Red Hat OpenShift Cluster Observability Operator 1-latest

About Red Hat OpenShift Cluster Observability Operator

Introduction to Cluster Observability Operator.

Last Updated: 2025-12-17

Red Hat OpenShift Cluster Observability Operator 1-latest About Red Hat OpenShift Cluster Observability Operator

Introduction to Cluster Observability Operator.

Legal Notice

Abstract

This document provides an overview of Cluster Observability Operator features, and also includes release notes and support information.

Table of Contents

CHAPTER 1. CLUSTER OBSERVABILITY OPERATOR OVERVIEW

The Cluster Observability Operator (COO) is an optional component of the OpenShift Container Platform designed for creating and managing highly customizable monitoring stacks. It enables cluster administrators to automate configuration and management of monitoring needs extensively, offering a more tailored and detailed view of each namespace compared to the default OpenShift Container Platform monitoring system.

The COO deploys the following monitoring components:

- **Prometheus** - A highly available Prometheus instance capable of sending metrics to an external endpoint by using remote write.
- **Thanos Querier** (optional) - Enables querying of Prometheus instances from a central location.
- **Alertmanager** (optional) - Provides alert configuration capabilities for different services.
- **UI plugins** (optional) - Enhances the observability capabilities with plugins for monitoring, logging, distributed tracing and troubleshooting.
- **Korrel8r** (optional) - Provides observability signal correlation, powered by the open source Korrel8r project.
- **Incident detection** (optional) - Groups related alerts into incidents, to help you identify the root causes of alert bursts.

1.1. COO COMPARED TO DEFAULT MONITORING STACK

The COO components function independently of the default in-cluster monitoring stack, which is deployed and managed by the Cluster Monitoring Operator (CMO). Monitoring stacks deployed by the two Operators do not conflict. You can use a COO monitoring stack in addition to the default platform monitoring components deployed by the CMO.

The key differences between COO and the default in-cluster monitoring stack are shown in the following table:

Feature	COO	Default monitoring stack
Scope and integration	Offers comprehensive monitoring and analytics for enterprise-level needs, covering cluster and workload performance. However, it lacks direct integration with OpenShift Container Platform and typically requires an external Grafana instance for dashboards.	Limited to core components within the cluster, for example, API server and etcd, and to OpenShift-specific namespaces. There is deep integration into OpenShift Container Platform including console dashboards and alert management in the console.

Feature	COO	Default monitoring stack
Configuration and customization	Broader configuration options including data retention periods, storage methods, and collected data types. The COO can delegate ownership of single configurable fields in custom resources to users by using Server-Side Apply (SSA), which enhances customization.	Built-in configurations with limited customization options.
Data retention and storage	Long-term data retention, supporting historical analysis and capacity planning	Shorter data retention times, focusing on short-term monitoring and real-time detection.

1.2. KEY ADVANTAGES OF USING COO

Deploying COO helps you address monitoring requirements that are hard to achieve using the default monitoring stack.

1.2.1. Extensibility

- You can add more metrics to a COO-deployed monitoring stack, which is not possible with core platform monitoring without losing support.
- You can receive cluster-specific metrics from core platform monitoring through federation.
- COO supports advanced monitoring scenarios like trend forecasting and anomaly detection.

1.2.2. Multi-tenancy support

- You can create monitoring stacks per user namespace.
- You can deploy multiple stacks per namespace or a single stack for multiple namespaces.
- COO enables independent configuration of alerts and receivers for different teams.

1.2.3. Scalability

- Supports multiple monitoring stacks on a single cluster.
- Enables monitoring of large clusters through manual sharding.
- Addresses cases where metrics exceed the capabilities of a single Prometheus instance.

1.2.4. Flexibility

- Decoupled from OpenShift Container Platform release cycles.
- Faster release iterations and rapid response to changing requirements.
- Independent management of alerting rules.

1.3. TARGET USERS FOR COO

COO is ideal for users who need high customizability, scalability, and long-term data retention, especially in complex, multi-tenant enterprise environments.

1.3.1. Enterprise-level users and administrators

Enterprise users require in-depth monitoring capabilities for OpenShift Container Platform clusters, including advanced performance analysis, long-term data retention, trend forecasting, and historical analysis. These features help enterprises better understand resource usage, prevent performance issues, and optimize resource allocation.

1.3.2. Operations teams in multi-tenant environments

With multi-tenancy support, COO allows different teams to configure monitoring views for their projects and applications, making it suitable for teams with flexible monitoring needs.

1.3.3. Development and operations teams

COO provides fine-grained monitoring and customizable observability views for in-depth troubleshooting, anomaly detection, and performance tuning during development and operations.

1.4. USING SERVER-SIDE APPLY TO CUSTOMIZE PROMETHEUS RESOURCES

Server-Side Apply is a feature that enables collaborative management of Kubernetes resources. The control plane tracks how different users and controllers manage fields within a Kubernetes object. It introduces the concept of field managers and tracks ownership of fields. This centralized control provides conflict detection and resolution, and reduces the risk of unintended overwrites.

Compared to Client-Side Apply, it is more declarative, and tracks field management instead of last applied state.

Server-Side Apply

Declarative configuration management by updating a resource's state without needing to delete and recreate it.

Field management

Users can specify which fields of a resource they want to update, without affecting the other fields.

Managed fields

Kubernetes stores metadata about who manages each field of an object in the **managedFields** field within metadata.

Conflicts

If multiple managers try to modify the same field, a conflict occurs. The applier can choose to overwrite, relinquish control, or share management.

Merge strategy

Server-Side Apply merges fields based on the actor who manages them.

Procedure

1. Add a **MonitoringStack** resource using the following configuration:

Example MonitoringStack object

```
apiVersion: monitoring.rhobs/v1alpha1
kind: MonitoringStack
metadata:
  labels:
    coo: example
  name: sample-monitoring-stack
  namespace: coo-demo
spec:
  logLevel: debug
  retention: 1d
  resourceSelector:
    matchLabels:
      app: demo
```

2. A Prometheus resource named **sample-monitoring-stack** is generated in the **coo-demo** namespace. Retrieve the managed fields of the generated Prometheus resource by running the following command:

```
$ oc -n coo-demo get Prometheus.monitoring.rhobs -oyaml --show-managed-fields
```

Example output

```
managedFields:
- apiVersion: monitoring.rhobs/v1
  fieldsType: FieldsV1
  fieldsV1:
    f:metadata:
      f:labels:
        f:app.kubernetes.io/managed-by: {}
        f:app.kubernetes.io/name: {}
        f:app.kubernetes.io/part-of: {}
      f:ownerReferences:
        k:{"uid":"81da0d9a-61aa-4df3-affc-71015bcbde5a"}: {}
    f:spec:
      f:additionalScrapeConfigs: {}
      f:affinity:
        f:podAntiAffinity:
          f:requiredDuringSchedulingIgnoredDuringExecution: {}
      f:alerting:
        f:alertmanagers: {}
      f:arbitraryFSAccessThroughSMs: {}
      f:logLevel: {}
      f:podMetadata:
        f:labels:
          f:app.kubernetes.io/component: {}
          f:app.kubernetes.io/part-of: {}
      f:podMonitorSelector: {}
      f:replicas: {}
      f:resources:
        f:limits:
          f:cpu: {}
          f:memory: {}
        f:requests:
```

```
        f:cpu: {}
        f:memory: {}
    f:retention: {}
    f:ruleSelector: {}
    f:rules:
      f:alert: {}
    f:securityContext:
      f:fsGroup: {}
      f:runAsNonRoot: {}
      f:runAsUser: {}
    f:serviceAccountName: {}
    f:serviceMonitorSelector: {}
    f:thanos:
      f:baseImage: {}
      f:resources: {}
      f:version: {}
    f:tsdb: {}
  manager: observability-operator
  operation: Apply
- apiVersion: monitoring.rhobs/v1
  fieldsType: FieldsV1
  fieldsV1:
    f:status:
      .: {}
      f:availableReplicas: {}
      f:conditions:
        .: {}
        k:{"type":"Available"}:
          .: {}
          f:lastTransitionTime: {}
          f:observedGeneration: {}
          f:status: {}
          f:type: {}
        k:{"type":"Reconciled"}:
          .: {}
          f:lastTransitionTime: {}
          f:observedGeneration: {}
          f:status: {}
          f:type: {}
      f:paused: {}
      f:replicas: {}
      f:shardStatuses:
        .: {}
        k:{"shardID":"0"}:
          .: {}
          f:availableReplicas: {}
          f:replicas: {}
          f:shardID: {}
          f:unavailableReplicas: {}
          f:updatedReplicas: {}
      f:unavailableReplicas: {}
      f:updatedReplicas: {}
  manager: PrometheusOperator
  operation: Update
  subresource: status
```

3. Check the **metadata.managedFields** values, and observe that some fields in **metadata** and **spec** are managed by the **MonitoringStack** resource.

4. Modify a field that is not controlled by the **MonitoringStack** resource:

   a. Change **spec.enforcedSampleLimit**, which is a field not set by the **MonitoringStack** resource. Create the file **prom-spec-edited.yaml**:

   prom-spec-edited.yaml

```
apiVersion: monitoring.rhobs/v1
kind: Prometheus
metadata:
  name: sample-monitoring-stack
  namespace: coo-demo
spec:
  enforcedSampleLimit: 1000
```

   b. Apply the YAML by running the following command:

```
$ oc apply -f ./prom-spec-edited.yaml --server-side
```

NOTE

You must use the **--server-side** flag.

   c. Get the changed Prometheus object and note that there is one more section in **managedFields** which has **spec.enforcedSampleLimit**:

```
$ oc get prometheus -n coo-demo
```

Example output

```
managedFields: (1)
- apiVersion: monitoring.rhobs/v1
  fieldsType: FieldsV1
  fieldsV1:
    f:metadata:
      f:labels:
        f:app.kubernetes.io/managed-by: {}
        f:app.kubernetes.io/name: {}
        f:app.kubernetes.io/part-of: {}
    f:spec:
      f:enforcedSampleLimit: {} (2)
  manager: kubectl
  operation: Apply
```

(1) **managedFields**

(2) **spec.enforcedSampleLimit**

5. Modify a field that is managed by the **MonitoringStack** resource:

a. Change **spec.LogLevel**, which is a field managed by the **MonitoringStack** resource, using the following YAML configuration:

```
# changing the logLevel from debug to info
apiVersion: monitoring.rhobs/v1
kind: Prometheus
metadata:
  name: sample-monitoring-stack
  namespace: coo-demo
spec:
  logLevel: info (1)
```

(1) **spec.logLevel** has been added

b. Apply the YAML by running the following command:

```
$ oc apply -f ./prom-spec-edited.yaml --server-side
```

Example output

```
error: Apply failed with 1 conflict: conflict with "observability-operator": .spec.logLevel
Please review the fields above--they currently have other managers. Here
are the ways you can resolve this warning:
* If you intend to manage all of these fields, please re-run the apply
  command with the `--force-conflicts` flag.
* If you do not intend to manage all of the fields, please edit your
  manifest to remove references to the fields that should keep their
  current managers.
* You may co-own fields by updating your manifest to match the existing
  value; in this case, you'll become the manager if the other manager(s)
  stop managing the field (remove it from their configuration).
See https://kubernetes.io/docs/reference/using-api/server-side-apply/#conflicts
```

c. Notice that the field **spec.logLevel** cannot be changed using Server-Side Apply, because it is already managed by **observability-operator**.

d. Use the **--force-conflicts** flag to force the change.

```
$ oc apply -f ./prom-spec-edited.yaml --server-side --force-conflicts
```

Example output

```
prometheus.monitoring.rhobs/sample-monitoring-stack serverside-applied
```

With **--force-conflicts** flag, the field can be forced to change, but since the same field is also managed by the **MonitoringStack** resource, the Observability Operator detects the change, and reverts it back to the value set by the **MonitoringStack** resource.

NOTE

Some Prometheus fields generated by the **MonitoringStack** resource are influenced by the fields in the **MonitoringStack spec** stanza, for example, **logLevel**. These can be changed by changing the **MonitoringStack spec**.

e. To change the **logLevel** in the Prometheus object, apply the following YAML to change the **MonitoringStack** resource:

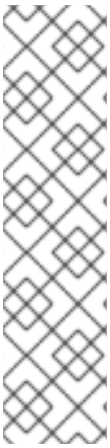
```
apiVersion: monitoring.rhobs/v1alpha1
kind: MonitoringStack
metadata:
  name: sample-monitoring-stack
  labels:
    coo: example
spec:
  logLevel: info
```

f. To confirm that the change has taken place, query for the log level by running the following command:

```
$ oc -n coo-demo get Prometheus.monitoring.rhobs -
o=jsonpath='{.items[0].spec.logLevel}'
```

Example output

```
info
```



NOTE

1. If a new version of an Operator generates a field that was previously generated and controlled by an actor, the value set by the actor will be overridden. For example, you are managing a field **enforcedSampleLimit** which is not generated by the **MonitoringStack** resource. If the Observability Operator is upgraded, and the new version of the Operator generates a value for **enforcedSampleLimit**, this will overide the value you have previously set.

2. The **Prometheus** object generated by the **MonitoringStack** resource may contain some fields which are not explicitly set by the monitoring stack. These fields appear because they have default values.

Additional resources

- Kubernetes documentation for Server-Side Apply (SSA)