# Red Hat OpenShift Cluster Observability Operator 1-latest

## Red Hat OpenShift Cluster Observability Operator について

Cluster Observability Operator の概要

Last Updated: 2025-12-07

# Red Hat OpenShift Cluster Observability Operator 1-latest Red Hat OpenShift Cluster Observability Operator について

Cluster Observability Operator の概要

## Legal Notice



## Abstract

このドキュメントでは、Cluster Observability Operator の機能の概要を説明します。これには、リリースノートとサポート情報も含まれています。

# Table of Contents

**第1章 CLUSTER OBSERVABILITY OPERATOR の概要** ..... 3
1.1. デフォルトのモニタリングスタックと比較した COO 3
1.2. COO を使用する主な利点 4
1.3. COO のターゲットユーザー 5
1.4. SERVER-SIDE APPLY を使用した PROMETHEUS リソースのカスタマイズ 5

# 第1章 CLUSTER OBSERVABILITY OPERATOR の概要

Cluster Observability Operator (COO) は、高度にカスタマイズ可能なモニタリングスタックを作成し、管理するために設計された OpenShift Container Platform のオプションのコンポーネントです。これにより、クラスター管理者はモニタリングの設定と管理を大幅に自動化でき、デフォルトの OpenShift Container Platform のモニタリングシステムと比べて、各 namespace に対するより詳細でカスタマイズされたビューを提供できます。

COO は、次のモニタリングコンポーネントをデプロイします。

- **Prometheus**: リモート書き込みを使用してメトリクスを外部エンドポイントに送信できる高可用性 Prometheus インスタンス。
- **Thanos Querier** (オプション): Prometheus インスタンスを中央の場所からクエリーできるようにします。
- **Alertmanager** (オプション): さまざまなサービスのアラート設定機能を提供します。
- UI plugins (オプション): モニタリング、ロギング、分散トレーシング、およびトラブルシューティング用にプラグインで可観測性機能を強化します。
- **Korrel8r** (オプション): オープンソースの Korrel8r プロジェクトが提供する可観測性シグナルの相関を提供します。
- Incident detection (オプション): アラートバーストの根本原因を特定するために、関連するアラートをインシデントにグループ化します。

## 1.1. デフォルトのモニタリングスタックと比較した COO

COO コンポーネントは、Cluster Monitoring Operator (CMO) でデプロイおよび管理されるデフォルトのクラスター内モニタリングスタックとは独立して機能します。2 つの Operator でデプロイされたモニタリングスタックは競合しません。CMO でデプロイされたデフォルトのプラットフォームモニタリングコンポーネントに加え、COO モニタリングスタックを使用できます。

COO とデフォルトのクラスター内のモニタリングスタックの主な相違点を次の表に示します。

| 機能 | COO | デフォルトのモニタリングスタック |
|---|---|---|
| **スコープおよびインテグレーション** | クラスターやワークロードのパフォーマンスを含め、エンタープライズレベルのニーズに対応した包括的なモニタリングと分析を提供します。<br><br>ただし、OpenShift Container Platform との直接統合がなく、通常はダッシュボードに外部 Grafana インスタンスが必要です。 | クラスター内のコアコンポーネント (API サーバーや etcd など) および OpenShift 固有の namespace に限定されます。<br><br>OpenShift Container Platform へのディープインテグレーションがあり、コンソールのダッシュボードやアラート管理が含まれています。 |

| 機能 | COO | デフォルトのモニタリングスタック |
| --- | --- | --- |
| **設定とカスタマイズ** | データ保持期間、ストレージ方法、収集したデータタイプなど、より広範な設定オプション。<br><br>COO は、カスタマイズを強化する Server-Side Apply (SSA) を使用して、カスタムリソース内にある 1 つの設定可能フィールドの所有権をユーザーに委譲できます。 | カスタマイズオプションが制限された組み込み設定。 |
| **データの保持とストレージ** | 長期のデータ保持。履歴分析と容量計画をサポートします。 | 短期間のデータ保持。短期間のモニタリングとリアルタイム検出に焦点を当てています。 |

## 1.2. COO を使用する主な利点

COO のデプロイは、デフォルトのモニタリングスタックを使用して達成することが困難なモニタリング要件に対応する際に役立ちます。

### 1.2.1. 拡張性

- COO でデプロイされたモニタリングスタックにさらにメトリクスを追加できますが、これをコアプラットフォームモニタリングで行った場合はサポートされません。
- フェデレーションを介して、コアプラットフォームのモニタリングからクラスター固有のメトリクスを受け取ることができます。
- COO は、トレンド予測や異常検出などの高度なモニタリングシナリオをサポートします。

### 1.2.2. マルチテナンシーのサポート

- ユーザー namespace ごとにモニタリングスタックを作成できます。
- namespace ごとに複数のスタックをデプロイすることや、複数の namespace に単一のスタックをデプロイすることができます。
- COO は、異なるチームのアラートとレシーバーの独立した設定を可能にします。

### 1.2.3. スケーラビリティー

- 単一クラスターで複数のモニタリングスタックをサポートします。
- 手動シャーディングを使用した大規模なクラスターのモニタリングを可能にします。
- メトリクスが単一の Prometheus インスタンスの機能を超えるケースに対応します。

### 1.2.4. 柔軟性

- OpenShift Container Platform リリースサイクルから切り離されます。
- より速いリリースサイクルを実現し、変化する要件へ迅速に対応します。

- アラートルールを独立して管理します。

## 1.3. COO のターゲットユーザー

COO は、特に複雑なマルチテナントエンタープライズ環境で、高いカスタマイズ性、スケーラビリティー、および長期のデータ保持が必要なユーザーに適しています。

### 1.3.1. エンタープライズレベルのユーザーおよび管理者

エンタープライズユーザーには、高度なパフォーマンス分析、長期のデータ保持、トレンド予測、履歴分析など、OpenShift Container Platform クラスターの詳細なモニタリング機能が必要です。これらの機能により、企業はリソースの使用状況をより深く理解し、パフォーマンスの問題を防ぎ、リソースの割り当てを最適化できます。

### 1.3.2. マルチテナント環境でのオペレーションチーム

マルチテナンシーのサポートにより、COO はさまざまなチームがプロジェクトやアプリケーションのモニタリングビューを設定できるため、柔軟なモニタリングニーズがあるチームに適しています。

### 1.3.3. 開発およびオペレーションチーム

COO は、詳細なトラブルシューティング、異常検出、開発および運用時のパフォーマンス調整のために、きめ細かなモニタリングとカスタマイズ可能な可観測性ビューを提供します。

## 1.4. SERVER-SIDE APPLY を使用した PROMETHEUS リソースのカスタマイズ

Server-Side Apply は、Kubernetes リソースの共同管理を可能にする機能です。コントロールプレーンは、さまざまなユーザーおよびコントローラーが Kubernetes オブジェクト内のフィールドをどのように管理するかを追跡します。フィールドマネージャーの概念を導入し、フィールドの所有権を追跡します。この集中制御により、競合検出および解決が行われ、意図しない上書きのリスクが軽減されます。

Client-Side Apply と比較すると、より宣言的であり、最後に適用された状態ではなく、フィールド管理を追跡します。

**Server-Side Apply**

リソースの状態を更新することで、削除や再作成を必要とせずに宣言型の設定を管理します。

**フィールド管理**

ユーザーは、他のフィールドに影響を与えずに、更新するリソースのフィールドを指定できます。

**管理対象フィールド**

Kubernetes は、メタデータ内の **managedFields** フィールドでオブジェクトの各フィールドを管理するユーザーに関するメタデータを保存します。

**Conflicts**

複数のマネージャーが同じフィールドを変更しようとすると、競合が発生します。アプライヤーは、上書きするか、制御を放棄するか、または管理を共有するかを選択できます。

**マージストラテジー**

Server-Side Apply は、管理しているアクターに基づいてフィールドをマージします。

**手順**

1. 次の設定を使用して **MonitoringStack** リソースを追加します。

**MonitoringStack オブジェクトの例**

```
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. **sample-monitoring-stack** という名前の Prometheus リソースが、**coo-demo** namespace に生成されます。次のコマンドを実行して、生成された Prometheus リソースの管理対象フィールドを取得します。

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

**出力例**

```
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. **metadata.managedFields** 値を確認し、**metadata** と **spec** の一部のフィールドが **MonitoringStack** リソースによって管理されていることを確認します。

4. **MonitoringStack** リソースで制御されないフィールドを変更します。

   a. **MonitoringStack** リソースによって設定されていないフィールドである **spec.enforcedSampleLimit** を変更します。**prom-spec-edited.yaml** ファイルを作成します。

   **prom-spec-edited.yaml**

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

   b. 以下のコマンドを実行して YAML を適用します。

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

**注記**

**--server-side** フラグを使用する必要があります。

   c. 変更された Prometheus オブジェクトを取得し、**spec.enforcedSampleLimit** を持つ **managedFields** に、もう 1 つセクションがあることに注意してください。

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

**出力例**

```
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. **MonitoringStack** リソースによって管理されるフィールドを変更します。

a. 次の YAML 設定を使用して、**MonitoringStack** リソースによって管理されるフィールドである **spec.LogLevel** を変更します。

```
# 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** が追加されました。

b. 以下のコマンドを実行して YAML を適用します。

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

**出力例**

```
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. フィールド **spec.logLevel** は **observability-operator** によってすでに管理されているため、Server-Side Apply を使用して変更できないことに注意してください。

d. この変更を強制するには、**--force-conflicts** フラグを使用します。

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

**出力例**

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

**--force-conflicts** フラグの場合、このフィールドは強制的に変更できますが、同じフィールドが **MonitoringStack** リソースでも管理されるため、Observability Operator は変更を検出し、**MonitoringStack** リソースによって設定された値に戻します。

**注記**

**MonitoringStack** リソースによって生成される一部の Prometheus フィールドは、**logLevel** など、**MonitoringStack spec** スタンザのフィールドの影響を受けます。これらは、**MonitoringStack spec** を変更することで変更できます。

e. Prometheus オブジェクトの **logLevel** を変更するには、次の YAML を適用して **MonitoringStack** リソースを変更します。

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

f. 変更が実行されたことを確認するには、次のコマンドを実行してログレベルをクエリーします。

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

**出力例**

```
info
```

**注記**

1. Operator の新規バージョンが、以前にアクターによって生成および制御されるフィールドを生成する場合、アクターによって設定された値はオーバーライドされます。
   たとえば、**MonitoringStack** リソースによって生成されないフィールド **enforcedSampleLimit** を管理しているとします。Observability Operator がアップグレードされ、新しいバージョンの Operator が **enforcedSampleLimit** の値を生成すると、以前に設定した値がオーバーライドされます。

2. **MonitoringStack** リソースによって生成された **Prometheus** オブジェクトには、モニタリングスタックによって明示的に設定されていないフィールドが含まれる場合があります。これらのフィールドは、デフォルト値があるために表示されます。

**関連情報**

- Server-Side Apply (SSA) に関する Kubernetes ドキュメント