project](#).
- **Red Hat OpenShift distributed tracing platform (Tempo)** which is based on the open source [Grafana Tempo project](#).
- **Red Hat OpenShift distributed tracing data collection**, which is based on the open source [OpenTelemetry project](#).

#### 2.1.2. Red Hat OpenShift distributed tracing platform features

Red Hat OpenShift distributed tracing platform provides the following capabilities:

- Integration with Kiali – When properly configured, you can view distributed tracing platform data from the Kiali console.

- High scalability – The distributed tracing platform back end is designed to have no single points of failure and to scale with the business needs.
- Distributed Context Propagation – Enables you to connect data from different components together to create a complete end-to-end trace.
- Backwards compatibility with Zipkin – Red Hat OpenShift distributed tracing platform has APIs that enable it to be used as a drop-in replacement for Zipkin, but Red Hat is not supporting Zipkin compatibility in this release.

### 2.1.3. Red Hat OpenShift distributed tracing platform architecture

Red Hat OpenShift distributed tracing platform is made up of several components that work together to collect, store, and display tracing data.

- **Red Hat OpenShift distributed tracing platform (Jaeger)**– This component is based on the open source [Jaeger project](#).
  - **Client** (Jaeger client, Tracer, Reporter, instrumented application, client libraries)– The distributed tracing platform (Jaeger) clients are language-specific implementations of the OpenTracing API. They can be used to instrument applications for distributed tracing either manually or with a variety of existing open source frameworks, such as Camel (Fuse), Spring Boot (RHOAR), MicroProfile (RHOAR/Thorntail), Wildfly (EAP), and many more, that are already integrated with OpenTracing.
  - **Agent** (Jaeger agent, Server Queue, Processor Workers) – The distributed tracing platform (Jaeger) agent is a network daemon that listens for spans sent over User Datagram Protocol (UDP), which it batches and sends to the Collector. The agent is meant to be placed on the same host as the instrumented application. This is typically accomplished by having a sidecar in container environments such as Kubernetes.
  - **Jaeger Collector** (Collector, Queue, Workers) – Similar to the Jaeger agent, the Jaeger Collector receives spans and places them in an internal queue for processing. This allows the Jaeger Collector to return immediately to the client/agent instead of waiting for the span to make its way to the storage.
  - **Storage** (Data Store) – Collectors require a persistent storage backend. Red Hat OpenShift distributed tracing platform (Jaeger) has a pluggable mechanism for span storage. Note that for this release, the only supported storage is Elasticsearch.
  - **Query** (Query Service) – Query is a service that retrieves traces from storage.
  - **Ingestor** (Ingestor Service) – Red Hat OpenShift distributed tracing platform can use Apache Kafka as a buffer between the Collector and the actual Elasticsearch backing storage. Ingestor is a service that reads data from Kafka and writes to the Elasticsearch storage backend.
  - **Jaeger Console** – With the Red Hat OpenShift distributed tracing platform (Jaeger) user interface, you can visualize your distributed tracing data. On the Search page, you can find traces and explore details of the spans that make up an individual trace.
- **Red Hat OpenShift distributed tracing platform (Tempo)**– This component is based on the open source [Grafana Tempo project](#).
  - **Gateway** – The Gateway handles authentication, authorization, and forwarding requests to the Distributor or Query front-end service.

- **Distributor** – The Distributor accepts spans in multiple formats including Jaeger, OpenTelemetry, and Zipkin. It routes spans to Ingesters by hashing the **traceID** and using a distributed consistent hash ring.
- **Ingester** – The Ingester batches a trace into blocks, creates bloom filters and indexes, and then flushes it all to the back end.
- **Query Frontend** – The Query Frontend is responsible for sharding the search space for an incoming query. The search query is then sent to the Queriers. The Query Frontend deployment exposes the Jaeger UI through the Tempo Query sidecar.
- **Querier** – The Querier is responsible for finding the requested trace ID in either the Ingesters or the back-end storage. Depending on parameters, it can query the Ingesters and pull Bloom indexes from the back end to search blocks in object storage.
- **Compactor** – The Compactors stream blocks to and from the back-end storage to reduce the total number of blocks.
- **Red Hat OpenShift distributed tracing data collection**– This component is based on the open source [OpenTelemetry project](#).
  - **OpenTelemetry Collector** – The OpenTelemetry Collector is a vendor-agnostic way to receive, process, and export telemetry data. The OpenTelemetry Collector supports open-source observability data formats, for example, Jaeger and Prometheus, sending to one or more open-source or commercial back-ends. The Collector is the default location instrumentation libraries export their telemetry data.



## CHAPTER 3. DISTRIBUTED TRACING PLATFORM (JAEGER)

### 3.1. INSTALLING THE DISTRIBUTED TRACING PLATFORM JAEGER

You can install Red Hat OpenShift distributed tracing platform on OpenShift Container Platform in either of two ways:

- You can install Red Hat OpenShift distributed tracing platform as part of Red Hat OpenShift Service Mesh. Distributed tracing is included by default in the Service Mesh installation. To install Red Hat OpenShift distributed tracing platform as part of a service mesh, follow the [Red Hat Service Mesh Installation](#) instructions. You must install Red Hat OpenShift distributed tracing platform in the same namespace as your service mesh, that is, the **ServiceMeshControlPlane** and the Red Hat OpenShift distributed tracing platform resources must be in the same namespace.
- If you do not want to install a service mesh, you can use the Red Hat OpenShift distributed tracing platform Operators to install distributed tracing platform by itself. To install Red Hat OpenShift distributed tracing platform without a service mesh, use the following instructions.

#### 3.1.1. Prerequisites

Before you can install Red Hat OpenShift distributed tracing platform, review the installation activities, and ensure that you meet the prerequisites:

- Possess an active OpenShift Container Platform subscription on your Red Hat account. If you do not have a subscription, contact your sales representative for more information.
- Review the [OpenShift Container Platform 4.13 overview](#).
- Install OpenShift Container Platform 4.13.
  - [Install OpenShift Container Platform 4.13 on AWS](#)
  - [Install OpenShift Container Platform 4.13 on user-provisioned AWS](#)
  - [Install OpenShift Container Platform 4.13 on bare metal](#)
  - [Install OpenShift Container Platform 4.13 on vSphere](#)
- Install the version of the **oc** CLI tool that matches your OpenShift Container Platform version and add it to your path.
- An account with the **cluster-admin** role.

#### 3.1.2. Red Hat OpenShift distributed tracing platform installation overview

The steps for installing Red Hat OpenShift distributed tracing platform are as follows:

- Review the documentation and determine your deployment strategy.
- If your deployment strategy requires persistent storage, install the OpenShift Elasticsearch Operator via the OperatorHub.
- Install the Red Hat OpenShift distributed tracing platform (Jaeger) Operator via the OperatorHub.

- Modify the custom resource YAML file to support your deployment strategy.
- Deploy one or more instances of Red Hat OpenShift distributed tracing platform (Jaeger) to your OpenShift Container Platform environment.

### 3.1.3. Installing the OpenShift Elasticsearch Operator

The default Red Hat OpenShift distributed tracing platform (Jaeger) deployment uses in-memory storage because it is designed to be installed quickly for those evaluating Red Hat OpenShift distributed tracing platform, giving demonstrations, or using Red Hat OpenShift distributed tracing platform (Jaeger) in a test environment. If you plan to use Red Hat OpenShift distributed tracing platform (Jaeger) in production, you must install and configure a persistent storage option, in this case, Elasticsearch.

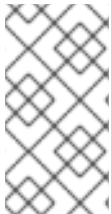
#### Prerequisites

- You have access to the OpenShift Container Platform web console.
- You have access to the cluster as a user with the **cluster-admin** role. If you use Red Hat OpenShift Dedicated, you must have an account with the **dedicated-admin** role.



#### WARNING

Do not install Community versions of the Operators. Community Operators are not supported.



#### NOTE

If you have already installed the OpenShift Elasticsearch Operator as part of OpenShift Logging, you do not need to install the OpenShift Elasticsearch Operator again. The Red Hat OpenShift distributed tracing platform (Jaeger) Operator creates the Elasticsearch instance using the installed OpenShift Elasticsearch Operator.

#### Procedure

1. Log in to the OpenShift Container Platform web console as a user with the **cluster-admin** role. If you use Red Hat OpenShift Dedicated, you must have an account with the **dedicated-admin** role.
2. Navigate to **Operators → OperatorHub**.
3. Type **Elasticsearch** into the filter box to locate the OpenShift Elasticsearch Operator.
4. Click the **OpenShift Elasticsearch Operator** provided by Red Hat to display information about the Operator.
5. Click **Install**.
6. On the **Install Operator** page, select the **stable** Update Channel. This automatically updates your Operator as new versions are released.

7. Accept the default **All namespaces on the cluster (default)** This installs the Operator in the default **openshift-operators-redhat** project and makes the Operator available to all projects in the cluster.



#### NOTE

The Elasticsearch installation requires the **openshift-operators-redhat** namespace for the OpenShift Elasticsearch Operator. The other Red Hat OpenShift distributed tracing platform Operators are installed in the **openshift-operators** namespace.

8. Accept the default **Automatic** approval strategy. By accepting the default, when a new version of this Operator is available, Operator Lifecycle Manager (OLM) automatically upgrades the running instance of your Operator without human intervention. If you select **Manual** updates, when a newer version of an Operator is available, OLM creates an update request. As a cluster administrator, you must then manually approve that update request to have the Operator updated to the new version.



#### NOTE

The **Manual** approval strategy requires a user with appropriate credentials to approve the Operator install and subscription process.

9. Click **Install**.
10. On the **Installed Operators** page, select the **openshift-operators-redhat** project. Wait until you see that the OpenShift Elasticsearch Operator shows a status of "InstallSucceeded" before continuing.

### 3.1.4. Installing the Red Hat OpenShift distributed tracing platform (Jaeger) Operator

To install Red Hat OpenShift distributed tracing platform (Jaeger), you use the [OperatorHub](#) to install the Red Hat OpenShift distributed tracing platform (Jaeger) Operator.

By default, the Operator is installed in the **openshift-operators** project.

#### Prerequisites

- You have access to the OpenShift Container Platform web console.
- You have access to the cluster as a user with the **cluster-admin** role. If you use Red Hat OpenShift Dedicated, you must have an account with the **dedicated-admin** role.
- If you require persistent storage, you must also install the OpenShift Elasticsearch Operator before installing the Red Hat OpenShift distributed tracing platform (Jaeger) Operator.

**WARNING**

Do not install Community versions of the Operators. Community Operators are not supported.

**Procedure**

1. Log in to the OpenShift Container Platform web console as a user with the **cluster-admin** role. If you use Red Hat OpenShift Dedicated, you must have an account with the **dedicated-admin** role.
2. Navigate to **Operators → OperatorHub**.
3. Type **distributed tracing platform** into the filter to locate the Red Hat OpenShift distributed tracing platform (Jaeger) Operator.
4. Click the **Red Hat OpenShift distributed tracing platform (Jaeger) Operator** provided by Red Hat to display information about the Operator.
5. Click **Install**.
6. On the **Install Operator** page, select the **stable** Update Channel. This automatically updates your Operator as new versions are released.
7. Accept the default **All namespaces on the cluster (default)** This installs the Operator in the default **openshift-operators** project and makes the Operator available to all projects in the cluster.
  - Accept the default **Automatic** approval strategy. By accepting the default, when a new version of this Operator is available, Operator Lifecycle Manager (OLM) automatically upgrades the running instance of your Operator without human intervention. If you select **Manual** updates, when a newer version of an Operator is available, OLM creates an update request. As a cluster administrator, you must then manually approve that update request to have the Operator updated to the new version.

**NOTE**

The **Manual** approval strategy requires a user with appropriate credentials to approve the Operator install and subscription process.

8. Click **Install**.
9. Navigate to **Operators → Installed Operators**.
10. On the **Installed Operators** page, select the **openshift-operators** project. Wait until you see that the Red Hat OpenShift distributed tracing platform (Jaeger) Operator shows a status of "Succeeded" before continuing.

## 3.2. CONFIGURING AND DEPLOYING THE DISTRIBUTED TRACING PLATFORM JAEGER

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

If you have installed distributed tracing platform as part of Red Hat OpenShift Service Mesh, you can perform basic configuration as part of the [ServiceMeshControlPlane](#), but for complete control, you must configure a Jaeger CR and then [reference your distributed tracing configuration file in the ServiceMeshControlPlane](#).

The Red Hat OpenShift distributed tracing platform (Jaeger) has predefined deployment strategies. You specify a deployment strategy in the custom resource file. When you create a distributed tracing platform (Jaeger) instance, the Operator uses this configuration file to create the objects necessary for the deployment.

### Jaeger custom resource file showing deployment strategy

```
apiVersion: jaegertracing.io/v1
kind: Jaeger
metadata:
  name: MyConfigFile
spec:
  strategy: production 1
```

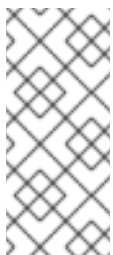
**1** Deployment strategy.

### 3.2.1. Supported deployment strategies

The Red Hat OpenShift distributed tracing platform (Jaeger) Operator currently supports the following deployment strategies:

#### allInOne

- This strategy is intended for development, testing, and demo purposes; it is not intended for production use. The main backend components, Agent, Collector, and Query service, are all packaged into a single executable which is configured, by default, to use in-memory storage.



#### NOTE

In-memory storage is not persistent, which means that if the distributed tracing platform (Jaeger) instance shuts down, restarts, or is replaced, that your trace data will be lost. And in-memory storage cannot be scaled, since each pod has its own memory. For persistent storage, you must use the **production** or **streaming** strategies, which use Elasticsearch as the default storage.

#### production

The production strategy is intended for production environments, where long term storage of trace data is important, as well as a more scalable and highly available architecture is required. Each of the backend components is therefore deployed separately. The Agent can be injected as a sidecar on the instrumented application. The Query and Collector services are configured with a supported storage type - currently Elasticsearch. Multiple instances of each of these components can be provisioned as required for performance and resilience purposes.

#### streaming

The streaming strategy is designed to augment the production strategy by providing a streaming capability that effectively sits between the Collector and the Elasticsearch backend storage. This provides the benefit of reducing the pressure on the backend storage, under high load situations, and enables other trace post-processing capabilities to tap into the real time span data directly from the streaming platform ([AMQ Streams](#)/ [Kafka](#)).



#### NOTE

- The streaming strategy requires an additional Red Hat subscription for AMQ Streams.
- The streaming deployment strategy is currently unsupported on IBM Z.

### 3.2.2. Deploying the distributed tracing platform default strategy from the web console

The custom resource definition (CRD) defines the configuration used when you deploy an instance of Red Hat OpenShift distributed tracing platform. The default CR is named **jaeger-all-in-one-inmemory** and it is configured with minimal resources to ensure that you can successfully install it on a default OpenShift Container Platform installation. You can use this default configuration to create a Red Hat OpenShift distributed tracing platform (Jaeger) instance that uses the **AllInOne** deployment strategy, or you can define your own custom resource file.



#### NOTE

In-memory storage is not persistent. If the Jaeger pod shuts down, restarts, or is replaced, your trace data will be lost. For persistent storage, you must use the **production** or **streaming** strategies, which use Elasticsearch as the default storage.

#### Prerequisites

- The Red Hat OpenShift distributed tracing platform (Jaeger) Operator has been installed.
- You have reviewed the instructions for how to customize the deployment.
- You have access to the cluster as a user with the **cluster-admin** role.

#### Procedure

1. Log in to the OpenShift Container Platform web console as a user with the **cluster-admin** role.
2. Create a new project, for example **tracing-system**.



#### NOTE

If you are installing as part of Service Mesh, the distributed tracing platform resources must be installed in the same namespace as the **ServiceMeshControlPlane** resource, for example **istio-system**.

- a. Go to **Home → Projects**.
- b. Click **Create Project**.

- c. Enter **tracing-system** in the **Name** field.
  - d. Click **Create**.
3. Navigate to **Operators → Installed Operators**.
  4. If necessary, select **tracing-system** from the **Project** menu. You may have to wait a few moments for the Operators to be copied to the new project.
  5. Click the Red Hat OpenShift distributed tracing platform (Jaeger) Operator. On the **Details** tab, under **Provided APIs**, the Operator provides a single link.
  6. Under **Jaeger**, click **Create Instance**.
  7. On the **Create Jaeger** page, to install using the defaults, click **Create** to create the distributed tracing platform (Jaeger) instance.
  8. On the **Jaegers** page, click the name of the distributed tracing platform (Jaeger) instance, for example, **jaeger-all-in-one-inmemory**.
  9. On the **Jaeger Details** page, click the **Resources** tab. Wait until the pod has a status of "Running" before continuing.

### 3.2.2.1. Deploying the distributed tracing platform default strategy from the CLI

Follow this procedure to create an instance of distributed tracing platform (Jaeger) from the command line.

#### Prerequisites

- The Red Hat OpenShift distributed tracing platform (Jaeger) Operator has been installed and verified.
- You have reviewed the instructions for how to customize the deployment.
- You have access to the OpenShift CLI (**oc**) that matches your OpenShift Container Platform version.
- You have access to the cluster as a user with the **cluster-admin** role.

#### Procedure

1. Log in to the OpenShift Container Platform CLI as a user with the **cluster-admin** role by running the following command:

```
$ oc login --username=<NAMEOFUSER> https://<HOSTNAME>:8443
```

2. Create a new project named **tracing-system** by running the following command:

```
$ oc new-project tracing-system
```

3. Create a custom resource file named **jaeger.yaml** that contains the following text:

**Example jaeger-all-in-one.yaml**

```
apiVersion: jaegertracing.io/v1
kind: Jaeger
metadata:
  name: jaeger-all-in-one-inmemory
```

- Run the following command to deploy distributed tracing platform (Jaeger):

```
$ oc create -n tracing-system -f jaeger.yaml
```

- Run the following command to watch the progress of the pods during the installation process:

```
$ oc get pods -n tracing-system -w
```

After the installation process has completed, the output is similar to the following example:

NAME	READY	STATUS	RESTARTS	AGE
jaeger-all-in-one-inmemory-cdff7897b-qhfdx	2/2	Running	0	24s

### 3.2.3. Deploying the distributed tracing platform production strategy from the web console

The **production** deployment strategy is intended for production environments that require a more scalable and highly available architecture, and where long-term storage of trace data is important.

#### Prerequisites

- The OpenShift Elasticsearch Operator has been installed.
- The Red Hat OpenShift distributed tracing platform (Jaeger) Operator has been installed.
- You have reviewed the instructions for how to customize the deployment.
- You have access to the cluster as a user with the **cluster-admin** role.

#### Procedure

- Log in to the OpenShift Container Platform web console as a user with the **cluster-admin** role.
- Create a new project, for example **tracing-system**.



#### NOTE

If you are installing as part of Service Mesh, the distributed tracing platform resources must be installed in the same namespace as the **ServiceMeshControlPlane** resource, for example **istio-system**.

- Navigate to **Home → Projects**.
- Click **Create Project**.
- Enter **tracing-system** in the **Name** field.
- Click **Create**.



3. Navigate to **Operators → Installed Operators**.
4. If necessary, select **tracing-system** from the **Project** menu. You may have to wait a few moments for the Operators to be copied to the new project.
5. Click the Red Hat OpenShift distributed tracing platform (Jaeger) Operator. On the **Overview** tab, under **Provided APIs**, the Operator provides a single link.
6. Under **Jaeger**, click **Create Instance**.
7. On the **Create Jaeger** page, replace the default **all-in-one** YAML text with your production YAML configuration, for example:

#### Example jaeger-production.yaml file with Elasticsearch

```
apiVersion: jaegertracing.io/v1
kind: Jaeger
metadata:
  name: jaeger-production
  namespace:
spec:
  strategy: production
  ingress:
    security: oauth-proxy
  storage:
    type: elasticsearch
    elasticsearch:
      nodeCount: 3
      redundancyPolicy: SingleRedundancy
    esIndexCleaner:
      enabled: true
      numberOfDays: 7
      schedule: 55 23 * * *
    esRollover:
      schedule: */30 * * * *
```

8. Click **Create** to create the distributed tracing platform (Jaeger) instance.
9. On the **Jaegers** page, click the name of the distributed tracing platform (Jaeger) instance, for example, **jaeger-prod-elasticsearch**.
10. On the **Jaeger Details** page, click the **Resources** tab. Wait until all the pods have a status of "Running" before continuing.

#### 3.2.3.1. Deploying the distributed tracing platform production strategy from the CLI

Follow this procedure to create an instance of distributed tracing platform (Jaeger) from the command line.

##### Prerequisites

- The OpenShift Elasticsearch Operator has been installed.
- The Red Hat OpenShift distributed tracing platform (Jaeger) Operator has been installed.
- You have reviewed the instructions for how to customize the deployment.

- You have access to the OpenShift CLI (**oc**) that matches your OpenShift Container Platform version.
- You have access to the cluster as a user with the **cluster-admin** role.

## Procedure

1. Log in to the OpenShift CLI (**oc**) as a user with the **cluster-admin** role by running the following command:

```
$ oc login --username=<NAMEOFUSER> https://<HOSTNAME>:8443
```

2. Create a new project named **tracing-system** by running the following command:

```
$ oc new-project tracing-system
```

3. Create a custom resource file named **jaeger-production.yaml** that contains the text of the example file in the previous procedure.

4. Run the following command to deploy distributed tracing platform (Jaeger):

```
$ oc create -n tracing-system -f jaeger-production.yaml
```

5. Run the following command to watch the progress of the pods during the installation process:

```
$ oc get pods -n tracing-system -w
```

After the installation process has completed, you will see output similar to the following example:

NAME	READY	STATUS	RESTARTS	AGE
elasticsearch-cdm-jaegersystemjaegerproduction-1-6676cf568gwhlw	2/2	Running	0	10m
elasticsearch-cdm-jaegersystemjaegerproduction-2-bcd4c8bf5l6g6w	2/2	Running	0	10m
elasticsearch-cdm-jaegersystemjaegerproduction-3-844d6d9694hhst	2/2	Running	0	10m
jaeger-production-collector-94cd847d-jwjlj	1/1	Running	3	8m32s
jaeger-production-query-5cbfbd499d-tv8zf	3/3	Running	3	8m32s

### 3.2.4. Deploying the distributed tracing platform streaming strategy from the web console

The **streaming** deployment strategy is intended for production environments that require a more scalable and highly available architecture, and where long-term storage of trace data is important.

The **streaming** strategy provides a streaming capability that sits between the Collector and the Elasticsearch storage. This reduces the pressure on the storage under high load situations, and enables other trace post-processing capabilities to tap into the real-time span data directly from the Kafka streaming platform.

**NOTE**

The streaming strategy requires an additional Red Hat subscription for AMQ Streams. If you do not have an AMQ Streams subscription, contact your sales representative for more information.

**NOTE**

The streaming deployment strategy is currently unsupported on IBM Z.

**Prerequisites**

- The AMQ Streams Operator has been installed. If using version 1.4.0 or higher you can use self-provisioning. Otherwise you must create the Kafka instance.
- The Red Hat OpenShift distributed tracing platform (Jaeger) Operator has been installed.
- You have reviewed the instructions for how to customize the deployment.
- You have access to the cluster as a user with the **cluster-admin** role.

**Procedure**

1. Log in to the OpenShift Container Platform web console as a user with the **cluster-admin** role.
2. Create a new project, for example **tracing-system**.

**NOTE**

If you are installing as part of Service Mesh, the distributed tracing platform resources must be installed in the same namespace as the **ServiceMeshControlPlane** resource, for example **istio-system**.

- a. Navigate to **Home → Projects**.
  - b. Click **Create Project**.
  - c. Enter **tracing-system** in the **Name** field.
  - d. Click **Create**.
3. Navigate to **Operators → Installed Operators**.
  4. If necessary, select **tracing-system** from the **Project** menu. You may have to wait a few moments for the Operators to be copied to the new project.
  5. Click the Red Hat OpenShift distributed tracing platform (Jaeger) Operator. On the **Overview** tab, under **Provided APIs**, the Operator provides a single link.
  6. Under **Jaeger**, click **Create Instance**.
  7. On the **Create Jaeger** page, replace the default **all-in-one** YAML text with your streaming YAML configuration, for example:

**Example jaeger-streaming.yaml file**

—

```

apiVersion: jaegertracing.io/v1
kind: Jaeger
metadata:
  name: jaeger-streaming
spec:
  strategy: streaming
  collector:
    options:
      kafka:
        producer:
          topic: jaeger-spans
          #Note: If brokers are not defined,AMQStreams 1.4.0+ will self-provision Kafka.
          brokers: my-cluster-kafka-brokers.kafka:9092
  storage:
    type: elasticsearch
  ingester:
    options:
      kafka:
        consumer:
          topic: jaeger-spans
          brokers: my-cluster-kafka-brokers.kafka:9092

```

8. Click **Create** to create the distributed tracing platform (Jaeger) instance.
9. On the **Jaegers** page, click the name of the distributed tracing platform (Jaeger) instance, for example, **jaeger-streaming**.
10. On the **Jaeger Details** page, click the **Resources** tab. Wait until all the pods have a status of "Running" before continuing.

### 3.2.4.1. Deploying the distributed tracing platform streaming strategy from the CLI

Follow this procedure to create an instance of distributed tracing platform (Jaeger) from the command line.

#### Prerequisites

- The AMQ Streams Operator has been installed. If using version 1.4.0 or higher you can use self-provisioning. Otherwise you must create the Kafka instance.
- The Red Hat OpenShift distributed tracing platform (Jaeger) Operator has been installed.
- You have reviewed the instructions for how to customize the deployment.
- You have access to the OpenShift CLI (**oc**) that matches your OpenShift Container Platform version.
- You have access to the cluster as a user with the **cluster-admin** role.

#### Procedure

1. Log in to the OpenShift CLI (**oc**) as a user with the **cluster-admin** role by running the following command:

```
$ oc login --username=<NAMEOFUSER> https://<HOSTNAME>:8443
```

2. Create a new project named **tracing-system** by running the following command:

```
$ oc new-project tracing-system
```

3. Create a custom resource file named **jaeger-streaming.yaml** that contains the text of the example file in the previous procedure.
4. Run the following command to deploy Jaeger:

```
$ oc create -n tracing-system -f jaeger-streaming.yaml
```

5. Run the following command to watch the progress of the pods during the installation process:

```
$ oc get pods -n tracing-system -w
```

After the installation process has completed, you should see output similar to the following example:

NAME	READY	STATUS	RESTARTS	AGE
elasticsearch-cdm-jaegersystemjaegerstreaming-1-697b66d6fcztcnn	2/2	Running	0	5m40s
elasticsearch-cdm-jaegersystemjaegerstreaming-2-5f4b95c78b9gckz	2/2	Running	0	5m37s
elasticsearch-cdm-jaegersystemjaegerstreaming-3-7b6d964576nnz97	2/2	Running	0	5m5s
jaeger-streaming-collector-6f6db7f99f-rtcfm	1/1	Running	0	80s
jaeger-streaming-entity-operator-6b6d67cc99-4lm9q	3/3	Running	2	2m18s
jaeger-streaming-ingester-7d479847f8-5h8kc	1/1	Running	0	80s
jaeger-streaming-kafka-0	2/2	Running	0	3m1s
jaeger-streaming-query-65bf5bb854-ncnc7	3/3	Running	0	80s
jaeger-streaming-zookeeper-0	2/2	Running	0	3m39s

### 3.2.5. Validating your deployment

#### 3.2.5.1. Accessing the Jaeger console

To access the Jaeger console you must have either Red Hat OpenShift Service Mesh or Red Hat OpenShift distributed tracing platform installed, and Red Hat OpenShift distributed tracing platform (Jaeger) installed, configured, and deployed.

The installation process creates a route to access the Jaeger console.

If you know the URL for the Jaeger console, you can access it directly. If you do not know the URL, use the following directions.

#### Procedure from the web console

1. Log in to the OpenShift Container Platform web console as a user with cluster-admin rights. If you use Red Hat OpenShift Dedicated, you must have an account with the **dedicated-admin** role.
2. Navigate to **Networking → Routes**.

3. On the **Routes** page, select the control plane project, for example **tracing-system**, from the **Namespace** menu.  
The **Location** column displays the linked address for each route.
4. If necessary, use the filter to find the **jaeger** route. Click the route **Location** to launch the console.
5. Click **Log In With OpenShift**

### Procedure from the CLI

1. Log in to the OpenShift Container Platform CLI as a user with the **cluster-admin** role by running the following command. If you use Red Hat OpenShift Dedicated, you must have an account with the **dedicated-admin** role.

```
$ oc login --username=<NAMEOFUSER> https://<HOSTNAME>:6443
```

2. To query for details of the route using the command line, enter the following command. In this example, **tracing-system** is the control plane namespace.

```
$ export JAEGER_URL=$(oc get route -n tracing-system jaeger -o jsonpath='{.spec.host}')
```

3. Launch a browser and navigate to **https://<JAEGER\_URL>**, where **<JAEGER\_URL>** is the route that you discovered in the previous step.
4. Log in using the same user name and password that you use to access the OpenShift Container Platform console.
5. If you have added services to the service mesh and have generated traces, you can use the filters and **Find Traces** button to search your trace data.  
If you are validating the console installation, there is no trace data to display.

## 3.2.6. Customizing your deployment

### 3.2.6.1. Deployment best practices

- Red Hat OpenShift distributed tracing platform instance names must be unique. If you want to have multiple Red Hat OpenShift distributed tracing platform (Jaeger) instances and are using sidecar injected agents, then the Red Hat OpenShift distributed tracing platform (Jaeger) instances should have unique names, and the injection annotation should explicitly specify the Red Hat OpenShift distributed tracing platform (Jaeger) instance name the tracing data should be reported to.
- If you have a multitenant implementation and tenants are separated by namespaces, deploy a Red Hat OpenShift distributed tracing platform (Jaeger) instance to each tenant namespace.

For information about configuring persistent storage, see [Understanding persistent storage](#) and the appropriate configuration topic for your chosen storage option.

### 3.2.6.2. Distributed tracing default configuration options

The Jaeger custom resource (CR) defines the architecture and settings to be used when creating the distributed tracing platform (Jaeger) resources. You can modify these parameters to customize your distributed tracing platform (Jaeger) implementation to your business needs.

## Generic YAML example of the Jaeger CR

```

apiVersion: jaegertracing.io/v1
kind: Jaeger
metadata:
  name: name
spec:
  strategy: <deployment_strategy>
  allInOne:
    options: {}
    resources: {}
  agent:
    options: {}
    resources: {}
  collector:
    options: {}
    resources: {}
  sampling:
    options: {}
  storage:
    type:
    options: {}
  query:
    options: {}
    resources: {}
  ingester:
    options: {}
    resources: {}
  options: {}

```

Table 3.1. Jaeger parameters

Parameter	Description	Values	Default value
<b>apiVersion:</b>	API version to use when creating the object.	<b>jaegertracing.io/v1</b>	<b>jaegertracing.io/v1</b>
<b>kind:</b>	Defines the kind of Kubernetes object to create.	<b>jaeger</b>	
<b>metadata:</b>	Data that helps uniquely identify the object, including a <b>name</b> string, <b>UID</b> , and optional <b>namespace</b> .		OpenShift Container Platform automatically generates the <b>UID</b> and completes the <b>namespace</b> with the name of the project where the object is created.

Parameter	Description	Values	Default value
<b>name:</b>	Name for the object.	The name of your distributed tracing platform (Jaeger) instance.	<b>jaeger-all-in-one-inmemory</b>
<b>spec:</b>	Specification for the object to be created.	Contains all of the configuration parameters for your distributed tracing platform (Jaeger) instance. When a common definition for all Jaeger components is required, it is defined under the <b>spec</b> node. When the definition relates to an individual component, it is placed under the <b>spec/&lt;component&gt;</b> node.	N/A
<b>strategy:</b>	Jaeger deployment strategy	<b>allInOne</b> , <b>production</b> , or <b>streaming</b>	<b>allInOne</b>
<b>allInOne:</b>	Because the <b>allInOne</b> image deploys the Agent, Collector, Query, Ingester, and Jaeger UI in a single pod, configuration for this deployment must nest component configuration under the <b>allInOne</b> parameter.		
<b>agent:</b>	Configuration options that define the Agent.		
<b>collector:</b>	Configuration options that define the Jaeger Collector.		
<b>sampling:</b>	Configuration options that define the sampling strategies for tracing.		



Parameter	Description	Values	Default value
<b>storage:</b>	Configuration options that define the storage. All storage-related options must be placed under <b>storage</b> , rather than under the <b>allInOne</b> or other component options.		
<b>query:</b>	Configuration options that define the Query service.		
<b>ingester:</b>	Configuration options that define the Ingester service.		

The following example YAML is the minimum required to create a Red Hat OpenShift distributed tracing platform (Jaeger) deployment using the default settings.

#### Example minimum required dist-tracing-all-in-one.yaml

```
apiVersion: jaegertracing.io/v1
kind: Jaeger
metadata:
  name: jaeger-all-in-one-inmemory
```

#### 3.2.6.3. Jaeger Collector configuration options

The Jaeger Collector is the component responsible for receiving the spans that were captured by the tracer and writing them to persistent Elasticsearch storage when using the **production** strategy, or to AMQ Streams when using the **streaming** strategy.

The Collectors are stateless and thus many instances of Jaeger Collector can be run in parallel. Collectors require almost no configuration, except for the location of the Elasticsearch cluster.

**Table 3.2. Parameters used by the Operator to define the Jaeger Collector**

Parameter	Description	Values
<b>collector:</b> <b>replicas:</b>	Specifies the number of Collector replicas to create.	Integer, for example, <b>5</b>

**Table 3.3. Configuration parameters passed to the Collector**

Parameter	Description	Values
<code>spec: collector: options: {}</code>	Configuration options that define the Jaeger Collector.	
<code>options: collector: num-workers:</code>	The number of workers pulling from the queue.	Integer, for example, <b>50</b>
<code>options: collector: queue-size:</code>	The size of the Collector queue.	Integer, for example, <b>2000</b>
<code>options: kafka: producer: topic: jaeger-spans</code>	The <b>topic</b> parameter identifies the Kafka configuration used by the Collector to produce the messages, and the Ingestor to consume the messages.	Label for the producer.
<code>options: kafka: producer: brokers: my-cluster-kafka-brokers.kafka:9092</code>	Identifies the Kafka configuration used by the Collector to produce the messages. If brokers are not specified, and you have AMQ Streams 1.4.0+ installed, the Red Hat OpenShift distributed tracing platform (Jaeger) Operator will self-provision Kafka.	
<code>options: log-level:</code>	Logging level for the Collector.	Possible values: <b>debug, info, warn, error, fatal, panic.</b>

#### 3.2.6.4. Distributed tracing sampling configuration options

The Red Hat OpenShift distributed tracing platform (Jaeger) Operator can be used to define sampling strategies that will be supplied to tracers that have been configured to use a remote sampler.

While all traces are generated, only a few are sampled. Sampling a trace marks the trace for further processing and storage.



#### NOTE

This is not relevant if a trace was started by the Envoy proxy, as the sampling decision is made there. The Jaeger sampling decision is only relevant when the trace is started by an application using the client.

When a service receives a request that contains no trace context, the client starts a new trace, assigns it a random trace ID, and makes a sampling decision based on the currently installed sampling strategy. The sampling decision propagates to all subsequent requests in the trace so that other services are not making the sampling decision again.

distributed tracing platform (Jaeger) libraries support the following samplers:

- **Probabilistic** - The sampler makes a random sampling decision with the probability of sampling equal to the value of the **sampling.param** property. For example, using **sampling.param=0.1** samples approximately 1 in 10 traces.
- **Rate Limiting** - The sampler uses a leaky bucket rate limiter to ensure that traces are sampled with a certain constant rate. For example, using **sampling.param=2.0** samples requests with the rate of 2 traces per second.

Table 3.4. Jaeger sampling options

Parameter	Description	Values	Default value
spec: sampling: options: {} default_strategy:  service_strategy:	Configuration options that define the sampling strategies for tracing.		If you do not provide configuration, the Collectors will return the default probabilistic sampling policy with 0.001 (0.1%) probability for all services.
default_strategy: type: service_strategy: type:	Sampling strategy to use. See descriptions above.	Valid values are <b>probabilistic</b> , and <b>ratelimiting</b> .	<b>probabilistic</b>
default_strategy: param: service_strategy: param:	Parameters for the selected sampling strategy.	Decimal and integer values (0, .1, 1, 10)	1

This example defines a default sampling strategy that is probabilistic, with a 50% chance of the trace instances being sampled.

### Probabilistic sampling example

```
apiVersion: jaegertracing.io/v1
kind: Jaeger
metadata:
  name: with-sampling
spec:
  sampling:
    options:
      default_strategy:
```

```

    type: probabilistic
    param: 0.5
  service_strategies:
    - service: alpha
      type: probabilistic
      param: 0.8
    operation_strategies:
      - operation: op1
        type: probabilistic
        param: 0.2
      - operation: op2
        type: probabilistic
        param: 0.4
    - service: beta
      type: ratelimiting
      param: 5

```

If there are no user-supplied configurations, the distributed tracing platform (Jaeger) uses the following settings:

### Default sampling

```

spec:
  sampling:
    options:
      default_strategy:
        type: probabilistic
        param: 1

```

#### 3.2.6.5. Distributed tracing storage configuration options

You configure storage for the Collector, Ingester, and Query services under **spec.storage**. Multiple instances of each of these components can be provisioned as required for performance and resilience purposes.

**Table 3.5. General storage parameters used by the Red Hat OpenShift distributed tracing platform (Jaeger) Operator to define distributed tracing storage**

Parameter	Description	Values	Default value
-----------	-------------	--------	---------------

Parameter	Description	Values	Default value
<code>spec: storage: type:</code>	Type of storage to use for the deployment.	<b>memory</b> or <b>elasticsearch</b> . Memory storage is only appropriate for development, testing, demonstrations, and proof of concept environments as the data does not persist if the pod is shut down. For production environments distributed tracing platform (Jaeger) supports Elasticsearch for persistent storage.	<b>memory</b>
<code>storage: secretname:</code>	Name of the secret, for example <b>tracing-secret</b> .		N/A
<code>storage: options: {}</code>	Configuration options that define the storage.		

Table 3.6. Elasticsearch index cleaner parameters

Parameter	Description	Values	Default value
<code>storage: esIndexCleaner: enabled:</code>	When using Elasticsearch storage, by default a job is created to clean old traces from the index. This parameter enables or disables the index cleaner job.	<b>true/ false</b>	<b>true</b>
<code>storage: esIndexCleaner: numberOfDays:</code>	Number of days to wait before deleting an index.	Integer value	<b>7</b>

Parameter	Description	Values	Default value
<b>storage:</b> <b>esIndexCleaner:</b> <b>schedule:</b>	Defines the schedule for how often to clean the Elasticsearch index.	Cron expression	"55 23 * * *"

### 3.2.6.5.1. Auto-provisioning an Elasticsearch instance

When you deploy a Jaeger custom resource, the Red Hat OpenShift distributed tracing platform (Jaeger) Operator uses the OpenShift Elasticsearch Operator to create an Elasticsearch cluster based on the configuration provided in the **storage** section of the custom resource file. The Red Hat OpenShift distributed tracing platform (Jaeger) Operator will provision Elasticsearch if the following configurations are set:

- **spec.storage.type** is set to **elasticsearch**
- **spec.storage.elasticsearch.doNotProvision** set to **false**
- **spec.storage.options.es.server-urls** is not defined, that is, there is no connection to an Elasticsearch instance that was not provisioned by the Red Hat Elasticsearch Operator.

When provisioning Elasticsearch, the Red Hat OpenShift distributed tracing platform (Jaeger) Operator sets the Elasticsearch custom resource **name** to the value of **spec.storage.elasticsearch.name** from the Jaeger custom resource. If you do not specify a value for **spec.storage.elasticsearch.name**, the Operator uses **elasticsearch**.

### Restrictions

- You can have only one distributed tracing platform (Jaeger) with self-provisioned Elasticsearch instance per namespace. The Elasticsearch cluster is meant to be dedicated for a single distributed tracing platform (Jaeger) instance.
- There can be only one Elasticsearch per namespace.



### NOTE

If you already have installed Elasticsearch as part of OpenShift Logging, the Red Hat OpenShift distributed tracing platform (Jaeger) Operator can use the installed OpenShift Elasticsearch Operator to provision storage.

The following configuration parameters are for a *self-provisioned* Elasticsearch instance, that is an instance created by the Red Hat OpenShift distributed tracing platform (Jaeger) Operator using the OpenShift Elasticsearch Operator. You specify configuration options for self-provisioned Elasticsearch under **spec:storage:elasticsearch** in your configuration file.

**Table 3.7. Elasticsearch resource configuration parameters**

Parameter	Description	Values	Default value
<b>elasticsearch:</b> <b>properties:</b> <b>doNotProvision:</b>	Use to specify whether or not an Elasticsearch instance should be provisioned by the Red Hat OpenShift distributed tracing platform (Jaeger) Operator.	<b>true/false</b>	<b>true</b>
<b>elasticsearch:</b> <b>properties:</b> <b>name:</b>	Name of the Elasticsearch instance. The Red Hat OpenShift distributed tracing platform (Jaeger) Operator uses the Elasticsearch instance specified in this parameter to connect to Elasticsearch.	string	<b>elasticsearch</b>
<b>elasticsearch:</b> <b>nodeCount:</b>	Number of Elasticsearch nodes. For high availability use at least 3 nodes. Do not use 2 nodes as “split brain” problem can happen.	Integer value. For example, Proof of concept = 1, Minimum deployment =3	3
<b>elasticsearch:</b> <b>resources:</b> <b>requests:</b> <b>cpu:</b>	Number of central processing units for requests, based on your environment’s configuration.	Specified in cores or millicores, for example, 200m, 0.5, 1. For example, Proof of concept = 500m, Minimum deployment =1	1
<b>elasticsearch:</b> <b>resources:</b> <b>requests:</b> <b>memory:</b>	Available memory for requests, based on your environment’s configuration.	Specified in bytes, for example, 200Ki, 50Mi, 5Gi. For example, Proof of concept = 1Gi, Minimum deployment = 16Gi*	16Gi
<b>elasticsearch:</b> <b>resources:</b> <b>limits:</b> <b>cpu:</b>	Limit on number of central processing units, based on your environment’s configuration.	Specified in cores or millicores, for example, 200m, 0.5, 1. For example, Proof of concept = 500m, Minimum deployment =1	

Parameter	Description	Values	Default value
<code>elasticsearch:resources:limits:memory:</code>	Available memory limit based on your environment's configuration.	Specified in bytes, for example, 200Ki, 50Mi, 5Gi. For example, Proof of concept = 1Gi, Minimum deployment = 16Gi*	
<code>elasticsearch:redundancyPolicy:</code>	Data replication policy defines how Elasticsearch shards are replicated across data nodes in the cluster. If not specified, the Red Hat OpenShift distributed tracing platform (Jaeger) Operator automatically determines the most appropriate replication based on number of nodes.	<b>ZeroRedundancy</b> (no replica shards), <b>SingleRedundancy</b> (one replica shard), <b>MultipleRedundancy</b> (each index is spread over half of the Data nodes), <b>FullRedundancy</b> (each index is fully replicated on every Data node in the cluster).	
<code>elasticsearch:useCertManagement:</code>	Use to specify whether or not distributed tracing platform (Jaeger) should use the certificate management feature of the Red Hat Elasticsearch Operator. This feature was added to logging subsystem for Red Hat OpenShift 5.2 in OpenShift Container Platform 4.7 and is the preferred setting for new Jaeger deployments.	<b>true/false</b>	<b>true</b>
	*Each Elasticsearch node can operate with a lower memory setting though this is NOT recommended for production deployments. For production use, you should have no less than 16Gi allocated to each pod by default, but preferably allocate as much as you can, up to 64Gi per pod.		

## Production storage example

```

apiVersion: jaegertracing.io/v1
kind: Jaeger
metadata:
  name: simple-prod
spec:
  strategy: production

```



```

storage:
  type: elasticsearch
  elasticsearch:
    nodeCount: 3
  resources:
    requests:
      cpu: 1
      memory: 16Gi
  limits:
    memory: 16Gi

```

### Storage example with persistent storage:

```

apiVersion: jaegertracing.io/v1
kind: Jaeger
metadata:
  name: simple-prod
spec:
  strategy: production
  storage:
    type: elasticsearch
    elasticsearch:
      nodeCount: 1
    storage: 1
      storageClassName: gp2
      size: 5Gi
  resources:
    requests:
      cpu: 200m
      memory: 4Gi
  limits:
    memory: 4Gi
  redundancyPolicy: ZeroRedundancy

```

- 1 Persistent storage configuration. In this case AWS **gp2** with **5Gi** size. When no value is specified, distributed tracing platform (Jaeger) uses **emptyDir**. The OpenShift Elasticsearch Operator provisions **PersistentVolumeClaim** and **PersistentVolume** which are not removed with distributed tracing platform (Jaeger) instance. You can mount the same volumes if you create a distributed tracing platform (Jaeger) instance with the same name and namespace.

#### 3.2.6.5.2. Connecting to an existing Elasticsearch instance

You can use an existing Elasticsearch cluster for storage with distributed tracing platform. An existing Elasticsearch cluster, also known as an *external* Elasticsearch instance, is an instance that was not installed by the Red Hat OpenShift distributed tracing platform (Jaeger) Operator or by the Red Hat Elasticsearch Operator.

When you deploy a Jaeger custom resource, the Red Hat OpenShift distributed tracing platform (Jaeger) Operator will not provision Elasticsearch if the following configurations are set:

- **spec.storage.elasticsearch.doNotProvision** set to **true**
- **spec.storage.options.es.server-urls** has a value

- **spec.storage.elasticsearch.name** has a value, or if the Elasticsearch instance name is **elasticsearch**.

The Red Hat OpenShift distributed tracing platform (Jaeger) Operator uses the Elasticsearch instance specified in **spec.storage.elasticsearch.name** to connect to Elasticsearch.

## Restrictions

- You cannot share or reuse a OpenShift Container Platform logging Elasticsearch instance with distributed tracing platform (Jaeger). The Elasticsearch cluster is meant to be dedicated for a single distributed tracing platform (Jaeger) instance.



## NOTE

Red Hat does not provide support for your external Elasticsearch instance. You can review the tested integrations matrix on the [Customer Portal](#).

The following configuration parameters are for an already existing Elasticsearch instance, also known as an *external* Elasticsearch instance. In this case, you specify configuration options for Elasticsearch under **spec:storage:options:es** in your custom resource file.

**Table 3.8. General ES configuration parameters**

Parameter	Description	Values	Default value
<b>es:</b> server-urls:	URL of the Elasticsearch instance.	The fully-qualified domain name of the Elasticsearch server.	<a href="http://elasticsearch.&lt;namespace&gt;.svc:9200">http://elasticsearch.&lt;namespace&gt;.svc:9200</a>
<b>es:</b> max-doc-count:	The maximum document count to return from an Elasticsearch query. This will also apply to aggregations. If you set both <b>es.max-doc-count</b> and <b>es.max-num-spans</b> , Elasticsearch will use the smaller value of the two.		10000
<b>es:</b> max-num-spans:	<b>[Deprecated - Will be removed in a future release, use <b>es.max-doc-count</b> instead.]</b> The maximum number of spans to fetch at a time, per query, in Elasticsearch. If you set both <b>es.max-num-spans</b> and <b>es.max-doc-count</b> , Elasticsearch will use the smaller value of the two.		10000

Parameter	Description	Values	Default value
<b>es:</b> max-span-age:	The maximum lookback for spans in Elasticsearch.		72h0m0s
<b>es:</b> sniffer:	The sniffer configuration for Elasticsearch. The client uses the sniffing process to find all nodes automatically. Disabled by default.	<b>true/ false</b>	<b>false</b>
<b>es:</b> sniffer-tls-enabled:	Option to enable TLS when sniffing an Elasticsearch Cluster. The client uses the sniffing process to find all nodes automatically. Disabled by default	<b>true/ false</b>	<b>false</b>
<b>es:</b> timeout:	Timeout used for queries. When set to zero there is no timeout.		0s
<b>es:</b> username:	The username required by Elasticsearch. The basic authentication also loads CA if it is specified. See also <b>es.password</b> .		
<b>es:</b> password:	The password required by Elasticsearch. See also, <b>es.username</b> .		
<b>es:</b> version:	The major Elasticsearch version. If not specified, the value will be auto-detected from Elasticsearch.		0

Table 3.9. ES data replication parameters

Parameter	Description	Values	Default value
<b>es:</b> num-replicas:	The number of replicas per index in Elasticsearch.		1

Parameter	Description	Values	Default value
<code>es: num-shards:</code>	The number of shards per index in Elasticsearch.		5

Table 3.10. ES index configuration parameters

Parameter	Description	Values	Default value
<code>es: create-index-templates:</code>	Automatically create index templates at application startup when set to <b>true</b> . When templates are installed manually, set to <b>false</b> .	<b>true/ false</b>	<b>true</b>
<code>es: index-prefix:</code>	Optional prefix for distributed tracing platform (Jaeger) indices. For example, setting this to "production" creates indices named "production-tracing-*".		

Table 3.11. ES bulk processor configuration parameters

Parameter	Description	Values	Default value
<code>es: bulk: actions:</code>	The number of requests that can be added to the queue before the bulk processor decides to commit updates to disk.		1000
<code>es: bulk: flush-interval:</code>	A <b>time.Duration</b> after which bulk requests are committed, regardless of other thresholds. To disable the bulk processor flush interval, set this to zero.		200ms
<code>es: bulk: size:</code>	The number of bytes that the bulk requests can take up before the bulk processor decides to commit updates to disk.		5000000

Parameter	Description	Values	Default value
<code>es: bulk: workers:</code>	The number of workers that are able to receive and commit bulk requests to Elasticsearch.		1

Table 3.12. ES TLS configuration parameters

Parameter	Description	Values	Default value
<code>es: tls: ca:</code>	Path to a TLS Certification Authority (CA) file used to verify the remote servers.		Will use the system truststore by default.
<code>es: tls: cert:</code>	Path to a TLS Certificate file, used to identify this process to the remote servers.		
<code>es: tls: enabled:</code>	Enable transport layer security (TLS) when talking to the remote servers. Disabled by default.	<b>true/ false</b>	<b>false</b>
<code>es: tls: key:</code>	Path to a TLS Private Key file, used to identify this process to the remote servers.		
<code>es: tls: server-name:</code>	Override the expected TLS server name in the certificate of the remote servers.		
<code>es: token-file:</code>	Path to a file containing the bearer token. This flag also loads the Certification Authority (CA) file if it is specified.		

Table 3.13. ES archive configuration parameters

Parameter	Description	Values	Default value
<code>es-archive: bulk: actions:</code>	The number of requests that can be added to the queue before the bulk processor decides to commit updates to disk.		0
<code>es-archive: bulk: flush-interval:</code>	A <b>time.Duration</b> after which bulk requests are committed, regardless of other thresholds. To disable the bulk processor flush interval, set this to zero.		0s
<code>es-archive: bulk: size:</code>	The number of bytes that the bulk requests can take up before the bulk processor decides to commit updates to disk.		0
<code>es-archive: bulk: workers:</code>	The number of workers that are able to receive and commit bulk requests to Elasticsearch.		0
<code>es-archive: create-index- templates:</code>	Automatically create index templates at application startup when set to <b>true</b> . When templates are installed manually, set to <b>false</b> .	<b>true/ false</b>	<b>false</b>
<code>es-archive: enabled:</code>	Enable extra storage.	<b>true/ false</b>	<b>false</b>
<code>es-archive: index-prefix:</code>	Optional prefix for distributed tracing platform (Jaeger) indices. For example, setting this to "production" creates indices named "production-tracing-*".		

Parameter	Description	Values	Default value
<b>es-archive:</b> max-doc-count:	The maximum document count to return from an Elasticsearch query. This will also apply to aggregations.		0
<b>es-archive:</b> max-num-spans:	[ <b>Deprecated</b> - Will be removed in a future release, use <b>es-archive.max-doc-count</b> instead.] The maximum number of spans to fetch at a time, per query, in Elasticsearch.		0
<b>es-archive:</b> max-span-age:	The maximum lookback for spans in Elasticsearch.		0s
<b>es-archive:</b> num-replicas:	The number of replicas per index in Elasticsearch.		0
<b>es-archive:</b> num-shards:	The number of shards per index in Elasticsearch.		0
<b>es-archive:</b> password:	The password required by Elasticsearch. See also, <b>es.username</b> .		
<b>es-archive:</b> server-urls:	The comma-separated list of Elasticsearch servers. Must be specified as fully qualified URLs, for example, <b>http://localhost:9200</b> .		
<b>es-archive:</b> sniffer:	The sniffer configuration for Elasticsearch. The client uses the sniffing process to find all nodes automatically. Disabled by default.	<b>true/ false</b>	<b>false</b>

Parameter	Description	Values	Default value
<b>es-archive:</b> <b>sniffer-tls-</b> <b>enabled:</b>	Option to enable TLS when sniffing an Elasticsearch Cluster. The client uses the sniffing process to find all nodes automatically. Disabled by default.	<b>true/ false</b>	<b>false</b>
<b>es-archive:</b> <b>timeout:</b>	Timeout used for queries. When set to zero there is no timeout.		0s
<b>es-archive:</b> <b>tls:</b> <b>ca:</b>	Path to a TLS Certification Authority (CA) file used to verify the remote servers.		Will use the system truststore by default.
<b>es-archive:</b> <b>tls:</b> <b>cert:</b>	Path to a TLS Certificate file, used to identify this process to the remote servers.		
<b>es-archive:</b> <b>tls:</b> <b>enabled:</b>	Enable transport layer security (TLS) when talking to the remote servers. Disabled by default.	<b>true/ false</b>	<b>false</b>
<b>es-archive:</b> <b>tls:</b> <b>key:</b>	Path to a TLS Private Key file, used to identify this process to the remote servers.		
<b>es-archive:</b> <b>tls:</b> <b>server-name:</b>	Override the expected TLS server name in the certificate of the remote servers.		
<b>es-archive:</b> <b>token-file:</b>	Path to a file containing the bearer token. This flag also loads the Certification Authority (CA) file if it is specified.		



Parameter	Description	Values	Default value
<code>es-archive:username:</code>	The username required by Elasticsearch. The basic authentication also loads CA if it is specified. See also <b><code>es-archive.password</code></b> .		
<code>es-archive:version:</code>	The major Elasticsearch version. If not specified, the value will be auto-detected from Elasticsearch.		0

### Storage example with volume mounts

```

apiVersion: jaegertracing.io/v1
kind: Jaeger
metadata:
  name: simple-prod
spec:
  strategy: production
  storage:
    type: elasticsearch
    options:
      es:
        server-urls: https://quickstart-es-http.default.svc:9200
        index-prefix: my-prefix
        tls:
          ca: /es/certificates/ca.crt
      secretName: tracing-secret
  volumeMounts:
    - name: certificates
      mountPath: /es/certificates/
      readOnly: true
  volumes:
    - name: certificates
      secret:
        secretName: quickstart-es-http-certs-public

```

The following example shows a Jaeger CR using an external Elasticsearch cluster with TLS CA certificate mounted from a volume and user/password stored in a secret.

### External Elasticsearch example

```

apiVersion: jaegertracing.io/v1
kind: Jaeger
metadata:
  name: simple-prod
spec:
  strategy: production
  storage:

```

```

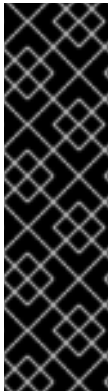
type: elasticsearch
options:
  es:
    server-urls: https://quickstart-es-http.default.svc:9200 ❶
    index-prefix: my-prefix
    tls: ❷
      ca: /es/certificates/ca.crt
    secretName: tracing-secret ❸
volumeMounts: ❹
  - name: certificates
    mountPath: /es/certificates/
    readOnly: true
volumes:
  - name: certificates
    secret:
      secretName: quickstart-es-http-certs-public

```

- ❶ URL to Elasticsearch service running in default namespace.
- ❷ TLS configuration. In this case only CA certificate, but it can also contain es.tls.key and es.tls.cert when using mutual TLS.
- ❸ Secret which defines environment variables ES\_PASSWORD and ES\_USERNAME. Created by `kubectrl create secret generic tracing-secret --from-literal=ES_PASSWORD=changeme --from-literal=ES_USERNAME=elastic`
- ❹ Volume mounts and volumes which are mounted into all storage components.

### 3.2.6.6. Managing certificates with Elasticsearch

You can create and manage certificates using the Red Hat Elasticsearch Operator. Managing certificates using the Red Hat Elasticsearch Operator also lets you use a single Elasticsearch cluster with multiple Jaeger Collectors.



#### IMPORTANT

Managing certificates with Elasticsearch 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](#).

Starting with version 2.4, the Red Hat OpenShift distributed tracing platform (Jaeger) Operator delegates certificate creation to the Red Hat Elasticsearch Operator by using the following annotations in the Elasticsearch custom resource:

- `logging.openshift.io/elasticsearch-cert-management: "true"`
- `logging.openshift.io/elasticsearch-cert.jaeger-<shared-es-node-name>: "user.jaeger"`

- **logging.openshift.io/elasticsearch-cert.curator-<shared-es-node-name>:**  
"system.logging.curator"

Where the **<shared-es-node-name>** is the name of the Elasticsearch node. For example, if you create an Elasticsearch node named **custom-es**, your custom resource might look like the following example.

### Example Elasticsearch CR showing annotations

```
apiVersion: logging.openshift.io/v1
kind: Elasticsearch
metadata:
  annotations:
    logging.openshift.io/elasticsearch-cert-management: "true"
    logging.openshift.io/elasticsearch-cert.jaeger-custom-es: "user.jaeger"
    logging.openshift.io/elasticsearch-cert.curator-custom-es: "system.logging.curator"
  name: custom-es
spec:
  managementState: Managed
  nodeSpec:
    resources:
      limits:
        memory: 16Gi
      requests:
        cpu: 1
        memory: 16Gi
    nodes:
      - nodeCount: 3
        proxyResources: {}
        resources: {}
        roles:
          - master
          - client
          - data
        storage: {}
  redundancyPolicy: ZeroRedundancy
```

### Prerequisites

- OpenShift Container Platform 4.7
- logging subsystem for Red Hat OpenShift 5.2
- The Elasticsearch node and the Jaeger instances must be deployed in the same namespace. For example, **tracing-system**.

You enable certificate management by setting **spec.storage.elasticsearch.useCertManagement** to **true** in the Jaeger custom resource.

### Example showing useCertManagement

```
apiVersion: jaegertracing.io/v1
kind: Jaeger
metadata:
  name: jaeger-prod
spec:
```

```

strategy: production
storage:
  type: elasticsearch
  elasticsearch:
    name: custom-es
    doNotProvision: true
    useCertManagement: true

```

The Red Hat OpenShift distributed tracing platform (Jaeger) Operator sets the Elasticsearch custom resource **name** to the value of **spec.storage.elasticsearch.name** from the Jaeger custom resource when provisioning Elasticsearch.

The certificates are provisioned by the Red Hat Elasticsearch Operator and the Red Hat OpenShift distributed tracing platform (Jaeger) Operator injects the certificates.

### 3.2.6.7. Query configuration options

Query is a service that retrieves traces from storage and hosts the user interface to display them.

**Table 3.14. Parameters used by the Red Hat OpenShift distributed tracing platform (Jaeger) Operator to define Query**

Parameter	Description	Values	Default value
spec: query: replicas:	Specifies the number of Query replicas to create.	Integer, for example, <b>2</b>	

**Table 3.15. Configuration parameters passed to Query**

Parameter	Description	Values	Default value
spec: query: options: {}	Configuration options that define the Query service.		
options: log-level:	Logging level for Query.	Possible values: <b>debug</b> , <b>info</b> , <b>warn</b> , <b>error</b> , <b>fatal</b> , <b>panic</b> .	

Parameter	Description	Values	Default value
<pre>options:   query:     base-path:</pre>	The base path for all jaeger-query HTTP routes can be set to a non-root value, for example, <b>/jaeger</b> would cause all UI URLs to start with <b>/jaeger</b> . This can be useful when running jaeger-query behind a reverse proxy.	/<path>	

### Sample Query configuration

```
apiVersion: jaegertracing.io/v1
kind: "Jaeger"
metadata:
  name: "my-jaeger"
spec:
  strategy: allInOne
  allInOne:
    options:
      log-level: debug
    query:
      base-path: /jaeger
```

#### 3.2.6.8. Ingester configuration options

Ingester is a service that reads from a Kafka topic and writes to the Elasticsearch storage backend. If you are using the **allInOne** or **production** deployment strategies, you do not need to configure the Ingester service.

Table 3.16. Jaeger parameters passed to the Ingester

Parameter	Description	Values
<pre>spec:   ingester:     options: {}</pre>	Configuration options that define the Ingester service.	
<pre>options:   deadlockInterval:</pre>	Specifies the interval, in seconds or minutes, that the Ingester must wait for a message before terminating. The deadlock interval is disabled by default (set to <b>0</b> ), to avoid terminating the Ingester when no messages arrive during system initialization.	Minutes and seconds, for example, <b>1m0s</b> . Default value is <b>0</b> .

Parameter	Description	Values
<pre>options:   kafka:     consumer:       topic:</pre>	The <b>topic</b> parameter identifies the Kafka configuration used by the collector to produce the messages, and the Ingestor to consume the messages.	Label for the consumer. For example, <b>jaeger-spans</b> .
<pre>options:   kafka:     consumer:       brokers:</pre>	Identifies the Kafka configuration used by the Ingestor to consume the messages.	Label for the broker, for example, <b>my-cluster-kafka-brokers.kafka:9092</b> .
<pre>options:   log-level:</pre>	Logging level for the Ingestor.	Possible values: <b>debug, info, warn, error, fatal, dpanic, panic</b> .

## Streaming Collector and Ingestor example

```
apiVersion: jaegertracing.io/v1
kind: Jaeger
metadata:
  name: simple-streaming
spec:
  strategy: streaming
  collector:
    options:
      kafka:
        producer:
          topic: jaeger-spans
          brokers: my-cluster-kafka-brokers.kafka:9092
  ingestor:
    options:
      kafka:
        consumer:
          topic: jaeger-spans
          brokers: my-cluster-kafka-brokers.kafka:9092
      ingestor:
        deadlockInterval: 5
  storage:
    type: elasticsearch
    options:
      es:
        server-urls: http://elasticsearch:9200
```

### 3.2.7. Injecting sidecars

The Red Hat OpenShift distributed tracing platform (Jaeger) relies on a proxy sidecar within the application's pod to provide the Agent. The Red Hat OpenShift distributed tracing platform (Jaeger)

Operator can inject Agent sidecars into deployment workloads. You can enable automatic sidecar injection or manage it manually.

### 3.2.7.1. Automatically injecting sidecars

The Red Hat OpenShift distributed tracing platform (Jaeger) Operator can inject Jaeger Agent sidecars into deployment workloads. To enable automatic injection of sidecars, add the **sidecar.jaegertracing.io/inject** annotation set to either the string **true** or to the distributed tracing platform (Jaeger) instance name that is returned by running **\$ oc get jaegers**. When you specify **true**, there must be only a single distributed tracing platform (Jaeger) instance for the same namespace as the deployment. Otherwise, the Operator is unable to determine which distributed tracing platform (Jaeger) instance to use. A specific distributed tracing platform (Jaeger) instance name on a deployment has a higher precedence than **true** applied on its namespace.

The following snippet shows a simple application that will inject a sidecar, with the agent pointing to the single distributed tracing platform (Jaeger) instance available in the same namespace:

#### Automatic sidecar injection example

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
  annotations:
    "sidecar.jaegertracing.io/inject": "true" 1
spec:
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: myapp
          image: acme/myapp:myversion
```

1 Set to either the string **true** or to the Jaeger instance name.

When the sidecar is injected, the agent can then be accessed at its default location on **localhost**.

### 3.2.7.2. Manually injecting sidecars

The Red Hat OpenShift distributed tracing platform (Jaeger) Operator can only automatically inject Jaeger Agent sidecars into Deployment workloads. For controller types other than **Deployments**, such as **StatefulSets** and **DaemonSets**, you can manually define the Jaeger agent sidecar in your specification.

The following snippet shows the manual definition you can include in your containers section for a Jaeger agent sidecar:

#### Sidecar definition example for a StatefulSet

```

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: example-statefulset
  namespace: example-ns
  labels:
    app: example-app
spec:

  spec:
    containers:
      - name: example-app
        image: acme/myapp:myversion
        ports:
          - containerPort: 8080
            protocol: TCP
      - name: jaeger-agent
        image: registry.redhat.io/distributed-tracing/jaeger-agent-rhel7:<version>
        # The agent version must match the Operator version
        imagePullPolicy: IfNotPresent
        ports:
          - containerPort: 5775
            name: zk-compact-trft
            protocol: UDP
          - containerPort: 5778
            name: config-rest
            protocol: TCP
          - containerPort: 6831
            name: jg-compact-trft
            protocol: UDP
          - containerPort: 6832
            name: jg-binary-trft
            protocol: UDP
          - containerPort: 14271
            name: admin-http
            protocol: TCP
        args:
          - --reporter.grpc.host-port=dns:///jaeger-collector-headless.example-ns:14250
          - --reporter.type=grpc

```

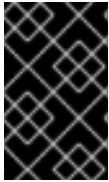
The agent can then be accessed at its default location on localhost.

### 3.3. UPDATING THE DISTRIBUTED TRACING PLATFORM JAEGER

Operator Lifecycle Manager (OLM) controls the installation, upgrade, and role-based access control (RBAC) of Operators in a cluster. The OLM runs by default in OpenShift Container Platform. OLM queries for available Operators as well as upgrades for installed Operators.

During an update, the Red Hat OpenShift distributed tracing platform Operators upgrade the managed distributed tracing platform instances to the version associated with the Operator. Whenever a new version of the Red Hat OpenShift distributed tracing platform (Jaeger) Operator is installed, all the distributed tracing platform (Jaeger) application instances managed by the Operator are upgraded to the Operator's version. For example, after upgrading the Operator from 1.10 installed to 1.11, the Operator scans for running distributed tracing platform (Jaeger) instances and upgrades them to 1.11 as well.





## IMPORTANT

If you have not already updated your OpenShift Elasticsearch Operator as described in [Updating OpenShift Logging](#), complete that update before updating your Red Hat OpenShift distributed tracing platform (Jaeger) Operator.

### 3.3.1. Additional resources

- [Operator Lifecycle Manager concepts and resources](#)
- [Updating installed Operators](#)
- [Updating OpenShift Logging](#)

## 3.4. REMOVING THE DISTRIBUTED TRACING PLATFORM JAEGER

The steps for removing Red Hat OpenShift distributed tracing platform from an OpenShift Container Platform cluster are as follows:

1. Shut down any Red Hat OpenShift distributed tracing platform pods.
2. Remove any Red Hat OpenShift distributed tracing platform instances.
3. Remove the Red Hat OpenShift distributed tracing platform (Jaeger) Operator.
4. Remove the Red Hat OpenShift distributed tracing data collection Operator.

### 3.4.1. Removing a distributed tracing platform (Jaeger) instance by using the web console

You can remove a distributed tracing platform (Jaeger) instance in the **Administrator** view of the web console.



## WARNING


When deleting an instance that uses in-memory storage, all data is irretrievably lost. Data stored in persistent storage such as Elasticsearch is not deleted when a Red Hat OpenShift distributed tracing platform (Jaeger) instance is removed.

### Prerequisites

- You are logged in to the web console as a cluster administrator with the **cluster-admin** role.

### Procedure

1. Log in to the OpenShift Container Platform web console.
2. Navigate to **Operators → Installed Operators**.

3. Select the name of the project where the Operators are installed from the **Project** menu, for example, **openshift-operators**.
4. Click the Red Hat OpenShift distributed tracing platform (Jaeger) Operator.
5. Click the **Jaeger** tab.
6. Click the Options menu  next to the instance you want to delete and select **Delete Jaeger**.
7. In the confirmation message, click **Delete**.

### 3.4.2. Removing a distributed tracing platform (Jaeger) instance by using the CLI

You can remove a distributed tracing platform (Jaeger) 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. Log in with the OpenShift CLI (**oc**) by running the following command:

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

2. To display the distributed tracing platform (Jaeger) instances, run the following command:

```
$ oc get deployments -n <jaeger-project>
```

For example,

```
$ oc get deployments -n openshift-operators
```

The names of Operators have the suffix **-operator**. The following example shows two Red Hat OpenShift distributed tracing platform (Jaeger) Operators and four distributed tracing platform (Jaeger) instances:

```
$ oc get deployments -n openshift-operators
```

You will see output similar to the following:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
elasticsearch-operator	1/1	1	1	93m
jaeger-operator	1/1	1	1	49m
jaeger-test	1/1	1	1	7m23s
jaeger-test2	1/1	1	1	6m48s
tracing1	1/1	1	1	7m8s
tracing2	1/1	1	1	35m

- To remove an instance of distributed tracing platform (Jaeger), run the following command:

```
$ oc delete jaeger <deployment-name> -n <jaeger-project>
```

For example:

```
$ oc delete jaeger tracing2 -n openshift-operators
```

- To verify the deletion, run the **oc get deployments** command again:

```
$ oc get deployments -n <jaeger-project>
```

For example:

```
$ oc get deployments -n openshift-operators
```

You will see generated output that is similar to the following example:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
elasticsearch-operator	1/1	1	1	94m
jaeger-operator	1/1	1	1	50m
jaeger-test	1/1	1	1	8m14s
jaeger-test2	1/1	1	1	7m39s
tracing1	1/1	1	1	7m59s

### 3.4.3. Removing the Red Hat OpenShift distributed tracing platform Operators

#### Procedure

- Follow the instructions in [Deleting Operators from a cluster](#) to remove the Red Hat OpenShift distributed tracing platform (Jaeger) Operator.
- Optional: After the Red Hat OpenShift distributed tracing platform (Jaeger) Operator has been removed, remove the OpenShift Elasticsearch Operator.

## CHAPTER 4. DISTRIBUTED TRACING PLATFORM (TEMPO)

### 4.1. INSTALLING THE DISTRIBUTED TRACING PLATFORM (TEMPO)



#### IMPORTANT

The Tempo Operator 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](#).

Installing the distributed tracing platform (Tempo) involves the following steps:

1. Setting up supported object storage.
2. Installing the Tempo Operator.
3. Creating a secret for the object storage credentials.
4. Creating a namespace for a TempoStack instance.
5. Creating a **TempoStack** custom resource to deploy at least one TempoStack instance.

#### 4.1.1. Installing the distributed tracing platform (Tempo) from the web console

You can install the distributed tracing platform (Tempo) from the **Administrator** view of the web console.

##### Prerequisites

- You are logged in to the OpenShift Container Platform 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.
- You are using a supported provider of object storage: [Red Hat OpenShift Data Foundation](#), [MinIO](#), [Amazon S3](#), [Azure Blob Storage](#), [Google Cloud Storage](#).

##### Procedure

1. Install the Tempo Operator:
  - a. Go to **Operators** → **OperatorHub** and search for **Tempo Operator**.
  - b. Select the **Tempo Operator** that is **OpenShift Operator for Tempo** → **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-tempo-operator
- Update approval → Automatic

- c. In the **Details** tab of the page of the installed Operator, under **ClusterServiceVersion details**, verify that the installation **Status** is **Succeeded**.
2. Create a secret for your object storage bucket: go to **Workloads → Secrets → Create → From YAML**.

Table 4.1. Required secret parameters

Storage provider
Secret parameters
<a href="#">Red Hat OpenShift Data Foundation</a>
<b>name:</b> tempostack-dev-odf # example <b>bucket:</b> <bucket_name> # requires an ObjectBucketClaim <b>endpoint:</b> https://s3.openshift-storage.svc <b>access_key_id:</b> <data_foundation_access_key_id> <b>access_key_secret:</b> <data_foundation_access_key_secret>
MinIO
See <a href="#">MinIO Operator</a> . <b>name:</b> tempostack-dev-minio # example <b>bucket:</b> <minio_bucket_name> # <a href="#">MinIO documentation</a> <b>endpoint:</b> <minio_bucket_endpoint> <b>access_key_id:</b> <minio_access_key_id> <b>access_key_secret:</b> <minio_access_key_secret>
Amazon S3

Storage provider
<p><b>name:</b> <code>tempostack-dev-s3</code> # example</p> <p><b>bucket:</b> <code>&lt;s3_bucket_name&gt;</code> # <a href="#">Amazon S3 documentation</a></p> <p><b>endpoint:</b> <code>&lt;s3_bucket_endpoint&gt;</code></p> <p><b>access_key_id:</b> <code>&lt;s3_access_key_id&gt;</code></p> <p><b>access_key_secret:</b> <code>&lt;s3_access_key_secret&gt;</code></p>
Microsoft Azure Blob Storage
<p><b>name:</b> <code>tempostack-dev-azure</code> # example</p> <p><b>container:</b> <code>&lt;azure_blob_storage_container_name&gt;</code> # <a href="#">Microsoft Azure documentation</a></p> <p><b>account_name:</b> <code>&lt;azure_blob_storage_account_name&gt;</code></p> <p><b>account_key:</b> <code>&lt;azure_blob_storage_account_key&gt;</code></p>
Google Cloud Storage on Google Cloud Platform (GCP)
<p><b>name:</b> <code>tempostack-dev-gcs</code> # example</p> <p><b>bucketname:</b> <code>&lt;google_cloud_storage_bucket_name&gt;</code> # requires a <a href="#">bucket</a> created in a <a href="#">GCP project</a></p> <p><b>key.json:</b> <code>&lt;path/to/key.json&gt;</code> # requires a <a href="#">service account</a> in the bucket's GCP project for GCP authentication</p>

### Example secret for Amazon S3 and MinIO storage

```

apiVersion: v1
kind: Secret
metadata:
  name: minio-test
stringData:
  endpoint: http://minio.minio.svc:9000
  bucket: tempo
  access_key_id: tempo
  access_key_secret: <secret>
type: Opaque

```

3. Create a project of your choice for the **TempoStack** instance that you will create in the next step: go to **Home** → **Projects** → **Create Project**.
4. Create a **TempoStack** instance.

**NOTE**

You can create multiple **TempoStack** instances in separate projects on the same cluster.

- a. Go to **Operators → Installed Operators**.
- b. Select **TempoStack → Create TempoStack → YAML view**.
- c. In the **YAML view**, customize the **TempoStack** custom resource (CR):

```
apiVersion: tempo.grafana.com/v1alpha1
kind: TempoStack
metadata:
  name: sample
  namespace: <project_of_tempostack_instance>
spec:
  storageSize: 1Gi
  storage:
    secret:
      name: <secret-name> 1
      type: <secret-provider> 2
  template:
    queryFrontend:
      jaegerQuery:
        enabled: true
    ingress:
      route:
        termination: edge
      type: route
```

<sup>1</sup>

The value of the **name** in the **metadata** of the secret.

<sup>2</sup>

The accepted values are **azure** for Azure Blob Storage; **gcs** for Google Cloud Storage; and **s3** for Amazon S3, MinIO, or Red Hat OpenShift Data Foundation.

### Example of a TempoStack CR for AWS S3 and MinIO storage

```
apiVersion: tempo.grafana.com/v1alpha1
kind: TempoStack
metadata:
  name: simplest
  namespace: <project_of_tempostack_instance>
spec:
  storageSize: 1Gi
  storage:
    secret:
      name: minio-test
      type: s3
  resources:
    total:
      limits:
        memory: 2Gi
        cpu: 2000m
```

```

template:
  queryFrontend:
    jaegerQuery:
      enabled: true
    ingress:
      route:
        termination: edge
        type: route

```

The stack deployed in this example is configured to receive Jaeger Thrift over HTTP and OpenTelemetry Protocol (OTLP), which permits visualizing the data with the Jaeger UI.

- d. Select **Create**.

## Verification

1. Use the **Project**: dropdown list to select the project of the **TempoStack** instance.
2. Go to **Operators** → **Installed Operators** to verify that the **Status** of the **TempoStack** instance is **Condition: Ready**.
3. Go to **Workloads** → **Pods** to verify that all the component pods of the **TempoStack** instance are running.
4. Access the Tempo console:
  - a. Go to **Networking** → **Routes** and **Ctrl+F** to search for **tempo**.
  - b. In the **Location** column, open the URL to access the Tempo console.
  - c. Select **Log In With OpenShift** to use your cluster administrator credentials for the web console.



### NOTE

The Tempo console initially shows no trace data following the Tempo console installation.

## 4.1.2. Installing the distributed tracing platform (Tempo) by using the CLI

You can install the distributed tracing platform (Tempo) 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>
```



- You are using a supported provider of object storage: [Red Hat OpenShift Data Foundation](#) , [MinIO](#), [Amazon S3](#), [Azure Blob Storage](#), [Google Cloud Storage](#).

## Procedure

### 1. Install the Tempo Operator:

- a. Create a project for the Tempo Operator by running the following command:

```
$ oc apply -f - << EOF
apiVersion: project.openshift.io/v1
kind: Project
metadata:
  labels:
    kubernetes.io/metadata.name: openshift-tempo-operator
    openshift.io/cluster-monitoring: "true"
  name: openshift-tempo-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-tempo-operator
  namespace: openshift-tempo-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: tempo-product
  namespace: openshift-tempo-operator
spec:
  channel: stable
  installPlanApproval: Automatic
  name: tempo-product
  source: redhat-operators
  sourceNamespace: openshift-marketplace
EOF
```

- d. Check the operator status by running the following command:

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

### 2. Create a secret for your object storage bucket by running one of the following commands:

- To create a secret from a YAML file:

■

```
$ oc apply -f <secret_file>.yaml
```

- To create a secret from standard input:

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

Table 4.2. Required secret parameters

Storage provider
Secret parameters
<a href="#">Red Hat OpenShift Data Foundation</a>
<b>name:</b> tempostack-dev-odf # example <b>bucket:</b> <bucket_name> # requires an ObjectBucketClaim <b>endpoint:</b> https://s3.openshift-storage.svc <b>access_key_id:</b> <data_foundation_access_key_id> <b>access_key_secret:</b> <data_foundation_access_key_secret>
MinIO
See <a href="#">MinIO Operator</a> . <b>name:</b> tempostack-dev-minio # example <b>bucket:</b> <minio_bucket_name> # <a href="#">MinIO documentation</a> <b>endpoint:</b> <minio_bucket_endpoint> <b>access_key_id:</b> <minio_access_key_id> <b>access_key_secret:</b> <minio_access_key_secret>
Amazon S3
<b>name:</b> tempostack-dev-s3 # example <b>bucket:</b> <s3_bucket_name> # <a href="#">Amazon S3 documentation</a> <b>endpoint:</b> <s3_bucket_endpoint> <b>access_key_id:</b> <s3_access_key_id> <b>access_key_secret:</b> <s3_access_key_secret>
Microsoft Azure Blob Storage

**Storage provider**

**name:** `tempostack-dev-azure` # example

**container:** `<azure_blob_storage_container_name>` # [Microsoft Azure documentation](#)

**account\_name:** `<azure_blob_storage_account_name>`

**account\_key:** `<azure_blob_storage_account_key>`

Google Cloud Storage on Google Cloud Platform (GCP)

**name:** `tempostack-dev-gcs` # example

**bucketname:** `<google_cloud_storage_bucket_name>` # requires a [bucket](#) created in a [GCP project](#)

**key.json:** `<path/to/key.json>` # requires a [service account](#) in the bucket's GCP project for GCP authentication

**Example secret for Amazon S3 and MinIO storage**

```
apiVersion: v1
kind: Secret
metadata:
  name: minio-test
stringData:
  endpoint: http://minio.minio.svc:9000
  bucket: tempo
  access_key_id: tempo
  access_key_secret: <secret>
type: Opaque
```

3. Create a project of your choice for the **TempoStack** instance that you will create in the next step:

- To create a project from standard input without metadata:

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

- To create a project from standard input with metadata:

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

4. Create a **TempoStack** instance in the project that you created for the **TempoStack** instance in the previous step.

**NOTE**

You can create multiple **TempoStack** instances in separate projects on the same cluster.

- a. Customize the **TempoStack** custom resource (CR):

```
apiVersion: tempo.grafana.com/v1alpha1
kind: TempoStack
metadata:
  name: sample
  namespace: <project_of_tempostack_instance>
spec:
  storageSize: 1Gi
  storage:
    secret:
      name: <secret-name> 1
      type: <secret-provider> 2
  template:
    queryFrontend:
    jaegerQuery:
      enabled: true
    ingress:
      route:
        termination: edge
      type: route
```

**1** The value of the **name** in the **metadata** of the secret.

**2** The accepted values are **azure** for Azure Blob Storage; **gcs** for Google Cloud Storage; and **s3** for Amazon S3, MinIO, or Red Hat OpenShift Data Foundation.

### TempoStack CR for AWS S3 and MinIO storage

```
apiVersion: tempo.grafana.com/v1alpha1
kind: TempoStack
metadata:
  name: simplest
  namespace: project_of_tempostack_instance
spec:
  storageSize: 1Gi
  storage:
    secret:
      name: minio-test
      type: s3
  resources:
    total:
      limits:
        memory: 2Gi
        cpu: 2000m
  template:
    queryFrontend:
    jaegerQuery:
      enabled: true
```

```

ingress:
  route:
    termination: edge
    type: route

```

The stack deployed in this example is configured to receive Jaeger Thrift over HTTP and OpenTelemetry Protocol (OTLP), which permits visualizing the data with the Jaeger UI.

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

```

$ oc apply -f - << EOF
<TempoStack_custom_resource>
EOF

```

## Verification

1. Verify that the **status** of all TempoStack **components** is **Running** and the **conditions** are **type: Ready** by running the following command:

```
$ oc get tempostacks.tempop.grafana.com simplest -o yaml
```

2. Verify that all the TempoStack component pods are running by running the following command:

```
$ oc get pods
```

3. Access the Tempo console:

- a. Query the route details by running the following command:

```

$ export TEMPO_URL=$(oc get route -n <control_plane_namespace> tempo -o
jsonpath='{.spec.host}')

```

- b. Open **https://<route\_from\_previous\_step>** in a web browser.
- c. Log in using your cluster administrator credentials for the web console.



### NOTE

The Tempo console initially shows no trace data following the Tempo console installation.

## 4.1.3. 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](#)

## 4.2. CONFIGURING AND DEPLOYING THE DISTRIBUTED TRACING PLATFORM (TEMPO)

The Tempo Operator uses a custom resource definition (CRD) file that defines the architecture and configuration settings to be used when creating and deploying the distributed tracing platform (Tempo) resources. You can install the default configuration or modify the file.

### 4.2.1. Customizing your deployment

For information about configuring the back-end storage, see [Understanding persistent storage](#) and the appropriate configuration topic for your chosen storage option.

#### 4.2.1.1. Distributed tracing default configuration options

The Tempo custom resource (CR) defines the architecture and settings to be used when creating the distributed tracing platform (Tempo) resources. You can modify these parameters to customize your distributed tracing platform (Tempo) implementation to your business needs.

#### Example of a generic Tempo YAML file

```
apiVersion: tempo.grafana.com/v1alpha1
kind: TempoStack
metadata:
  name: name
spec:
  storage: {}
  resources: {}
  storageSize: 200M
  replicationFactor: 1
  retention: {}
  template:
    distributor: {}
    ingester: {}
    compactor: {}
    querier: {}
    queryFrontend: {}
    gateway: {}
```

Table 4.3. Tempo parameters

Parameter	Description	Values	Default value
<b>apiVersion:</b>	API version to use when creating the object.	<b>tempotracing.io/v1</b>	<b>tempotracing.io/v1</b>
<b>kind:</b>	Defines the kind of Kubernetes object to create.	<b>tempo</b>	

Parameter	Description	Values	Default value
<b>metadata:</b>	Data that uniquely identifies the object, including a <b>name</b> string, <b>UID</b> , and optional <b>namespace</b> .		OpenShift Container Platform automatically generates the <b>UID</b> and completes the <b>namespace</b> with the name of the project where the object is created.
<b>name:</b>	Name for the object.	Name of your TempoStack instance.	<b>tempo-all-in-one-inmemory</b>
<b>spec:</b>	Specification for the object to be created.	Contains all of the configuration parameters for your TempoStack instance. When a common definition for all Tempo components is required, it is defined under the <b>spec</b> node. When the definition relates to an individual component, it is placed under the <b>spec/template/&lt;component&gt;</b> node.	N/A
<b>resources:</b>	Resources assigned to the TempoStack.		
<b>storageSize:</b>	Storage size for ingester PVCs.		
<b>replicationFactor:</b>	Configuration for the replication factor.		
<b>retention:</b>	Configuration options for retention of traces.		
<b>storage:</b>	Configuration options that define the storage. All storage-related options must be placed under <b>storage</b> and not under the <b>allInOne</b> or other component options.		

Parameter	Description	Values	Default value
<b>template.distributor:</b>	Configuration options for the Tempo <b>distributor</b> .		
<b>template.ingester:</b>	Configuration options for the Tempo <b>ingester</b> .		
<b>template.compactor:</b>	Configuration options for the Tempo <b>compactor</b> .		
<b>template.querier:</b>	Configuration options for the Tempo <b>querier</b> .		
<b>template.queryFrontend:</b>	Configuration options for the Tempo <b>query-frontend</b> .		
<b>template.gateway:</b>	Configuration options for the Tempo <b>gateway</b> .		

### Minimum required configuration

The following is the required minimum for creating a distributed tracing platform (Tempo) deployment with the default settings:

```

apiVersion: tempo.grafana.com/v1alpha1
kind: TempoStack
metadata:
  name: simplest
spec:
  storage: 1
  secret:
    name: minio
    type: s3
  resources:
    total:
      limits:
        memory: 2Gi
        cpu: 2000m
  template:
    queryFrontend:
      jaegerQuery:
        enabled: true
    ingress:
      type: route

```



- 1 This section specifies the deployed object storage back end, which requires a created secret with credentials for access to the object storage.

#### 4.2.1.2. The distributed tracing platform (Tempo) storage configuration

You can configure object storage for the distributed tracing platform (Tempo) in the **TempoStack** custom resource under **spec.storage**. You can choose from among several storage providers that are supported.

**Table 4.4. General storage parameters used by the Tempo Operator to define distributed tracing storage**

Parameter	Description	Values	Default value
<code>spec: storage: secret type:</code>	Type of storage to use for the deployment.	<b>memory</b> . Memory storage is only appropriate for development, testing, demonstrations, and proof of concept environments because the data does not persist when the pod is shut down.	<b>memory</b>
<code>storage: secretname:</code>	Name of the secret that contains the credentials for the set object storage type.		N/A
<code>storage: tls: caName:</code>	CA is the name of a <b>ConfigMap</b> object containing a CA certificate.		

**Table 4.5. Required secret parameters**

Storage provider
Secret parameters
<a href="#">Red Hat OpenShift Data Foundation</a>

Storage provider
<p><b>name:</b> tempostack-dev-odf # example</p> <p><b>bucket:</b> &lt;bucket_name&gt; # requires an ObjectBucketClaim</p> <p><b>endpoint:</b> https://s3.openshift-storage.svc</p> <p><b>access_key_id:</b> &lt;data_foundation_access_key_id&gt;</p> <p><b>access_key_secret:</b> &lt;data_foundation_access_key_secret&gt;</p>
MinIO
<p>See <a href="#">MinIO Operator</a>.</p> <p><b>name:</b> tempostack-dev-minio # example</p> <p><b>bucket:</b> &lt;minio_bucket_name&gt; # <a href="#">MinIO documentation</a></p> <p><b>endpoint:</b> &lt;minio_bucket_endpoint&gt;</p> <p><b>access_key_id:</b> &lt;minio_access_key_id&gt;</p> <p><b>access_key_secret:</b> &lt;minio_access_key_secret&gt;</p>
Amazon S3
<p><b>name:</b> tempostack-dev-s3 # example</p> <p><b>bucket:</b> &lt;s3_bucket_name&gt; # <a href="#">Amazon S3 documentation</a></p> <p><b>endpoint:</b> &lt;s3_bucket_endpoint&gt;</p> <p><b>access_key_id:</b> &lt;s3_access_key_id&gt;</p> <p><b>access_key_secret:</b> &lt;s3_access_key_secret&gt;</p>
Microsoft Azure Blob Storage
<p><b>name:</b> tempostack-dev-azure # example</p> <p><b>container:</b> &lt;azure_blob_storage_container_name&gt; # <a href="#">Microsoft Azure documentation</a></p> <p><b>account_name:</b> &lt;azure_blob_storage_account_name&gt;</p> <p><b>account_key:</b> &lt;azure_blob_storage_account_key&gt;</p>
Google Cloud Storage on Google Cloud Platform (GCP)

## Storage provider

**name:** `tempostack-dev-gcs` # example

**bucketname:** `<google_cloud_storage_bucket_name>` # requires a [bucket](#) created in a [GCP project](#)

**key.json:** `<path/to/key.json>` # requires a [service account](#) in the bucket's GCP project for GCP authentication

## 4.2.1.3. Query configuration options

Query is a service that retrieves traces from storage and hosts the user interface to display them.

Table 4.6. Parameters used by the Tempo Operator to define Query

Parameter	Description	Values	Default value
<b>spec:</b> <b>query:</b> <b>replicas:</b>	Specifies the number of Query replicas to create.	Positive integer	

Table 4.7. Configuration parameters passed to Query

Parameter	Description	Values	Default value
<b>spec:</b> <b>query:</b> <b>options: {}</b>	Configuration options that define the Query service.		
<b>options:</b> <b>log-level:</b>	Logging level for Query.	<b>debug, info, warn, error, fatal, panic</b>	
<b>options:</b> <b>query:</b> <b>base-path:</b>	You can set the base path for all tempo-query HTTP routes to a non-root value: for example, <b>/tempo</b> will cause all UI URLs to start with <b>/tempo</b> . This can be useful when running <b>tempo-query</b> behind a reverse proxy.	<b>/&lt;path&gt;</b>	

## Sample Query configuration

```

apiVersion: tempotracing.io/v1
kind: "Tempo"
metadata:
  name: "my-tempo"
spec:
  strategy: allInOne
  allInOne:
    options:
      log-level: debug
    query:
      base-path: /tempo

```

## 4.2.2. Setting up monitoring for the distributed tracing platform (Tempo)

The Tempo Operator supports monitoring and alerting of each TempoStack component such as distributor, ingester, and so on, and exposes upgrade and operational metrics about the Operator itself.

### 4.2.2.1. Configuring TempoStack metrics and alerts

You can enable metrics and alerts of TempoStack instances.

#### Prerequisites

- Monitoring for user-defined projects is enabled in the cluster. See [Enabling monitoring for user-defined projects](#).

#### Procedure

- To enable metrics of a TempoStack instance, set the **spec.observability.metrics.createServiceMonitors** field to **true**:

```

apiVersion: tempo.grafana.com/v1alpha1
kind: TempoStack
metadata:
  name: <name>
spec:
  observability:
    metrics:
      createServiceMonitors: true

```

- To enable alerts for a TempoStack instance, set the **spec.observability.metrics.createPrometheusRules** field to **true**:

```

apiVersion: tempo.grafana.com/v1alpha1
kind: TempoStack
metadata:
  name: <name>
spec:
  observability:
    metrics:
      createPrometheusRules: true

```

#### Verification

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

1. Go to **Observe → Targets**, filter for **Source: User**, and check that **ServiceMonitors** in the format **tempo-<instance\_name>-<component>** have the **Up** status.
2. To verify that alerts are set up correctly, go to **Observe → Alerting → Alerting rules**, filter for **Source: User**, and check that the **Alert rules** for the TempoStack instance components are available.

#### 4.2.2.2. Configuring Tempo Operator metrics and alerts

When installing the Tempo Operator from the web console, you can select the **Enable Operator recommended cluster monitoring on this Namespace** checkbox, which enables creating metrics and alerts of the Tempo Operator.

If the checkbox was not selected during installation, you can manually enable metrics and alerts even after installing the Tempo Operator.

#### Procedure

- Add the **openshift.io/cluster-monitoring: "true"** label in the project where the Tempo Operator is installed, which is **openshift-tempo-operator** by default.

#### Verification

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

1. Go to **Observe → Targets**, filter for **Source: Platform**, and search for **tempo-operator**, which must have the **Up** status.
2. To verify that alerts are set up correctly, go to **Observe → Alerting → Alerting rules**, filter for **Source: Platform**, and locate the **Alert rules** for the **Tempo Operator**.

## 4.3. UPDATING THE DISTRIBUTED TRACING PLATFORM (TEMPO)

### 4.3.1. Automatic updates of the distributed tracing platform (Tempo)

For version upgrades, the Tempo 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 OpenShift Container Platform by default. The OLM queries for available Operators as well as upgrades for installed Operators.

When the Tempo Operator is upgraded to the new version, it scans for running TempoStack instances that it manages and upgrades them to the version corresponding to the Operator's new version.

### 4.3.2. Additional resources

- [Operator Lifecycle Manager concepts and resources](#)
- [Updating installed Operators](#)

## 4.4. REMOVING THE RED HAT OPENSIFT DISTRIBUTED TRACING PLATFORM (TEMPO)

The steps for removing the Red Hat OpenShift distributed tracing platform (Tempo) from an OpenShift Container Platform cluster are as follows:

1. Shut down all distributed tracing platform (Tempo) pods.
2. Remove any TempoStack instances.
3. Remove the Tempo Operator.

#### 4.4.1. Removing a TempoStack instance by using the web console


You can remove a TempoStack instance in the **Administrator** view of the web console.

##### Prerequisites

- You are logged in to the OpenShift Container Platform 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** → **Tempo Operator** → **TempoStack**.

2. To remove the TempoStack instance, select  → **Delete TempoStack** → **Delete**.
3. Optional: Remove the Tempo Operator.

#### 4.4.2. Removing a TempoStack instance by using the CLI

You can remove a TempoStack 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 TempoStack instance by running the following command:

```
$ oc get deployments -n <project_of_tempostack_instance>
```

2. Remove the TempoStack instance by running the following command:

```
$ oc delete tempo <tempostack_instance_name> -n <project_of_tempostack_instance>
```

3. Optional: Remove the Tempo Operator.

### Verification

1. Run the following command to verify that the TempoStack instance is not found in the output, which indicates its successful removal:

```
$ oc get deployments -n <project_of_tempostack_instance>
```

### 4.4.3. Additional resources

- [Deleting Operators from a cluster](#)
- [Getting started with the OpenShift CLI](#)

## CHAPTER 5. DISTRIBUTED TRACING DATA COLLECTION (OPENTELEMETRY)

### 5.1. INSTALLING THE DISTRIBUTED TRACING DATA COLLECTION



#### IMPORTANT

The Red Hat OpenShift distributed tracing data collection Operator 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](#).

Installing the distributed tracing data collection involves the following steps:

1. Installing the Red Hat OpenShift distributed tracing data collection Operator.
2. Creating a namespace for an OpenTelemetry Collector instance.
3. Creating an **OpenTelemetryCollector** custom resource to deploy the OpenTelemetry Collector instance.

#### 5.1.1. Installing the distributed tracing data collection from the web console

You can install the distributed tracing data collection 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.
- 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 OpenShift distributed tracing data collection Operator:



- a. Go to **Operators → OperatorHub** and search for **Red Hat OpenShift distributed tracing data collection Operator**.
- b. Select the **Red Hat OpenShift distributed tracing data collection 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 OpenTelemetryCollector → YAML view**.
    - c. In the **YAML view**, customize the **OpenTelemetryCollector** custom resource (CR) with the OTLP, Jaeger, Zipkin receiver, and logging 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:
    logging:
  service:
    pipelines:
      traces:
        receivers: [otlp,jaeger,zipkin]
        processors: [memory_limiter,batch]
        exporters: [logging]

```

d. Select **Create**.

## 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>
```

### 5.1.2. 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](#)

## 5.2. CONFIGURING AND DEPLOYING THE DISTRIBUTED TRACING DATA COLLECTION

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

### 5.2.1. OpenTelemetry Collector configuration options

The OpenTelemetry Collector consists of three 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 run through 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.

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
  ports:
  - name: promexporter
    port: 8889
    protocol: TCP
  config: |
    receivers:
      otlp:
        protocols:
          grpc:
          http:
    processors:
    exporters:
      jaeger:
        endpoint: jaeger-production-collector-headless.tracing-system.svc:14250
        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 5.1. Parameters used by the Operator to define the OpenTelemetry Collector

Parameter	Description	Values	Default
<b>receivers:</b>	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.	<b>otlp, jaeger, zipkin</b>	None
<b>processors:</b>	Processors run through the data between it is received and exported. By default, no processors are enabled.		None
<b>exporters:</b>	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.	<b>otlp, otlphttp, jaeger, logging, prometheus</b>	None
<b>service: pipelines:</b>	Components are enabled by adding them to a pipeline under <b>services.pipeline</b> .		

Parameter	Description	Values	Default
 <code>service: pipelines: traces: receivers:</code>	You enable receivers for tracing by adding them under <b><code>service.pipelines.traces</code></b> .		None
 <code>service: pipelines: traces: processors:</code>	You enable processors for tracing by adding them under <b><code>service.pipelines.traces</code></b> .		None
 <code>service: pipelines: traces: exporters:</code>	You enable exporters for tracing by adding them under <b><code>service.pipelines.traces</code></b> .		None
 <code>service: pipelines: metrics: receivers:</code>	You enable receivers for metrics by adding them under <b><code>service.pipelines.metrics</code></b> .		None
 <code>service: pipelines: metrics: processors:</code>	You enable processors for metrics by adding them under <b><code>service.pipelines.metrics</code></b> .		None
 <code>service: pipelines: metrics: exporters:</code>	You enable exporters for metrics by adding them under <b><code>service.pipelines.metrics</code></b> .		None

### 5.2.1.1. OpenTelemetry Collector components

#### 5.2.1.1.1. Receivers

##### 5.2.1.1.1.1. OTLP Receiver

The OTLP receiver ingests data using the OpenTelemetry protocol (OTLP).

- Support level: [Technology Preview](#)

- Supported signals: traces, metrics

### OpenTelemetry Collector custom resource with an enabled OTLP receiver

```
config: |
  receivers:
    otlp:
      protocols:
        grpc:
          endpoint: 0.0.0.0:4317 ❶
          tls: ❷
            ca_file: ca.pem
            cert_file: cert.pem
            key_file: key.pem
            client_ca_file: client.pem ❸
            reload_interval: 1h ❹
        http:
          endpoint: 0.0.0.0:4318 ❺
          tls: ❻
      service:
        pipelines:
          traces:
            receivers: [otlp]
          metrics:
            receivers: [otlp]
```

- ❶ The OTLP gRPC endpoint. If omitted, the default **0.0.0.0:4317** is used.
- ❷ The server-side TLS configuration. Defines paths to TLS certificates. If omitted, TLS is disabled.
- ❸ 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](#).
- ❹ Specifies the time interval at which the certificate is reloaded. If the value is not set, the certificate is never reloaded. **reload\_interval** accepts a string containing valid units of time such as **ns**, **us** (or **µs**), **ms**, **s**, **m**, **h**.
- ❺ The OTLP HTTP endpoint. The default value is **0.0.0.0:4318**.
- ❻ The server-side TLS configuration. For more information, see **grpc** protocol configuration section.

#### 5.2.1.1.1.2. Jaeger Receiver

The Jaeger receiver ingests data in Jaeger formats.

- Support level: [Technology Preview](#)
- Supported signals: traces

### OpenTelemetry Collector custom resource with an enabled Jaeger receiver

```
config: |
```

```

receivers:
  jaeger:
    protocols:
      grpc:
        endpoint: 0.0.0.0:14250 ❶
      thrift_http:
        endpoint: 0.0.0.0:14268 ❷
      thrift_compact:
        endpoint: 0.0.0.0:6831 ❸
      thrift_binary:
        endpoint: 0.0.0.0:6832 ❹
      tls: ❺
    service:
      pipelines:
        traces:
          receivers: [jaeger]

```

- ❶ The Jaeger gRPC endpoint. If omitted, the default **0.0.0.0:14250** is used.
- ❷ The Jaeger Thrift HTTP endpoint. If omitted, the default **0.0.0.0:14268** is used.
- ❸ The Jaeger Thrift Compact endpoint. If omitted, the default **0.0.0.0:6831** is used.
- ❹ The Jaeger Thrift Binary endpoint. If omitted, the default **0.0.0.0:6832** is used.
- ❺ The TLS server side configuration. See the OTLP receiver configuration section for more details.

#### 5.2.1.1.3. Zipkin Receiver

The Zipkin receiver ingests data in the Zipkin v1 and v2 formats.

- Support level: [Technology Preview](#)
- Supported signals: traces

#### OpenTelemetry Collector custom resource with enabled Zipkin receiver

```

config: |
  receivers:
    zipkin:
      endpoint: 0.0.0.0:9411 ❶
      tls: ❷
    service:
      pipelines:
        traces:
          receivers: [zipkin]

```

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

### 5.2.1.1.2. Processors

#### 5.2.1.1.2.1. Resource Detection processor

The Resource Detection processor is designed to identify host resource details in alignment with OpenTelemetry's resource semantic standards. Using this detected information, it can add or replace the resource values in telemetry data.

- Support level: [Technology Preview](#)
- Supported signals: traces, metrics

#### 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]
```

### 5.2.1.1.3. Exporters

#### 5.2.1.1.3.1. OTLP exporter

The OTLP gRPC exporter exports data using the OpenTelemetry protocol (OTLP).

- Support level: [Technology Preview](#)
- Supported signals: traces, metrics

#### 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 ❸
insecure_skip_verify: false ❹
reload_interval: 1h ❺
server_name_override: <name> ❻
headers: ❼
  X-Scope-OrgID: "dev"
service:
  pipelines:
    traces:
      exporters: [otlp]
    metrics:
      exporters: [otlp]

```

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

#### 5.2.1.1.3.2. OTLP HTTP exporter

The OTLP HTTP exporter exports data using the OpenTelemetry protocol (OTLP).

- Support level: [Technology Preview](#)
- Supported signals: traces, metrics

#### OpenTelemetry Collector custom resource with an enabled OTLP exporter

```

config: |
  exporters:
    otlphttp:
      endpoint: http://tempo-ingester:4318 ❶
      tls: ❷
      headers: ❸
        X-Scope-OrgID: "dev"
  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.

#### 5.2.1.1.3.3. Jaeger exporter

The Jaeger exporter exports data using the Jaeger proto format through gRPC.

- Support level: [Technology Preview](#)
- Supported signals: traces

#### OpenTelemetry Collector custom resource with enabled Jaeger exporter

```
config: |
  exporters:
    jaeger:
      endpoint: jaeger-all-in-one:14250 1
      tls: 2
  service:
    pipelines:
      traces:
        exporters: [jaeger]
```

- 1 The Jaeger gRPC endpoint.
- 2 The client side TLS configuration. Defines paths to TLS certificates.

#### 5.2.1.1.3.4. Logging exporter

The Logging exporter prints data to the standard output.

- Support level: [Technology Preview](#)
- Supported signals: traces, metrics

#### OpenTelemetry Collector custom resource with an enabled Logging exporter

```
config: |
  exporters:
    logging:
      verbosity: detailed 1
  service:
    pipelines:
      traces:
```

```

exporters: [logging]
metrics:
  exporters: [logging]

```

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

#### 5.2.1.1.3.5. Prometheus exporter

The Prometheus exporter exports data using the Prometheus or OpenMetrics formats.

- Support level: [Technology Preview](#)
- Supported signals: metrics

### 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
  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

- 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**.

### 5.2.2. Sending metrics to the monitoring stack

You can configure the monitoring stack to scrape OpenTelemetry Collector metrics endpoints and to remove duplicated labels that the monitoring stack has added during scraping.

#### Sample **PodMonitor** custom resource (CR) that configures the monitoring stack to scrape Collector metrics

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: otel-collector
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: otel-collector
  podMetricsEndpoints:
    - port: metrics 1
    - port: promexporter 2
    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 internal metrics port for the OpenTelemetry Collector. This port name is always **metrics**.
- 2 The name of the Prometheus exporter port for the OpenTelemetry Collector. This port name is defined in the **.spec.ports** section of the **OpenTelemetryCollector** CR.

### 5.2.3. Additional resources

- [Enabling monitoring for user-defined projects](#)

## 5.3. USING THE DISTRIBUTED TRACING DATA COLLECTION

### 5.3.1. 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 OpenShift distributed tracing data collection 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" ❶
        tls:
          insecure: true
    service:
      pipelines:
        traces:
          receivers: [jaeger, opencensus, otlp, zipkin] ❷
          processors: [memory_limiter, k8sattributes, resourcedetection, batch]
          exporters: [otlp]

```

- ❶ 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.
- ❷

- 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

## 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](#)

## 5.3.2. Sending traces and metrics to the OpenTelemetry Collector

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

### 5.3.2.1. Sending traces and metrics to the OpenTelemetry Collector with sidecar injection

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

The Red Hat OpenShift distributed tracing data collection 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 the OpenTelemetry Collector.

```
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 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 OpenTelemetryCollector instance.

### 5.3.2.2. Sending traces and metrics to the OpenTelemetry Collector without sidecar injection

You can set up sending telemetry data to an OpenTelemetryCollector 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

- Create a project for the OpenTelemetry Collector.

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

- Create a service account.

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

- Grant 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
```

- Deploy the OpenTelemetryCollector instance.

```
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" ❶
      tls:
        insecure: true
  service:
  pipelines:
    traces:
      receivers: [jaeger, opencensus, otlp, zipkin]
      processors: [memory_limiter, k8sattributes, resourcedetection, batch]
      exporters: [otlp]

```

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

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

Name	Description	Default value
OTEL_SERVICE_NAME	Sets the value of the <b>service.name</b> resource attribute.	""
OTEL_EXPORTER_OTLP_ENDPOINT	Base endpoint URL for any signal type with an optionally specified port number.	<b>https://localhost:4317</b>

Name	Description	Default value
<b>OTEL_EXPORTER_OTLP_CERTIFICATE</b>	Path to the certificate file for the TLS credentials of the gRPC client.	<b>https://localhost:4317</b>
<b>OTEL_TRACES_SAMPLER</b>	Sampler to be used for traces.	<b>parentbased_always_on</b>
<b>OTEL_EXPORTER_OTLP_PROTOCOL</b>	Transport protocol for the OTLP exporter.	<b>grpc</b>
<b>OTEL_EXPORTER_OTLP_TIMEOUT</b>	Maximum time the OTLP exporter will wait for each batch export.	<b>10s</b>
<b>OTEL_EXPORTER_OTLP_INSECURE</b>	Disables client transport security for gRPC requests; an HTTPS schema overrides it.	<b>False</b>

## 5.4. TROUBLESHOOTING THE DISTRIBUTED TRACING DATA COLLECTION

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

### 5.4.1. 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 OpenTelemetry Collector custom resource (CR):

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

- 1** Collector's log level. Select one of the following values: **info**, **warn**, **error**, or **debug**. Defaults to **info**.

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

### 5.4.2. 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 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 OpenTelemetry Collector custom resource:

```
config: |
  service:
    telemetry:
      metrics:
        address: ":8888" 1
```

- 1** The address on which internal collector metrics are exposed. Defaults to **:8888**.

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

```
$ oc port-forward <collector-pod>
```

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

### 5.4.3. Logging exporter

You can configure the logging exporter to export the collected data to the standard output.

#### Procedure

1. Configure the OpenTelemetry Collector custom resource as follows:

```
config: |
  exporters:
    logging:
      verbosity: detailed
  service:
    pipelines:
```

```
traces:
  exporters: [logging]
metrics:
  exporters: [logging]
logs:
  exporters: [logging]
```

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

## 5.5. MIGRATING FROM THE DISTRIBUTED TRACING PLATFORM (JAEGER) TO THE DISTRIBUTED TRACING DATA COLLECTION

If you are already using Red Hat OpenShift distributed tracing platform (Jaeger) for your applications, you can migrate to the Red Hat OpenShift distributed tracing data collection, which is based on the [OpenTelemetry](#) open-source project.

The distributed tracing data collection provides a set of APIs, libraries, agents, and instrumentation to facilitate observability in distributed systems. The OpenTelemetry Collector in the distributed tracing data collection 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 distributed tracing data collection requires configuring the OpenTelemetry Collector and your applications to report traces seamlessly. You can migrate sidecar and sidecarless deployments.

### 5.5.1. Migrating from the distributed tracing platform (Jaeger) to the distributed tracing data collection with sidecars

The distributed tracing data collection Operator supports sidecar injection into deployment workloads, so you can migrate from a distributed tracing platform (Jaeger) sidecar to a distributed tracing data collection sidecar.

#### Prerequisites

- The Red Hat OpenShift distributed tracing platform (Jaeger) is used on the cluster.
- The Red Hat OpenShift distributed tracing data collection 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" ❶
    tls:
      insecure: true
service:
  pipelines:
    traces:
      receivers: [jaeger]
      processors: [memory_limiter, resourcedetection, batch]
      exporters: [otlp]

```

- ❶ 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:
  ❶
  - apiGroups: ["config.openshift.io"]
    resources: ["infrastructures", "infrastructures/status"]
    verbs: ["get", "watch", "list"]

```

- ❶ 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.

### 5.5.2. Migrating from the distributed tracing platform (Jaeger) to the distributed tracing data collection without sidecars

You can migrate from the distributed tracing platform (Jaeger) to the distributed tracing data collection without sidecar deployment.

#### Prerequisites

- The Red Hat OpenShift distributed tracing platform (Jaeger) is used on the cluster.
- The Red Hat OpenShift distributed tracing data collection 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.

## 5.6. REMOVING THE DISTRIBUTED TRACING DATA COLLECTION

The steps for removing the distributed tracing data collection from an OpenShift Container Platform cluster are as follows:

1. Shut down all distributed tracing data collection pods.
2. Remove any OpenTelemetryCollector instances.
3. Remove the Red Hat OpenShift distributed tracing data collection Operator.

### 5.6.1. Removing a distributed tracing data collection instance by using the web console

You can remove a distributed tracing data collection 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 OpenShift distributed tracing data collection Operator** → **OpenTelemetryInstrumentation** or **OpenTelemetryCollector**.



2. To remove the relevant instance, select → **Delete ...** → **Delete**.
3. Optional: Remove the Red Hat OpenShift distributed tracing data collection Operator.

### 5.6.2. Removing a distributed tracing data collection instance by using the CLI

You can remove a distributed tracing data collection 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 distributed tracing data collection instance by running the following command:

```
$ oc get deployments -n <project_of_opentelemetry_instance>
```

2. Remove the distributed tracing data collection instance by running the following command:

```
$ oc delete opentelemetrycollectors <opentelemetry_instance_name> -n  
<project_of_opentelemetry_instance>
```

3. Optional: Remove the Red Hat OpenShift distributed tracing data collection Operator.

## Verification

- To verify successful removal of the distributed tracing data collection instance, run **oc get deployments** again:

```
$ oc get deployments -n <project_of_opentelemetry_instance>
```

### 5.6.3. Additional resources

- [Deleting Operators from a cluster](#)
- [Getting started with the OpenShift CLI](#)