

Operator for MySQL based on Percona Server for MySQL

Documentation

1.2.0 (July 3, 2026)

Table of Contents

- Home
- Get help from Percona
- Understand the Operator
  - Features and capabilities
  - How the Operator works
  - Design and architecture
- Quick start
  - Overview
  - 1. System requirements
  - 2. Install the Operator
    - Install with Helm
    - Install with kubectl
    - Install with customized parameters
  - 3. Connect to the database
  - 4. Insert data
  - 5. Make a backup
  - 6. Monitor the database with PMM
  - What's next?
- Platform-specific installation
  - Install on Google Kubernetes Engine (GKE)
  - Install on Amazon Elastic Kubernetes Service (AWS EKS)
  - Install on OpenShift
  - Install on Minikube
  - Generic Kubernetes installation
- Daily operations
  - Manage users
  - Connection secrets
  - Backup and restore
    - About backups

- Configure storage for backups
- Incremental backups
- Make scheduled backups
- Make on-demand backup
- Point-in-time recovery
- Compressed backups
- Encrypted backups
- Restore from a backup
  - On the same cluster
  - On a new cluster
- Fine-tune backups and restores
- Delete the unneeded backup

Monitoring

- Monitor your cluster with PMM
- Monitor Kubernetes

Scale your cluster

Production configuration

TLS/SSL encryption

- About TLS/SSL security
- Configure TLS using cert-manager
- Generate certificates manually
- Update certificates

Data-at-rest encryption

- About data-at-rest encryption
- Configure data-at-rest encryption

External access

- Expose your cluster
- Configure HAProxy
- Configure MySQL Router

Control Pod scheduling

Fine-tune MySQL options

Configure DNS suffix

Multi-namespace deployment

- Cross-site replication
  - About cross-site replication
  - Cross-site replication: setup and use
- Define environment variables
  - Overview
  - Configure Operator environment variables
  - Define environment variables for cluster components

Cluster lifecycle management

- Upgrade
  - About upgrades
  - Upgrade the Operator
    - Considerations
    - Manually
    - With Helm
    - Via OLM
  - Upgrade the database
- Pause or restart the cluster
- Cleanup
  - Delete the Operator

Advanced operations

- Change replication type
- Add sidecar containers
- Labels and annotations
- Configure telemetry
- Configure DNS suffix
- Configure concurrent reconciliation

Troubleshooting

- Initial troubleshooting
- Exec into the container
- Check the events
- Check the logs

- Troubleshoot backups and restores

Reference

- Custom Resource reference
  - Custom Resource options
  - ClusterSet Resource options
  - Backup Resource options
  - Restore Resource options
  - Custom resource statuses
- Certified images
  - Percona certified images
  - Retrieve certified images
- Version compatibility
- Legal
  - Copyright and licensing information
  - Trademark policy

Release Notes

- Release notes index
- Percona Operator for MySQL 1.2.0 (2026-07-03)
- Percona Operator for MySQL 1.1.0 (2026-04-17)
- Percona Operator for MySQL 1.0.0 (2025-11-17)
- Percona Operator for MySQL 0.12.0 (2025-09-23)
- Percona Operator for MySQL 0.11.0 (2025-09-01)
- Percona Operator for MySQL 0.10.0 (2025-06-04)
- Percona Operator for MySQL 0.9.0 (2025-02-11)
- Percona Operator for MySQL 0.8.0 (2024-07-16)
- Percona Operator for MySQL 0.7.0 (2024-03-25)
- Percona Operator for MySQL 0.6.0 (2023-09-05)
- Percona Operator for MySQL 0.5.0 (2023-03-30)
- Percona Operator for MySQL 0.4.0 (2023-01-30)
- Percona Operator for MySQL 0.3.0 (2022-09-29)
- Percona Operator for MySQL 0.2.0 (2022-06-30)
- Percona Distribution for MySQL Operator based on Percona Server for MySQL 0.1.0 (2022-01-25)

- Home
- Get help from Percona
- Understand the Operator
  - Features and capabilities
  - How the Operator works
  - Design and architecture
- Quick start
  - Overview
  - 1. System requirements
  - 2. Install the Operator
    - Install with Helm
    - Install with kubectl
    - Install with customized parameters
  - 3. Connect to the database
  - 4. Insert data
  - 5. Make a backup
  - 6. Monitor the database with PMM
  - What's next?
- Platform-specific installation
  - Install on Google Kubernetes Engine (GKE)
  - Install on Amazon Elastic Kubernetes Service (AWS EKS)
  - Install on OpenShift
  - Install on Minikube
  - Generic Kubernetes installation
- Daily operations
  - Manage users
  - Connection secrets
  - Backup and restore
    - About backups
    - Configure storage for backups
    - Incremental backups

- Make scheduled backups
- Make on-demand backup
- Point-in-time recovery
- Compressed backups
- Encrypted backups
- Restore from a backup
  - On the same cluster
  - On a new cluster
- Fine-tune backups and restores
- Delete the unneeded backup

Monitoring

- Monitor your cluster with PMM
- Monitor Kubernetes

Scale your cluster

Production configuration

TLS/SSL encryption

- About TLS/SSL security
- Configure TLS using cert-manager
- Generate certificates manually
- Update certificates

Data-at-rest encryption

- About data-at-rest encryption
- Configure data-at-rest encryption

External access

- Expose your cluster
- Configure HAProxy
- Configure MySQL Router

Control Pod scheduling

Fine-tune MySQL options

Configure DNS suffix

Multi-namespace deployment

Cross-site replication

- About cross-site replication

- Cross-site replication: setup and use
- Define environment variables
  - Overview
  - Configure Operator environment variables
  - Define environment variables for cluster components

Cluster lifecycle management

- Upgrade
  - About upgrades
  - Upgrade the Operator
    - Considerations
    - Manually
    - With Helm
    - Via OLM
  - Upgrade the database
- Pause or restart the cluster
- Cleanup
  - Delete the Operator

Advanced operations

- Change replication type
- Add sidecar containers
- Labels and annotations
- Configure telemetry
- Configure DNS suffix
- Configure concurrent reconciliation

Troubleshooting

- Initial troubleshooting
- Exec into the container
- Check the events
- Check the logs
- Troubleshoot backups and restores

Reference

- Custom Resource reference
  - Custom Resource options
  - ClusterSet Resource options
  - Backup Resource options
  - Restore Resource options
  - Custom resource statuses
- Certified images
  - Percona certified images
  - Retrieve certified images
- Version compatibility
- Legal
  - Copyright and licensing information
  - Trademark policy

Release Notes

- Release notes index
- Percona Operator for MySQL 1.2.0 (2026-07-03)
- Percona Operator for MySQL 1.1.0 (2026-04-17)
- Percona Operator for MySQL 1.0.0 (2025-11-17)
- Percona Operator for MySQL 0.12.0 (2025-09-23)
- Percona Operator for MySQL 0.11.0 (2025-09-01)
- Percona Operator for MySQL 0.10.0 (2025-06-04)
- Percona Operator for MySQL 0.9.0 (2025-02-11)
- Percona Operator for MySQL 0.8.0 (2024-07-16)
- Percona Operator for MySQL 0.7.0 (2024-03-25)
- Percona Operator for MySQL 0.6.0 (2023-09-05)
- Percona Operator for MySQL 0.5.0 (2023-03-30)
- Percona Operator for MySQL 0.4.0 (2023-01-30)
- Percona Operator for MySQL 0.3.0 (2022-09-29)
- Percona Operator for MySQL 0.2.0 (2022-06-30)
- Percona Distribution for MySQL Operator based on Percona Server for MySQL 0.1.0 (2022-01-25)

Percona Operator for MySQL based on Percona Server for MySQL

Percona Operator for MySQL automates managing your MySQL databases on Kubernetes, making this process simple, reliable, and worry-free. Built on Percona Server for MySQL, the Operator brings enterprise-grade reliability, performance, and observability right out of the box.

With Percona Operator for MySQL, you can quickly set up, scale, and protect your databases using easy-to-understand configuration files. The Operator takes care of everyday tasks such as deployment, backups, updates, and failover, so you can focus on your applications and your business, not on manual database management.

Percona Operator for MySQL is generally available with **group replication**. Asynchronous replication is still in tech preview and is not recommended for production yet.

Get started ↓ | See what's new in version 1.2.0

Why choose Percona Operator for MySQL?

Deploy and manage with ease

No need for complicated scripts or manual setups. Define your database requirements in a YAML file, and have the Operator automatically create, configure, and manage your entire MySQL cluster.

Built for reliability

From day one, your database comes with robust features you need for production: high availability, automated backups, built-in monitoring, and strong security. Everything is ready to use right out of the box.

Cloud-native by design

Whether you use AWS, Google Cloud or any other Kubernetes platform, the Operator fits right in. Enjoy a consistent, cloud-native MySQL experience everywhere.

Get started today

Set up Percona Operator for MySQL in just a few minutes. Start with our simple guides and begin managing your databases with confidence.

Quickstart guide →

Discover the Operator

Learn about all the features Percona Operator for MySQL offers, how it works, and how it can help you.

Features →

Security you can trust

Your data safety is our priority. See how our Operator protects your information with advanced security and encryption options.

Security features →

Backup management

Learn how to keep your MySQL databases backed up and ready for a quick restore whenever you need it.

Backup options →

Troubleshooting

Need assistance? Our troubleshooting guides cover common questions and step-by-step solutions.

Diagnostics →

Get help from Percona

Our documentation guides are packed with information, but they can't cover everything you need to know about Percona Operator for MySQL based on Percona Server for MySQL. They also won't cover every scenario you might come across. Don't be afraid to try things out and ask questions when you get stuck.

Percona's Community Forum

Be a part of a space where you can tap into a wealth of knowledge from other database enthusiasts and experts who work with Percona's software every day. While our service is entirely free, keep in mind that response times can vary depending on the complexity of the question. You are engaging with people who genuinely love solving database challenges.

We recommend visiting our Community Forum. It's an excellent place for discussions, technical insights, and support around Percona database software. If you're new and feeling a bit unsure, our FAQ and Guide for New Users ease you in.

If you have thoughts, feedback, or ideas, the community team would like to hear from you at Any ideas on how to make the forum better?. We're always excited to connect and improve everyone's experience.

Percona experts

Percona experts bring years of experience in tackling tough database performance issues and design challenges.

Talk to a Percona Expert

We understand your challenges when managing complex database environments. That's why we offer various services to help you simplify your operations and achieve your goals.

Service	Description
24/7 Expert Support	Our dedicated team of database experts is available 24/7 to assist you with any database issues. We provide flexible support plans tailored to your specific needs.
Hands-On	Our managed services team can take over the day-to-day management of your database

Database Management	infrastructure, freeing up your time to focus on other priorities.
Expert Consulting	Our experienced consultants provide guidance on database topics like architecture design, migration planning, performance optimization, and security best practices.
Comprehensive Training	Our training programs help your team develop skills to manage databases effectively, offering virtual and in-person courses.

We’re here to help you every step of the way. Whether you need a quick fix or a long-term partnership, we’re ready to provide your expertise and support.

Understand the Operator

Features

Percona Operator for MySQL is a Kubernetes-native controller that automatically manages the full lifecycle of Percona Server for MySQL clusters. The Operator offloads your teams from manual day-to-day database management operations empowering them to focus on tasks that matter instead.

Core capabilities

Here's what the Operator brings to your infrastructure:

High availability and failover

Never lose sleep over database downtime again. The Operator provides robust high availability through:

- Automatic failover with intelligent primary election handled by the Orchestrator
- Multi-node deployments with anti-affinity rules to prevent single points of failure
- Health monitoring with automatic recovery from node failures
- Zero-downtime upgrades with rolling update strategies

Choose between group replication (GA) for stronger consistency or asynchronous replication (tech preview) for lower latency—both with automatic failover capabilities.

Cross-site replication

Link Group Replication clusters across regions, availability zones, or Kubernetes environments into the InnoDB ClusterSet managed by the Operator. Instead of building custom replication tooling between sites, you apply a `PerconaServerMySQLClusterSet` Custom Resource and let the Operator handle topology lifecycle.

Cross-site replication requires Percona Operator for MySQL 1.2.0 or later and Group Replication clusters on Percona Server for MySQL 8.0.27+ or 8.4. Learn more in the cross-site replication documentation.

Automated backup and restore flows

Protect your data with Percona XtraBackup - an enterprise-grade backup solution for hot, non blocking backups. Run:

- Scheduled backups with configurable retention policies
- On-demand backups for critical operations
- Encrypted backups for compliance-ready protection in cloud object storage

Automated scaling and resource management

Scale your database infrastructure effortlessly:

- Horizontal scaling with automatic replica management
- Vertical scaling with resource limit adjustments
- Storage expansion with volume growth capabilities
- Load balancing through HAProxy or MySQL Router
- Resource optimization with intelligent pod placement

Security and compliance

Keep your data secure with built-in security features:

- Transport encryption with TLS/SSL support
- Data-at-rest encryption with key management integration
- Encrypted backups for object storage
- Role-based access control with fine-grained permissions
- Secret management with Kubernetes-native secrets
- Orchestrator API authentication for async replication clusters (available starting with Operator version 1.2.0)

Monitoring and observability

Gain deep insights into your database performance:

- Percona Monitoring and Management (PMM) integration for comprehensive monitoring
- Custom metrics and alerting capabilities
- Log aggregation and centralized logging

- Performance insights with query analysis
- Health dashboards for operational visibility

How Operator works

The Operator extends Kubernetes with custom resources that represent your MySQL cluster's desired state.

Here's what happens under the hood:

1. You define your cluster requirements in a `PerconaServerMySQL` custom resource
2. The Operator watches for changes and reconciles the actual state with your desired state
3. Kubernetes resources are automatically created and managed (Pods, Services, StatefulSets, etc.)
4. The cluster self-heals by detecting and recovering from failures
5. Updates and scaling happen automatically based on your configuration changes

This declarative approach means you describe what you want, not how to achieve it. The Operator handles all the complex orchestration, ensuring your database cluster always matches your specifications.

Explore the architecture →

What's next?

- Quickstart guides - Get up and running in minutes
- Installation options - Deploy on your preferred platform
- Backup and restore - Protect your data with automated backups
- Monitoring setup - Gain visibility into your database performance
- Security configuration - Secure your database communications
- Cross-site replication - Link Group Replication clusters across regions and sites

How the Operator works

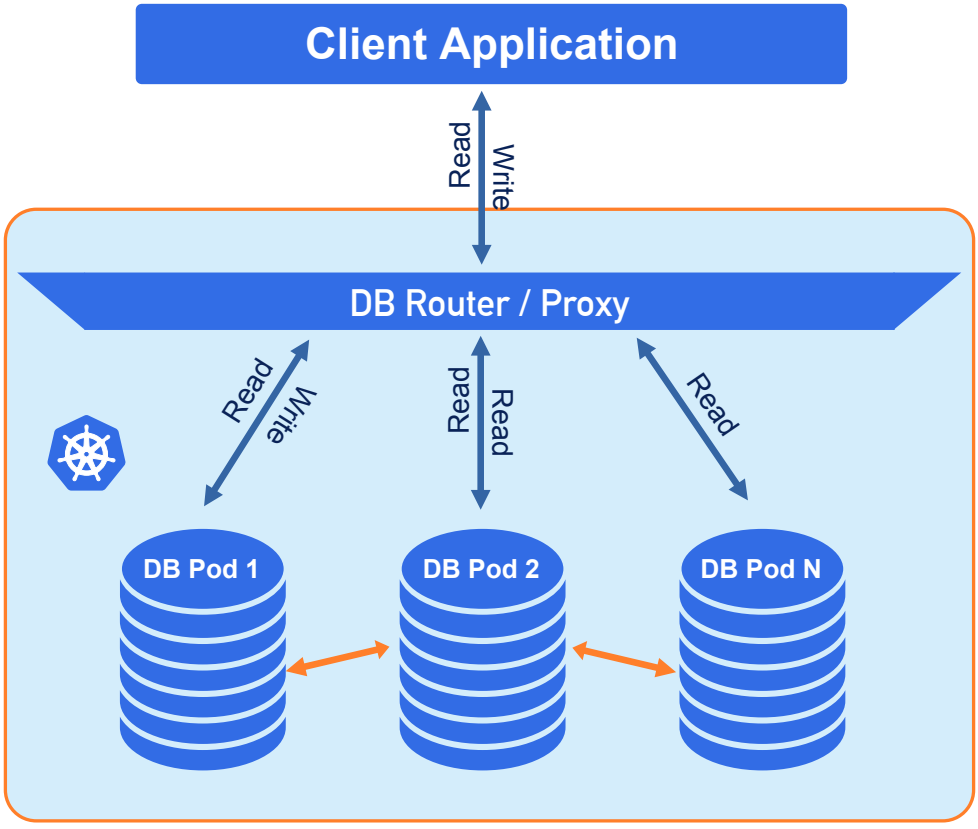
The Percona Operator for MySQL acts as your assistant in managing databases on Kubernetes. It extends the Kubernetes API with a custom `PerconaServerMySQL` resource. Think of it as a blueprint that defines how you want your MySQL database to look and behave.

Whenever you create or update a `PerconaServerMySQL` Custom Resource, the Operator steps in and handles the hard work for you. It automatically does the following:

1. Creates and manages the necessary Kubernetes resources (StatefulSets, Services, Pods)
2. Ensures your cluster matches the desired state you've defined
3. Monitors the cluster health and automatically recovers from failures
4. Coordinates upgrades and scaling operations

These operations ensure that your actual database environment always matches your request.

Each MySQL node in your cluster contains a complete copy of your data, synchronized across all nodes.



The recommended configuration is to use at least 3 nodes. Such setup provides high availability — if any node fails, the cluster continues operating normally. Read more about high-availability

To keep your data safe and persistent, the Operator uses Kubernetes storage systems called Persistent Volumes (PVs) and PersistentVolumeClaims (PVCs). When you request storage for your database, a PVC automatically finds and attaches available storage for you. If a node fails, the Kubernetes storage system can move your data to another node, making sure your database remains available and your data stays protected.

Ready to get started? Continue to the quickstart guide to deploy your first cluster, or explore the architecture overview to understand the inner workings of the Operator. For hands-on steps and best practices, check out What next?.

Architecture

Percona Operator for MySQL automates deploying and operating Percona Server for MySQL clusters on Kubernetes. This document explains what components the Operator uses and how they work together to provide a highly available MySQL database. Also, read more about How the Operator works.

Components

The StatefulSet deployed with the Operator includes the following components:

- Percona Server for MySQL - a free, fully compatible, enhanced, and open source drop-in replacement for any MySQL database
- Percona XtraBackup - a hot backup utility for MySQL based servers that doesn't lock your database during the backup
- Orchestrator - a replication topology manager for MySQL used when asynchronous replication between MySQL instances is turned on,
- HAProxy - a proxy and load balancing service serving as the entry point to your database cluster. It is compatible with both asynchronous replication and group replication between MySQL instances,
- MySQL Router - a proxy solution which can be used instead of HAProxy for MySQL clusters with group replication is turned on,
- Percona Toolkit - a set of tools for debugging MySQL Pods.

It can also include sidecar containers such as PMM Client or your custom ones. This depends on how you further fine-tune your cluster. Learn more about sidecar containers.

Replication types

Each MySQL node in your cluster contains a complete copy of your data, replicated across all nodes.

The Operator supports two replication types, each with different characteristics for performance, consistency, and availability. You choose the replication type when configuring your cluster.

Asynchronous replication (tech preview)

With asynchronous replication, writes complete on the primary instance without waiting for replicas. After a write completes, the primary records the change in its binary log, and replicas apply these changes independently.

Characteristics:

- **Performance** - Asynchronous replication provides faster write operations with lower latency.
- **Read scaling** - You can distribute application read requests to different replica instances, improving read throughput.
- **Consistency** - Eventual consistency: replicas may lag behind the primary instance, which can affect applications requiring real-time data. There is a risk that some transactions committed on the primary may be lost if it fails before replicas catch up.
- **Write scaling** - Does not allow for horizontal write scaling; scaling writes relies on vertical scaling, which is increasing the resources (RAM, CPU) of the primary instance, rather than on adding more write nodes.
- **Failover** - Orchestrator handles automatic primary election and replication topology recovery.
- **Status** - Currently in tech preview and not recommended for production use.

Group replication

With group replication, write transactions require consensus from the group before completing. Read transactions can execute on any instance, while writes only occur on the primary.

Characteristics:

- **Consistency** – Provides strong consistency, and when set to a high transaction consistency level, helps prevent stale reads.
- **Read scaling** – Enables horizontal scaling of reads without stale reads when set with a high transaction consistency level.
- **Performance** – Write operations are slower than asynchronous replication due to the group consensus mechanism.
- **Failover** – Built-in native group membership protocol automatically handles member recovery and primary election.
- **Limitations** – Group replication limits the cluster to a maximum of 9 MySQL instances per group. Large transactions can noticeably slow down the system, and especially large transactions may even trigger a replication member fault if the transaction message cannot be copied between group members within a 5-second network window.

- **Status** – General Availability (GA) and recommended for production use.

> **Note**
>
> MySQL documentation may also use the terms “source/replica” instead of “primary/replica”.

Proxy solutions

The proxy you use depends on your replication type and requirements:

- **HAProxy** - Works with both asynchronous replication and group replication. Provides load balancing, health checks, and connection pooling.
- **MySQL Router** - Available only for group replication. Offers intelligent routing, connection pooling, and read-write splitting capabilities.

Replication type and proxy comparison

Feature	Asynchronous replication + HAProxy	Group Replication + HAProxy/MySQL Router
Writes	Single primary	Single primary
Read scaling	Yes	Yes
Write scaling	No	No
Consistency	Eventual on replicas	Stronger (depending the Transaction Consistency)
Latency	Low	Higher (due to sync)
Failover	Orchestrator elects new primary	Native group membership
Max nodes	Higher (practical limits)	9 per group
Proxy	HAProxy	HAProxy or MySQL Router

Tip: Choose Group Replication for stronger consistency and read scaling; choose asynchronous replication for lower write latency and simpler topology when it reaches GA status.

You can change the replication type if needed. Refer to the Change replication type guide for step-by-step instructions. Note that replication type change is not supported on a running cluster.

High availability

The Operator provides high availability through multiple layers of protection:

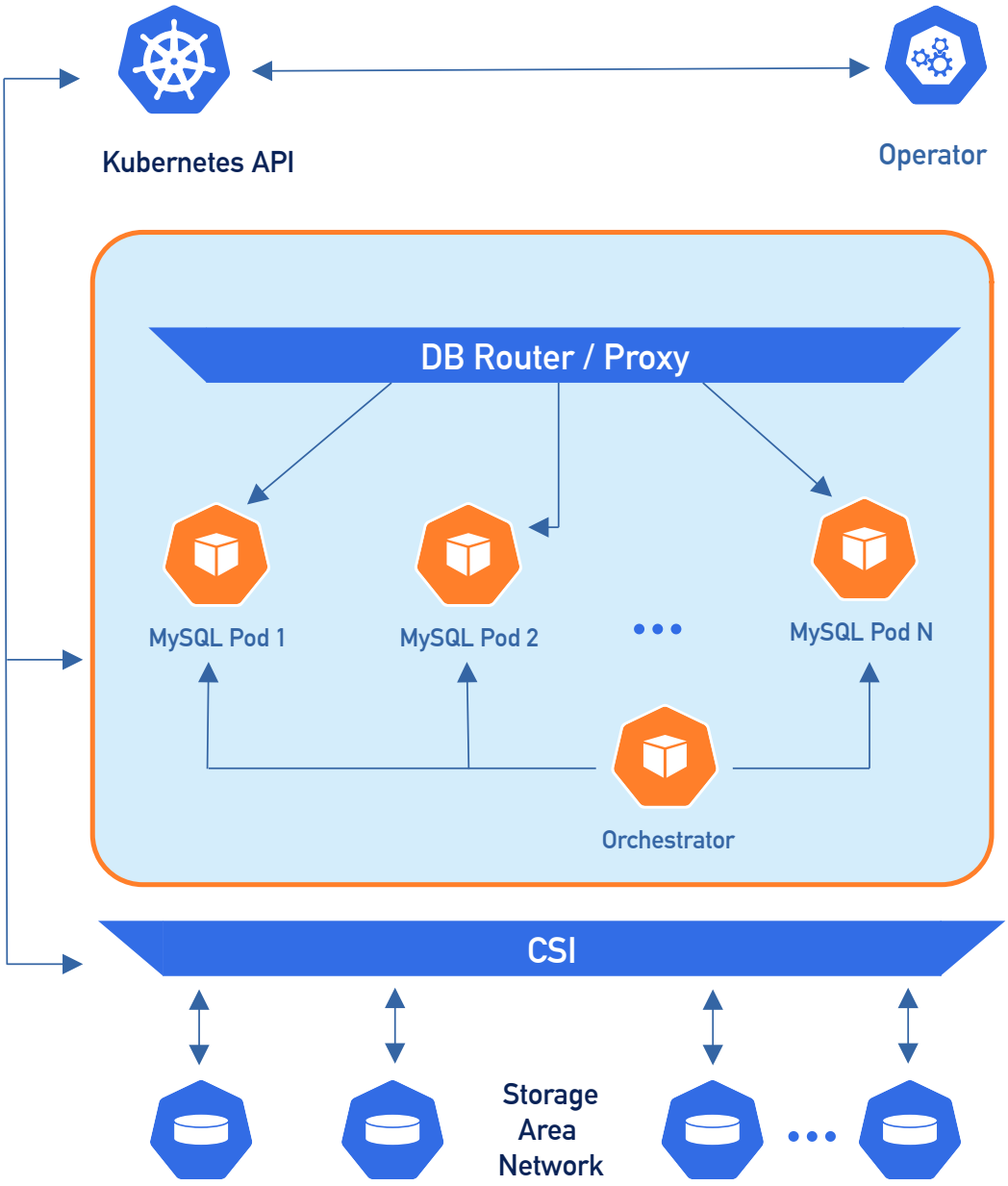
Pod distribution

The Operator uses node affinity to distribute Percona Server for MySQL instances across separate worker nodes when possible. This prevents a single node failure from taking down multiple database instances.

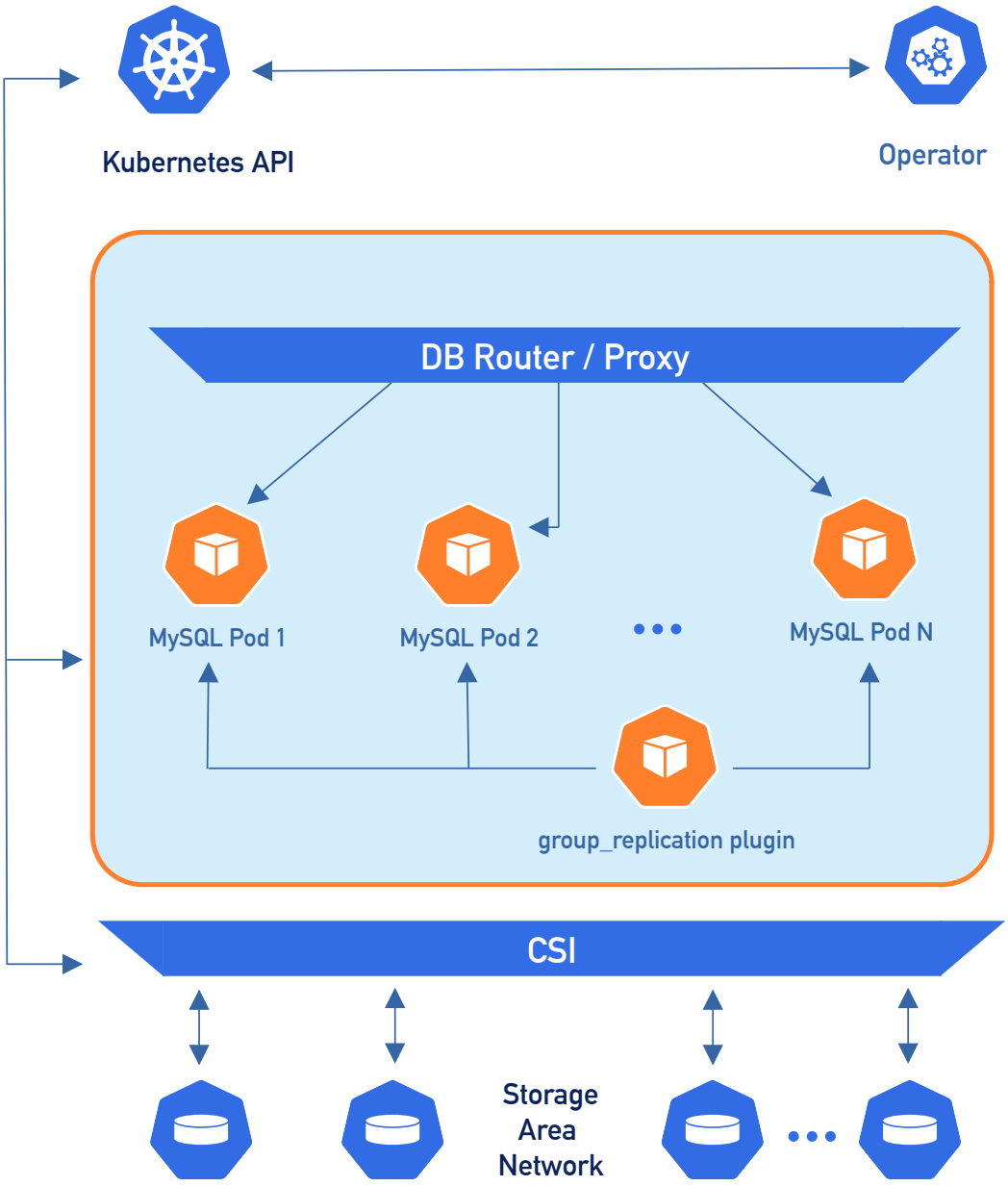
Automatic recovery

If a node fails, Kubernetes automatically reschedules the affected Pod on another healthy node. Inside your cluster, automatic recovery is handled as follows:

- In **asynchronous replication** clusters, the Orchestrator detects the failure, promotes a healthy replica to primary, and updates the replication topology.



- In **group replication** clusters, the native group membership protocol automatically handles member removal, primary election, and topology recovery.



Client connectivity

Clients connect through HAProxy or MySQL Router, which automatically route traffic to healthy MySQL instances. These proxies detect failures and redirect connections away from failed nodes, ensuring your applications always connect to available database instances.

For configuration details, see:

- HAProxy configuration
- MySQL Router configuration

What to read next

Now that you understand the architecture, explore these topics:

- Backups - Understand backup and restore operations
- Scaling - Scale your cluster horizontally or vertically
- High availability configuration - Configure anti-affinity and pod distribution
- Updating and upgrades - Keep your cluster up to date
- Operator Custom Resource reference - Description of available configuration options

Quick start

Overview

Ready to get started with the Percona Operator for MySQL? In this section, you will learn some basic operations, such as:

- Install and deploy an Operator
- Connect to the MySQL instance in your database cluster
- Insert sample data to the database
- Set up and make a logical backup
- Monitor the database health with Percona Monitoring and Management (PMM)

Next steps

Install the Operator →

System requirements

The Operator was developed and tested with the following software:

- Percona Server for MySQL 8.4.10-10.1
- Percona Server for MySQL 8.0.46-37.1
- XtraBackup 8.4.0-6.1
- XtraBackup 8.0.35-36.1
- MySQL Router 8.4.10
- MySQL Router 8.0.46
- HAProxy 2.8.18-1
- Orchestrator 3.2.6-22
- Percona Toolkit 3.7.1-4
- PMM Client 3.8.1
- Cert Manager 1.20.2
- Percona Binlog Server 0.3.1

Other options may also work but have not been tested.

Supported platforms

Percona Operators are designed for compatibility with all CNCF-certified Kubernetes distributions. Our release process includes targeted testing and validation on major cloud provider platforms and OpenShift, as detailed below for Operator version 1.2.0:

- Google Kubernetes Engine (GKE) 1.33 - 1.35
- Amazon Elastic Kubernetes Service (EKS) 1.33 - 1.36
- OpenShift 4.18 - 4.22
- Minikube 1.38.1 with Kubernetes v1.35.1

Other Kubernetes platforms may also work but have not been tested.

Resource limits

A cluster running an officially supported platform contains at least three Nodes, with the following resources:

- 2GB of RAM,
- 2 CPU threads per Node for Pods provisioning,
- at least 60GB of available storage for Persistent Volumes provisioning.

Installation guidelines

Choose how you wish to install the Operator:

- with Helm
- on Minikube
- on Google Kubernetes Engine (GKE)
- on Amazon Elastic Kubernetes Service (AWS EKS)
- in a Kubernetes-based environment

2. Install the Operator

Install Percona Server for MySQL using Helm

Helm is the package manager for Kubernetes. Percona Helm charts can be found in percona/percona-helm-charts repository on Github.

Prerequisites

Install Helm v3 and above following its official installation instructions.

Installation

1. Add the Percona Helm charts repository and make your Helm client up to date with it:

```
helm repo add percona https://percona.github.io/percona-helm-charts/
helm repo update
```

2. Install the Percona Operator for MySQL. It is a good practice to isolate workloads in Kubernetes by installing the Operator in a custom namespace. Replace the `my-namespace` value with your desired namespace in the following command:

```
helm install my-op percona/ps-operator --namespace my-namespace --create-namespace
```

The `my-op` parameter in the above example is the name of a new release object which is created for the Operator when you install its Helm chart. You can use any other name you like instead.

Expected output

```
NAME: my-op
LAST DEPLOYED: Sun Nov  9 10:23:13 2025
NAMESPACE: my-namespace
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

3. Install Percona Server for MySQL:

```
helm install my-db percona/ps-db --namespace my-namespace
```

The `my-db` parameter in the above example is the name of a new release object which is created for the Percona Server for MySQL when you install its Helm chart (use any name you like).

Expected output

```
NAME: my-db
LAST DEPLOYED: Sun Nov  9 10:26:41 2025
NAMESPACE: my-namespace
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
```

The command above installs Percona Server for MySQL with default parameters. Custom options can be passed to a `helm install` command as a `--set key=value[,key=value]` argument. The options passed with a chart can be any of the Custom Resource options.

The following example will deploy a Percona Server for MySQL in the `my-namespace` namespace, with disabled backups and 20 Gi storage:

```
helm install my-db percona/ps-db \
  --set mysql.volumeSpec.pvc.resources.requests.storage=20Gi \
  --set backup.enabled=false
```

You can find in the documentation for the charts which Operator and database parameters can be customized during installation.

Next steps

Connect to Percona Server for MySQL →

Install Percona Server for MySQL cluster using kubectl

A Kubernetes Operator is a special type of controller introduced to simplify complex deployments. The Operator extends the Kubernetes API with custom resources.

The Percona Operator for MySQL is based on best practices for configuration and setup of a Percona Server for MySQL in a Kubernetes-based environment on-premises or in the cloud.

We recommend installing the Operator with the kubectl command line utility. It is the universal way to interact with Kubernetes. Alternatively, you can install it using the Helm package manager.

Install with kubectl ↓ Install with Helm →

Prerequisites

To install Percona Server for MySQL cluster, you need the following:

1. The **kubectl** tool to manage and deploy applications on Kubernetes, included in most Kubernetes distributions. If not already installed, follow its official installation instructions.
2. A Kubernetes environment. You can deploy it on Minikube for testing purposes or using any cloud provider of your choice. Check the list of our officially supported platforms.

See also

- Set up Minikube
- Create and configure the GKE cluster
- Set up Amazon Elastic Kubernetes Service

Procedure

Here's a sequence of steps to follow:

1. Create the Kubernetes namespace for your cluster. It is a good practice to isolate workloads in Kubernetes by installing the Operator in a custom namespace. Replace the `<namespace>` placeholder with your value.

```
$ kubectl create namespace <namespace>
```

Expected output

```
namespace/<namespace> was created
```

2. Deploy the Operator with the following command:

```
$ kubectl apply -f https://raw.githubusercontent.com/percona/percona-server-mysql-operator/v1.2.0/deploy/bundle.yaml  -n <namespace>
```

Expected output

```
customresourcedefinition.apiextensions.k8s.io/perconaservermysqlbackups.ps.percona.com created
customresourcedefinition.apiextensions.k8s.io/perconaservermysqlrestores.ps.percona.com created
customresourcedefinition.apiextensions.k8s.io/perconaservermysqls.ps.percona.com created
serviceaccount/percona-server-mysql-operator created
role.rbac.authorization.k8s.io/percona-server-mysql-operator-leaderelection created
role.rbac.authorization.k8s.io/percona-server-mysql-operator created
rolebinding.rbac.authorization.k8s.io/percona-server-mysql-operator-leaderelection created
rolebinding.rbac.authorization.k8s.io/percona-server-mysql-operator created
configmap/percona-server-mysql-operator-config created
deployment.apps/percona-server-mysql-operator created
```

As the result you will have the Operator Pod up and running.

3. Deploy Percona Server for MySQL cluster:

```
$ kubectl apply -f https://raw.githubusercontent.com/percona/percona-server-mysql-operator/v1.2.0/deploy/cr.yaml -n <namespace>
```

Expected output

```
perconaservermysql.ps.percona.com/ps-cluster1 created
```

4. Check the Operator and the database Pods status.

```
$ kubectl get ps -n <namespace>
```

Expected output

```
NAME         REPLICATION          ENDPOINT                      STATE   MYSQL
ORCHESTRATOR   HAPROXY   ROUTER   AGE
ps-cluster1   group-replication   ps-cluster1-haproxy.<namespace>   ready   3
3        20m
```

The creation process may take some time. When the process is over your cluster obtains the `ready` status.

You have successfully installed and deployed the Operator with default parameters.

The default configuration includes three HAProxy and three Percona Server for MySQL instances.

You can check the rest of the Operator's parameters in the Custom Resource options reference.

Next steps

Connect to Percona Server for MySQL →

Install Percona Operator for MySQL with customized parameters

You can customize the configuration of Percona Server for MySQL and install it with customized parameters.

To check available configuration options, see deploy/cr.yaml and Custom Resource Options.

Note

Deploy the Operator in your namespace before you apply a customized cluster manifest. If you have not installed it yet, follow Install with kubectl or Install with Helm.

kubectl

To customize the configuration when installing with `kubectl`, do the following:

1. Clone the repository with all manifests and source code by executing the following command:

   ```
   git clone -b v1.2.0 https://github.com/percona/percona-server-mysql-operator
   ```

2. Edit the required options and apply your modified `deploy/cr.yaml` file as follows:

   ```
   kubectl apply -f deploy/cr.yaml -n <namespace>
   ```

Helm

You can install the Operator deployment and the Percona Server for MySQL cluster with custom parameters using Helm. Find what options you can customize in the Operator chart documentation and the Percona Server for MySQL chart documentation.

You can provide custom parameters to Helm using either the `--set` flag or a `values.yaml` file. The `--set` flag is convenient for overriding a small number of parameters directly in the command line, while a `values.yaml` file is preferable when you want to manage many custom settings in one place. Both methods are fully supported by Helm and can be used as needed for your deployment.

Using `--set` flags

To pass a custom parameter to Helm, use the `--set key=value` flag with the `helm install` command.

For example, to enable Percona Monitoring and Management (PMM) for the database cluster, run:

```
helm install my-db percona/ps-db --version 1.2.0 --namespace my-namespace \
  --set mysql.image.tag=8.4.10-10.1 \
  --set pmm.enabled=true
```

Using a `values.yaml` file

Create a `values.yaml` file with your custom parameters and pass it to `helm install` with the `-f` or `--values` flag:

```
helm install my-db percona/ps-db --version 1.2.0 --namespace my-namespace -f
values.yaml
```

Example `values.yaml`:

```
mysql:
  image:
    tag: 8.4.10-10.1
pmm:
  enabled: true
```

Naming conventions for Helm resources

When you install a chart, Helm creates a release and uses the release name and chart name to generate resource names. By default, resources are named `release-name-chart-name`.

You can override the default naming with the `nameOverride` or `fullnameOverride` options. Pass them using the `--set` flag or in your `values.yaml` file.

Option	Effect	Example
`nameOverrid e`	Replaces the chart name but keeps the release name in the generated name	`release-name-name- override`

`fullnameOverride`	Replaces the entire generated name with the specified value	`fullname-override`

Using `nameOverride` — replaces the chart name but keeps the release name:

```
helm install my-operator percona/ps-operator --namespace my-namespace \
  --set nameOverride=mysql-operator
```

Deployment name: `my-operator-mysql-operator`.

```
helm install cluster1 percona/ps-db -n my-namespace \
  --set nameOverride=mysql
```

Cluster name: `cluster1-mysql`.

Using `fullnameOverride` — replaces the full resource name:

```
helm install my-operator percona/ps-operator --namespace my-namespace \
  --set fullnameOverride=percona-server-mysql-operator
```

Deployment name: `percona-server-mysql-operator`.

```
helm install cluster1 percona/ps-db -n my-namespace \
  --set fullnameOverride=my-db
```

Cluster name: `my-db`.

Cluster naming

Use names that satisfy Kubernetes naming rules for resources and DNS labels. If you use long release names together with `nameOverride` or `fullnameOverride`, ensure the resulting names stay within length limits your environment allows.

Common Helm values reference

The following table lists commonly used values for the Operator and database charts. For the full list of options, see the chart values files.

Value	Charts	Description

`nameOverride`	ps-operator, ps-db	Replaces the chart name in generated resource names
`fullnameOverride`	ps-operator, ps-db	Replaces the entire generated resource name
`watchAllNamespaces`	ps-operator	Deploy the Operator in cluster-wide mode to watch all namespaces
`disableTelemetry`	ps-operator	Disable telemetry collection. See Telemetry for details

Connect to Percona Server for MySQL

In this tutorial, you will connect to the Percona Server for MySQL you deployed previously.

The Operator creates a dedicated Secret named `<cluster_name>-psuser-root` that contains all connection details for the `root` user: hostname, port, username, password, and ready-to-use URIs. The Operator keeps this Secret up to date on every reconciliation.

Here's how to connect:

1. Export the namespace, cluster name and the Secret name as environment variables:

   ```
   export NAMESPACE=my-namespace
   export CLUSTER_NAME=ps-cluster1
   export SECRET_NAME=${CLUSTER_NAME}-psuser-root
   ```

   Replace `ps-cluster1` with the name of your cluster if you changed it during installation.

2. Verify that the connection Secret exists:

   ```
   kubectl get secret $SECRET_NAME -n $NAMESPACE
   ```

   Look for the Secret named `<cluster_name>-psuser-root`. The default name differs on how you installed the Operator:

   via kubectl

   `ps-cluster1-psuser-root`

   via Helm

   `my-db-ps-db-psuser-root`

3. Retrieve the user credentials from the Secret:

   `bash`

   ```
   kubectl get secret "$SECRET_NAME" -n "$NAMESPACE" \
     -o jsonpath='{.data.user}' | base64 --decode && echo
   kubectl get secret "$SECRET_NAME" -n "$NAMESPACE" \
     -o jsonpath='{.data.password}' | base64 --decode && echo
   ```

4. Run a container with the `mysql` client and connect its console output to your terminal. The following command does this, naming the new Pod `percona-client`:

```
kubectl run -n $NAMESPACE -i --rm --tty percona-client \
--image=percona/percona-server:8.4 --restart=Never -- bash -il
```

Executing it may require some time to deploy the corresponding Pod.

5. Connect to Percona Server for MySQL. To do this, run `mysql` tool in the `percona-client` command shell using your cluster name and the password obtained from the secret instead of the `<root_password>` placeholder. The command will look different depending on whether your cluster uses load balancing with HAProxy (the default behavior) or uses MySQL Router (can be used with Group Replication clusters only):

with HAProxy (default)

```
mysql -h <cluster_name>-haproxy -uroot -p'<root_password>'
```

with MySQL Router

```
mysql -h <cluster_name>-router -uroot -p'<root_password>'
```

Congratulations! You have connected to Percona Server for MySQL.

Next steps

Insert sample data →

Insert sample data

In this tutorial you will learn how to insert sample data into Percona Server for MySQL.

We will enter SQL statements via the same MySQL shell we used to connect to the database.

1. Let's create a separate database for our experiments:

```
CREATE DATABASE mydb;
use mydb;
```

Output

```
Query OK, 1 row affected (0.01 sec)
Database changed
```

2. Now let's create a table which we will later fill with some sample data:

```
CREATE TABLE extraordinary_gentlemen (
    id int NOT NULL AUTO_INCREMENT,
    name varchar(255) NOT NULL,
    occupation varchar(255),
    PRIMARY KEY (id)
);
```

Output

```
Query OK, 0 rows affected (0.04 sec)
```

3. Adding data to the newly created table will look as follows:

```
INSERT INTO extraordinary_gentlemen (name, occupation)
  VALUES
  ("Allan Quartermain","hunter"),
  ("Nemo","fish"),
  ("Dorian Gray", NULL),
  ("Tom Sawyer", "secret service agent");
```

Output

```
Query OK, 4 rows affected (0.01 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

4. Query the database to verify the data insertion

```
SELECT *
FROM extraordinary_gentlemen;
```

Output

```
+----+------------------+----------------------+
| id | name             | occupation           |
+----+------------------+----------------------+
|  1 | Allan Quartermain | hunter               |
|  2 | Nemo             | fish                 |
|  3 | Dorian Gray      | NULL                 |
|  4 | Tom Sawyer       | secret service agent |
+----+------------------+----------------------+
```

5. To update data in the database, execute the following statement:

```
UPDATE  extraordinary_gentlemen
   SET occupation = "submariner"
WHERE name = "Nemo";
```

Output

```
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

6. Now if you repeat the SQL statement from step 4, you will see the changes take effect:

```
SELECT *
FROM extraordinary_gentlemen;
```

Output

```
+----+-------------------+----------------------+
| id | name              | occupation           |
+----+-------------------+----------------------+
|  1 | Allan Quartermain | hunter               |
|  2 | Nemo              | submariner           |
|  3 | Dorian Gray       | NULL                 |
|  4 | Tom Sawyer        | secret service agent |
+----+-------------------+----------------------+
```

Next steps

Make a backup →

Make a backup

In this tutorial, you will learn how to make a logical backup of your data manually. To learn more about backups, see the Backup and restore section.

Considerations and prerequisites

In this tutorial, we use the AWS S3 as the backup storage. You need the following S3-related information:

- the name of the S3 storage
- the name of the S3 bucket
- the region - the location of the bucket
- the S3 credentials to be used to access the storage.

If you don't have access to AWS, you can use any S3-compatible storage like MinIO. Also check the list of supported storages.

Also, we will use some files from the Operator repository for setting up backups. So, clone the percona-server-mysql-operator repository:

```
git clone -b v1.2.0 https://github.com/percona/percona-server-mysql-operator
cd percona-server-mysql-operator
```

Note

It is important to specify the right branch with `-b` option while cloning the code on this step. Please be careful.

Configure backup storage

1. Encode S3 credentials, substituting `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` with your real values:

on Linux

```
echo -n 'AWS_ACCESS_KEY_ID' | base64 --wrap=0
echo -n 'AWS_SECRET_ACCESS_KEY' | base64 --wrap=0
```

on MacOS

```
echo -n 'AWS_ACCESS_KEY_ID' | base64
echo -n 'AWS_SECRET_ACCESS_KEY' | base64
```

2. Edit the deploy/backup/backup-s3.yaml example Secrets configuration file and specify the following:

   - the `metadata.name` key is the name which you use to refer your Kubernetes Secret
   - the base64-encoded S3 credentials

deploy/backup/backup-secret-s3.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: ps-cluster1-s3-credentials
type: Opaque
data:
  AWS_ACCESS_KEY_ID: <YOUR_AWS_ACCESS_KEY_ID>
  AWS_SECRET_ACCESS_KEY: <YOUR_AWS_SECRET_ACCESS_KEY>
```

3. Create the Secrets object from this yaml file. Specify your namespace instead of the `<namespace>` placeholder:

```
kubectl apply -f deploy/backup/backup-secret-s3.yaml -n <namespace>
```

4. Update your `deploy/cr.yaml` configuration. Specify the following parameters in the `backup` section:

   - set the `storages.<NAME>.type` to `s3`. Substitute the `<NAME>` part with some arbitrary name that you will later use to refer this storage when making backups and restores.
   - set the `storages.<NAME>.s3.credentialsSecret` to the name you used to refer your Kubernetes Secret (`ps-cluster1-s3-credentials` in the previous step).

- specify the S3 bucket name for the `storages.<NAME>.s3.bucket` option
- specify the region in the `storages.<NAME>.s3.region` option. Also you can use the `storages.<NAME>.s3.prefix` option to specify the path (a sub-folder) to the backups inside the S3 bucket. If prefix is not set, backups are stored in the root directory.

```
...
backup:
  enabled: true
  ...
  storages:
    s3-us-west:
      type: s3
      s3:
        bucket: S3-BACKUP-BUCKET-NAME-HERE
        region: us-west-2
        credentialsSecret: ps-cluster1-s3-credentials
      ...
```

If you use a different S3-compatible storage instead of AWS S3, add the `endpointURL` key in the `s3` subsection, which should point to the actual cloud used for backups. This value is specific to the cloud provider. For example, using Google Cloud involves the following `endpointUrl`:

```
endpointUrl: https://storage.googleapis.com
```

5. Apply the configuration. Specify your namespace instead of the `<namespace>` placeholder:

```
kubectl apply -f deploy/cr.yaml -n <namespace>
```

Make a backup

Now that you have the configured storage in your Custom Resource, you can make your first backup.

1. To make a backup, you need the configuration file. Edit the sample deploy/backup/backup.yaml configuration file and specify the following:
   - `metadata.name` - specify the backup name. You will use this name to restore from this backup

- `spec.clusterName` - specify the name of your cluster. This is the name you specified when deploying Percona Server for MySQL cluster.
- `spec.storageName` - specify the name of your already configured storage.

Here's the example configuration:

deploy/backup/backup.yaml

```
apiVersion: ps.percona.com/v1
kind: PerconaServerMySQLBackup
metadata:
  name: backup1
  finalizers:
    - percona.com/delete-backup
spec:
  clusterName: ps-cluster1
  storageName: s3-us-west
```

2. Apply the configuration. This instructs the Operator to start a backup. Specify your namespace instead of the `<namespace>` placeholder:

```
kubectl apply -f deploy/backup/backup.yaml -n <namespace>
```

3. Track the backup progress.

```
kubectl get ps-backup -n <namespace>
```

Output

```
NAME      CLUSTER       STORAGE      DESTINATION
STATUS    COMPLETED   AGE
backup1   ps-cluster1      s3-us-west   s3://ps-operator-testing/2023-10-
10T16:36:46Z   Running               43s
```

When the status changes to `Succeeded`, backup is made.

Troubleshooting

You may face issues with the backup. To identify the issue, you can do the following:

1. View the information about the backup with the following command:

   ```
   kubectl get ps-backup <backup-name> -n <namespace> -o yaml
   ```

2. View the backup-agent logs. Use the command from step 1 to find the name of the pod where the backup was made. Check for the information in the `.status.backupSource` field to find the Pod where the backup process was run. To view the logs, run the following command:

   ```
   kubectl logs pod/<pod-name> -c xtrabackup -n <namespace>
   ```

Congratulations! You have made the first backup manually. Want to learn more about backups? See the Backup and restore section for how to restore from a previously saved backup.

Next steps

Monitor the database →

Monitor database with Percona Monitoring and Management (PMM)

In this section you will learn how to monitor Percona Server for MySQL cluster with Percona Monitoring and Management (PMM).

PMM is a client/server application. It includes the PMM Server and the number of PMM Clients running on each node with the database you wish to monitor.

A PMM Client collects server metrics, general system metrics, query analytics and sends it to the server As a user, you connect to the PMM Server to see database metrics on a number of dashboards.

PMM Server and PMM Client are installed separately.

Considerations

1. Starting with the version 0.10.0, the Operator supports only PMM 3.x versions. The support for PMM 2.x is dropped.
2. You must run the Operator version 0.10.0 and later to monitor your database with PMM 3.x. Check the Upgrade the Operator tutorial for the update steps.
3. To use PMM3, PMM Server version must be equal to or newer than the PMM Client.

Install PMM Server

You must have PMM Server up and running. You can run PMM Server as a *Docker container*, a *virtual appliance*, or on an *AWS instance*. Please refer to the official PMM documentation for the installation instructions.

For Kubernetes environment, we recommend to install PMM from the Helm chart.

Install PMM Client

PMM Client is installed as a sidecar container in the database Pods in your Kubernetes-based environment. To install PMM Client, do the following:

1. Authorize PMM Client within PMM Server.

    1. PMM3 uses Grafana service accounts to control access to PMM server components and resources. To authenticate in PMM server, you need a service account token. Use PMM documentation to generate a service account with the **Admin** role and token.

        The token must have the format `glsa_**********_9e35351b`.

        > **Warning**
        >
        > When you create a service account token, you can select its lifetime: it can be either a permanent token that never expires or the one with the expiration date. PMM server cannot rotate service account tokens after they expire. So you must take care of reconfiguring PMM Client in this case.

    2. Add the service account token to the `pmmservertoken` option in the users Secrets object. Use the following command and replace the `<my-token>` placeholder with your value:

        in Linux

        ```
        kubectl patch secret/ps-cluster1-secrets -p "$(echo -n '{"data":
        {"pmmservertoken":"'$(echo -n <my-token> | base64 --wrap=0)'"}}')"
        ```

        in macOS

        ```
        kubectl patch secret/ps-cluster1-secrets -p "$(echo -n '{"data":
        {"pmmservertoken":"'$(echo -n new_key | base64)'"}}')"
        ```

2. Update the `pmm` section in the deploy/cr.yaml file:

    - Set `pmm.enabled` = `true`.
    - Specify your PMM Server hostname or an IP address for the `pmm.serverHost` option. The PMM Server IP address should be resolvable and reachable from within your cluster.

    ```
    pmm:
      enabled: true
      image: percona/pmm-client:3.8.1
      serverHost: monitoring-service
    ```

3. Apply the changes:

```
kubectl apply -f deploy/cr.yaml -n <namespace>
```

This triggers the Operator to restart your cluster Pods.

4. Check that corresponding Pods are not in a cycle of stopping and restarting. This cycle occurs if there are errors on the previous steps:

```
kubectl get pods -n <namespace>
kubectl logs  <cluster-name>-mysql-0 -c pmm-client -n <namespace>
```

Check the metrics

Let's see how the collected data is visualized in PMM.

1. Log in to PMM Server.
2. Click MySQL **MySQL** from the left-hand navigation menu.
3. Select your cluster from the **Clusters** drop-down menu and the desired time range on the top of the page. You should see the metrics.
4. Click MySQL **MySQL** → Other dashboards to see the list of available dashboards that allow you to drill down to the metrics you are interested in.
5. Click **Explore** from the left-hand navigation menu. In the **Metric** drop-down, start typing `mysql` to see the list of available metrics.
6. To see the data for the selected metric, click **Run query**.

Next steps

What's next →

What's next?

Congratulations! You've successfully completed the getting started guide. Your MySQL cluster is up and running. Now it's time to prepare it for production use and learn the essential day-to-day operations.

Immediate next steps

These are the most important tasks to tackle right away:

1. Set up automated backups

You've created a single backup, but production workloads need automated backup schedules to protect your data continuously.

- Schedule automatic backups - Configure daily, weekly, or custom backup schedules with retention policies
- Restore from a backup - Learn how to restore your data when needed

2. Create application users

Your cluster is currently using the root user. For production, you'll want to create dedicated users for your applications with appropriate permissions.

- Create application and system users - Set up unprivileged users for your applications with fine-grained access control

3. Enable high availability

Ensure your cluster can survive node failures by distributing pods across different nodes.

- Configure anti-affinity and tolerations - Control pod placement to prevent single points of failure

Production preparation

Once you've covered the basics, prepare your cluster for production workloads:

Security

- Enable TLS/SSL encryption - Secure connections between your applications and the database
- Configure data-at-rest encryption - Protect your data at rest with encryption keys
- Configure encrypted backups - Protect backup files stored in cloud object storage

Configure access to your cluster for external applications

- Configure HAProxy or MySQL Router - Customize load balancing behavior
- Expose your cluster - Configure external access if your applications need it

Performance and reliability

- Scale your cluster - Add more nodes for better performance or increase resources for existing nodes
- Configure MySQL options - Tune MySQL settings for your workload requirements
- Fine-tune backups and restores - Optimize backup performance and customize restore operations

Monitoring and observability

- Monitor with Percona Monitoring and Management (PMM) - Set up comprehensive monitoring, alerts, and performance insights beyond the basic setup

Advanced operations

As you become more comfortable with the Operator, explore these advanced features:

- Add sidecar containers - Extend functionality with custom containers
- Multi-namespace deployment - Configure Operator-wide or namespace-specific deployments

Learn more

- Understand the architecture - Deep dive into how the Operator works and its components
- Review all configuration options - Complete reference for Custom Resource settings

Platform-specific installation

Install Percona Server for MySQL on Google Kubernetes Engine (GKE)

This guide shows you how to deploy Percona Operator for MySQL on Google Kubernetes Engine (GKE). The document assumes some experience with the platform. For more information on the GKE, see the Kubernetes Engine Quickstart.

Prerequisites

All commands from this guide can be run either in the **Google Cloud shell** or in **your local shell**.

To use *Google Cloud shell*, you need nothing but a modern web browser.

If you would like to use *your local shell*, install the following:

1. gcloud. This tool is part of the Google Cloud SDK. To install it, select your operating system on the official Google Cloud SDK documentation page and then follow the instructions.
2. kubectl. It is the Kubernetes command-line tool you will use to manage and deploy applications. To install the tool, run the following command:

```
gcloud auth login
gcloud components install kubectl
```

Before you start

Check the System Requirements to ensure your environment meets the necessary prerequisites.

Create and configure the GKE cluster

You can configure the settings using the `gcloud` tool. You can run it either in the Cloud Shell or in your local shell (if you have installed Google Cloud SDK locally on the previous step). The following command will create a cluster named `ps-cluster1`:

```
gcloud container clusters create ps-cluster1 --project <project ID> --zone
us-central1-a --cluster-version 1.35 --machine-type n1-standard-4 --num-
nodes=3
```

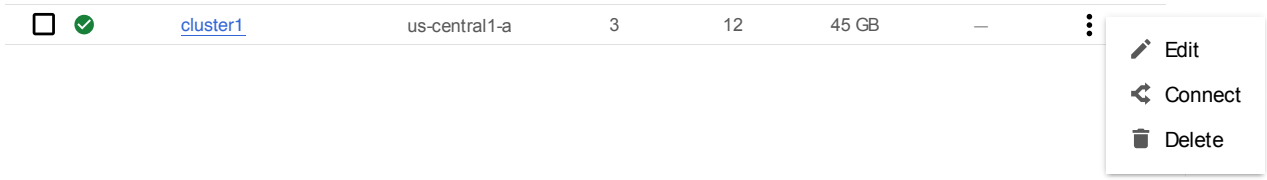
Note

You must edit the above command and other command-line statements to replace the `<project ID>` placeholder with your project ID (see available projects with `gcloud projects list` command). You may also be required to edit the *zone location*, which is set to `us-central1-a` in the above example. Other parameters specify that we are creating a cluster with 3 nodes and with machine type of 4 vCPUs.

You may wait a few minutes for the cluster to be generated.

When the process is over, you can see it listed in the Google Cloud console

Select *Kubernetes Engine* → *Clusters* in the left menu panel:



Now you should configure the command-line access to your newly created cluster to make `kubectl` be able to use it.

In the Google Cloud Console, select your cluster and then click the *Connect* shown on the above image. You will see the connect statement which configures the command-line access. After you have edited the statement, you may run the command in your local shell:

```
gcloud container clusters get-credentials ps-cluster1 --zone us-central1-a --
project <project name>
```

Finally, use your Cloud Identity and Access Management (Cloud IAM) to control access to the cluster. The following command will give you the ability to create Roles and RoleBindings:

```
kubectl create clusterrolebinding cluster-admin-binding --clusterrole
cluster-admin --user $(gcloud config get-value core/account)
```

Expected output

```
clusterrolebinding.rbac.authorization.k8s.io/cluster-admin-binding created
```

Install the Operator and deploy your MySQL cluster

1. Deploy the Operator. By default deployment will be done in the `default` namespace. If that's not the desired one, you can create a new namespace and/or set the context for the namespace as follows (replace the `<namespace name>` placeholder with some descriptive name):

   ```
   kubectl create namespace <namespace name>
   kubectl config set-context $(kubectl config current-context) --namespace=
   <namespace name>
   ```

   At success, you will see the message that `namespace/<namespace name>` was created, and the context (`gke_<project name>_<zone location>_<cluster name>`) was modified.

   Deploy the Operator using the following command:

   ```
   kubectl apply -f https://raw.githubusercontent.com/percona/percona-server
   mysql-operator/v1.2.0/deploy/bundle.yaml
   ```

   Expected output

   ```
   customresourcedefinition.apiextensions.k8s.io/perconaservermysqlbackups.ps.percona.
   com created
   customresourcedefinition.apiextensions.k8s.io/perconaservermysqlrestores.ps.percona
   .com created
   customresourcedefinition.apiextensions.k8s.io/perconaservermysqls.ps.percona.com
   created
   serviceaccount/percona-server-mysql-operator created
   role.rbac.authorization.k8s.io/percona-server-mysql-operator-leaderelection created
   role.rbac.authorization.k8s.io/percona-server-mysql-operator created
   rolebinding.rbac.authorization.k8s.io/percona-server-mysql-operator-leaderelection
   created
   rolebinding.rbac.authorization.k8s.io/percona-server-mysql-operator created
   configmap/percona-server-mysql-operator-config created
   deployment.apps/percona-server-mysql-operator created
   ```

2. The operator has been started, and you can deploy your MySQL cluster:

   ```
   kubectl apply -f https://raw.githubusercontent.com/percona/percona-server
   mysql-operator/v1.2.0/deploy/cr.yaml
   ```

Expected output

```
perconaservermysql.ps.percona.com/ps-cluster1 created
```

Note

This deploys default MySQL cluster configuration. Please see deploy/cr.yaml and Custom Resource Options for the configuration options. You can clone the repository with all manifests and source code by executing the following command:

```
git clone -b v1.2.0 https://github.com/percona/percona-server-mysql-operator
```

After editing the needed options, apply your modified `deploy/cr.yaml` file as follows:

```
kubectl apply -f deploy/cr.yaml
```

The creation process may take some time. When the process is over your cluster will obtain the `ready` status. You can check it with the following command:

```
kubectl get ps
```

Expected output

```
NAME        REPLICATION   ENDPOINT                    STATE   MYSQL   ORCHESTRATOR
HAPROXY   ROUTER   AGE
ps-cluster1   async         ps-cluster1-haproxy.default   ready   3       3
3                 5m50s
```

You can also track the creation process in Google Cloud console via the Object Browser

When the creation process is finished, it will look as follows:

Name	Status	Type	Namespace	Cluster
core		API Group		
Pod		Kind		
cluster1-haproxy-0	Running	Pod	default	cluster1
cluster1-haproxy-1	Running	Pod	default	cluster1
cluster1-haproxy-2	Running	Pod	default	cluster1
cluster1-mysql-0	Running	Pod	default	cluster1
cluster1-mysql-1	Running	Pod	default	cluster1
cluster1-mysql-2	Running	Pod	default	cluster1
cluster1-orc-0	Running	Pod	default	cluster1
cluster1-orc-1	Running	Pod	default	cluster1
cluster1-orc-2	Running	Pod	default	cluster1
percona-server-mysql-operator-7c984f7c9-mgwh4	Running	Pod	default	cluster1

Verifying the cluster operation

It may take ten minutes to get the cluster started. When `kubectl get ps` command finally shows you the cluster status as `ready`, you can try to connect to the cluster.

To connect to Percona Server for MySQL, use the connection Secret that the Operator creates for the `root` user. It is named `<cluster_name>-psuser-root` (by default, `ps-cluster1-psuser-root`) and contains hostnames, ports, credentials, and ready-to-use connection URIs. Read Connection secrets for the full reference.

Here's how to connect:

1. Export the namespace, cluster name and the Secret name as environment variables:

   ```
   export NAMESPACE=my-namespace
   export CLUSTER_NAME=ps-cluster1
   export SECRET_NAME=${CLUSTER_NAME}-psuser-root
   ```

   Replace `ps-cluster1` with the name of your cluster if you changed it during installation.

2. List the Secrets objects.

```
$ kubectl get secrets -n $NAMESPACE
```

Look for the Secret named `<cluster_name>-psuser-root`. By default, it is `ps-cluster1-psuser-root`.

3. Retrieve the user credentials from the Secret:

```
bash
kubectl get secret "$SECRET_NAME" -n "$NAMESPACE" \
  -o jsonpath='{.data.user}' | base64 --decode && echo
kubectl get secret "$SECRET_NAME" -n "$NAMESPACE" \
  -o jsonpath='{.data.password}' | base64 --decode && echo
```

4. Run a container with the `mysql` tool and connect its console output to your terminal:

```
kubectl run -i --rm --tty percona-client --image=percona/percona-server:8.4 --restart=Never -- bash -il
```

It may require some time to execute the command and deploy the corresponding Pod.

5. Now run `mysql` tool in the `percona-client` command shell using the password obtained from the Secret instead of the `<root password>` placeholder. The command will look different depending on whether the cluster uses load balancing with HAProxy (the default behavior) or uses MySQL Router (can be used with Group Replication clusters):

If using HAProxy (default)

```
mysql -h ps-cluster1-haproxy -uroot -p<root password>
```

If using MySQL Router

```
mysql -h ps-cluster1-router -uroot -p<root password>
```

Expected output

```
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1665
Server version: 8.4.10-10.1 Percona Server (GPL), Release 6, Revision dbba4396

Copyright (c) 2009-2026 Percona LLC and/or its affiliates
Copyright (c) 2000, 2026, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

The following example uses the MySQL prompt to check the `max_connections` variable:

```
SHOW VARIABLES LIKE "max_connections";
```

Expected output

```
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_connections | 158   |
+-----------------+-------+
1 row in set (0.02 sec)

mysql>
```

Troubleshooting

If `kubectl get ps` command doesn't show `ready` status too long, you can check the creation process with the `kubectl get pods` command:

```
kubectl get pods
```

Expected output

```
NAME                                               READY   STATUS    RESTARTS       AGE
ps-cluster1-haproxy-0                                 2/2     Running   0              44m
ps-cluster1-haproxy-1                                 2/2     Running   0              44m
ps-cluster1-haproxy-2                                 2/2     Running   0              44m
ps-cluster1-mysql-0                                   3/3     Running   0              46m
ps-cluster1-mysql-1                                   3/3     Running   2 (44m ago)    45m
ps-cluster1-mysql-2                                   3/3     Running   2 (42m ago)    43m
ps-cluster1-orc-0                                     2/2     Running   0              46m
ps-cluster1-orc-1                                     2/2     Running   0              45m
ps-cluster1-orc-2                                     2/2     Running   0              44m
percona-server-mysql-operator-7c984f7c9-mgwh4   1/1     Running   0              47m
```

If the command output had shown some errors, you can examine the problematic Pod with the `kubectl describe <pod name>` command as follows:

```
kubectl describe pod ps-cluster1-mysql-2
```

Review the detailed information for `Warning` statements and then correct the configuration. An example of a warning is as follows:

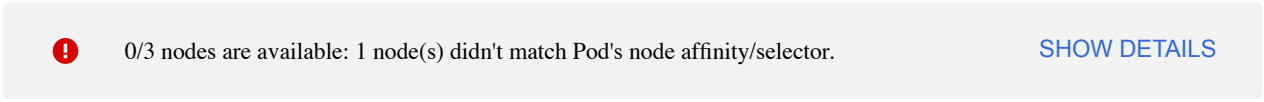
```
Warning  FailedScheduling  68s (x4 over 2m22s)  default-scheduler  0/1 nodes are available: 1 node(s) didn't match pod affinity/anti-affinity, 1 node(s) didn't satisfy existing pods anti-affinity rules.
```

Alternatively, you can examine your Pods via the object browser

The errors will look as follows:

Name	Status	Type	Namespace	Cluster
core		API Group		
Pod		Kind		
cluster1-haproxy-0	Running	Pod	default	cluster1
cluster1-haproxy-1	Running	Pod	default	cluster1
cluster1-haproxy-2	Running	Pod	default	cluster1
cluster1-mysql-0	Running	Pod	default	cluster1
cluster1-mysql-1	Running	Pod	default	cluster1
cluster1-mysql-2	Unschedulable	Pod	default	cluster1
cluster1-orc-0	Running	Pod	default	cluster1
cluster1-orc-1	Running	Pod	default	cluster1
cluster1-orc-2	Running	Pod	default	cluster1
percona-server-mysql-operator-7c984f7c9-mgwh4	Running	Pod	default	cluster1

Clicking the problematic Pod will bring you to the details page with the same warning:



Removing the GKE cluster

There are several ways that you can delete the cluster.

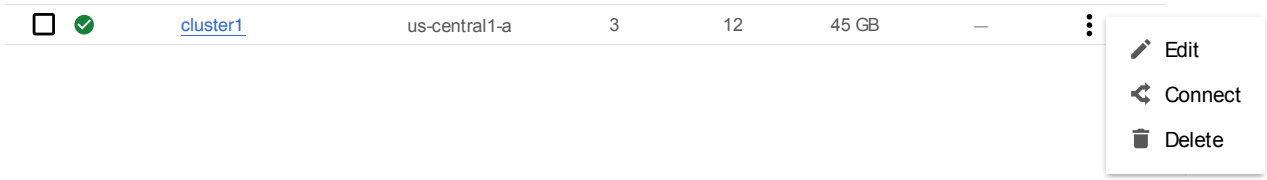
You can clean up the cluster with the `gcloud container clusters delete <cluster name> --zone <zone location>` command. The return statement requests your confirmation of the deletion. Type `y` to confirm.

```
gcloud container clusters delete ps-cluster1 --zone us-central1-a --project
<project ID>
```

The return statement requests your confirmation of the deletion. Type `y` to confirm.

Also, you can delete your cluster via the Google Cloud console

Just click the `Delete` popup menu item in the clusters list:



The cluster deletion may take time.

Warning

After deleting the cluster, all data stored in it will be lost!

Install Percona Distribution for MySQL on Amazon Elastic Kubernetes Service (EKS)

This guide shows you how to deploy Percona Operator for MySQL on Amazon Elastic Kubernetes Service (EKS). The document assumes some experience with Amazon EKS. For more information on the EKS, see the Amazon EKS official documentation.

Prerequisites

The following tools are used in this guide and therefore should be preinstalled:

1. **AWS Command Line Interface (AWS CLI)** for interacting with the different parts of AWS. You can install it following the official installation instructions for your system.
2. **eksctl** to simplify cluster creation on EKS. It can be installed along its installation notes on GitHub.
3. **kubectl** to manage and deploy applications on Kubernetes. Install it following the official installation instructions.

Also, you need to configure AWS CLI with your credentials according to the official guide.

Before you start

Check the System Requirements to ensure your environment meets the necessary prerequisites.

Create the EKS cluster

1. To create your cluster, you will need the following data:
   - name of your EKS cluster,
   - AWS region in which you wish to deploy your cluster,
   - the amount of nodes you would like to have,
   - the desired ratio between on-demand and spot instances in the total number of nodes.

> **Note**
>
> spot instances are not recommended for production environment, but may be useful e.g. for testing purposes.

After you have settled all the needed details, create your EKS cluster following the official cluster creation instructions.

2. After you have created the EKS cluster, you also need to install the Amazon EBS CSI driver on your cluster. See the official documentation on adding it as an Amazon EKS add-on.

Install the Operator and deploy your MySQL cluster

1. Create a namespace and set the context for the namespace. The resource names must be unique within the namespace and provide a way to divide cluster resources between users spread across multiple projects.

   So, create the namespace and save it in the namespace context for subsequent commands as follows (replace the `<namespace name>` placeholder with some descriptive name):

   ```
   kubectl create namespace <namespace name>
   kubectl config set-context $(kubectl config current-context) --namespace=
   <namespace name>
   ```

   At success, you will see the message that namespace/ was created, and the context was modified.

2. Use the following `git clone` command to download the correct branch of the percona-server-mysql-operator repository:

   ```
   git clone -b v1.2.0 https://github.com/percona/percona-server-mysql-
   operator
   ```

   After the repository is downloaded, change the directory to run the rest of the commands in this document:

   ```
   cd percona-server-mysql-operator
   ```

3. Deploy the Operator using the following command:

```
kubectl apply --server-side -f deploy/bundle.yaml
```

The following confirmation is returned:

```
customresourcedefinition.apiextensions.k8s.io/perconaserverformysqlbackups
.ps.percona.com created
customresourcedefinition.apiextensions.k8s.io/perconaserverformysqlrestore
s.ps.percona.com created
customresourcedefinition.apiextensions.k8s.io/perconaserverformysqls.ps.pe
rcona.com created
serviceaccount/percona-server-for-mysql-operator created
role.rbac.authorization.k8s.io/percona-server-for-mysql-operator-leader-
election-role created
role.rbac.authorization.k8s.io/percona-server-for-mysql-operator-role
created
rolebinding.rbac.authorization.k8s.io/percona-server-for-mysql-operator-
leader-election-rolebinding created
rolebinding.rbac.authorization.k8s.io/percona-server-for-mysql-operator-
rolebinding created
configmap/percona-server-for-mysql-operator-config created
deployment.apps/percona-server-for-mysql-operator created
```

4. The operator has been started, and you can create the Percona Distribution for MySQL cluster:

Warning

Starting with 1.30, Amazon EKS no longer automatically applies the default annotation for the `gp2` StorageClass to newly created clusters.

You need to specify the storageClassName explicitly in `deploy/cr.yaml`:

```
mysql:
  ...
  volumeSpec:
    persistentVolumeClaim:
      storageClassName: gp2
      resources:
        requests:
          storage: 20Gi
```

```
kubectl apply -f deploy/cr.yaml
```

The process could take some time. The return statement confirms the creation:

```
perconaserverformysql.ps.percona.com/ps-cluster1 created
```

Verify the cluster operation

To connect to Percona Server for MySQL, use the connection Secret that the Operator creates for the `root` user. It is named `<cluster_name>-psuser-root` (by default, `ps-cluster1-psuser-root`) and contains hostnames, ports, credentials, and ready-to-use connection URIs. Read Connection secrets for the full reference.

Here's how to connect:

1. Export the namespace, cluster name and the Secret name as environment variables:

   ```
   export NAMESPACE=my-namespace
   export CLUSTER_NAME=ps-cluster1
   export SECRET_NAME=${CLUSTER_NAME}-psuser-root
   ```

   Replace `ps-cluster1` with the name of your cluster if you changed it during installation.

2. List the Secrets objects.

   ```
   $ kubectl get secrets -n $NAMESPACE
   ```

   Look for the Secret named `<cluster_name>-psuser-root`. By default, it is `ps-cluster1-psuser-root`.

3. Retrieve the user credentials from the Secret:

   `bash`
   `kubectl get secret "$SECRET_NAME" -n "$NAMESPACE" \`
   `  -o jsonpath='{.data.user}' | base64 --decode && echo`
   `kubectl get secret "$SECRET_NAME" -n "$NAMESPACE" \`
   `  -o jsonpath='{.data.password}' | base64 --decode && echo`

4. Run a container with the `mysql` tool and connect its console output to your terminal:

   ```
   kubectl run -i --rm --tty percona-client --image=percona/percona-server:8.4 --restart=Never -- bash -il
   ```

   It may require some time to execute the command and deploy the corresponding Pod.

5. Now run `mysql` tool in the `percona-client` command shell using the password obtained from the Secret instead of the `<root password>` placeholder. The command will look different depending on whether the cluster uses load balancing with HAProxy (the default behavior) or uses MySQL Router (can be used with Group Replication clusters):

   If using HAProxy (default)

   ```
   mysql -h ps-cluster1-haproxy -uroot -p<root password>
   ```

   If using MySQL Router

   ```
   mysql -h ps-cluster1-router -uroot -p<root password>
   ```

   Expected output

   ```
   mysql: [Warning] Using a password on the command line interface can be insecure.
   Welcome to the MySQL monitor.  Commands end with ; or \g.
   Your MySQL connection id is 1665
   Server version: 8.4.10-10.1 Percona Server (GPL), Release 6, Revision dbba4396

   Copyright (c) 2009-2026 Percona LLC and/or its affiliates
   Copyright (c) 2000, 2026, Oracle and/or its affiliates.

   Oracle is a registered trademark of Oracle Corporation and/or its
   affiliates. Other names may be trademarks of their respective
   owners.

   Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

   mysql>
   ```

   The following example uses the MySQL prompt to check the `max_connections` variable:

   ```
   SHOW VARIABLES LIKE "max_connections";
   ```

Expected output

```
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_connections | 158   |
+-----------------+-------+
1 row in set (0.02 sec)

mysql>
```

6. You can also check whether you can connect to MySQL from the outside with the help of the `kubectl port-forward` command as follows:

```
kubectl port-forward svc/ps-cluster1-mysql-primary 3306:3306 &
mysql -h 127.0.0.1 -P 3306 -uroot -p<root password>
```

Install Percona Server for MySQL on OpenShift

You can install Percona Operator for MySQL on OpenShift clusters. This makes it portable across hybrid clouds and it fully supports the Red Hat OpenShift lifecycle.

To install Percona Server for MySQL on OpenShift means:

- Install Percona Operator for MySQL,
- Install Percona Server for MySQL using the Operator.

Prerequisites

- OpenShift cluster with administrative access
- `oc` command-line tool installed
- Git client installed

Before you start

Check the System Requirements to ensure your environment meets the necessary prerequisites.

You can install Percona Operator for MySQL on OpenShift using either:

- The Operator Lifecycle Manager web interface
- The command-line interface

Choose the method that best suits your needs. The web interface is recommended for beginners, while the CLI method offers more control and automation capabilities.

Install the Operator via the Operator Lifecycle Manager (OLM)

Operator Lifecycle Manager (OLM) is a part of the Operator Framework that allows you to install, update, and manage the Operators lifecycle on the OpenShift platform via the web interface.

This tutorial provides guidelines for OpenShift v4.20. Follow closely the requirements for your OpenShift version.

Prerequisites

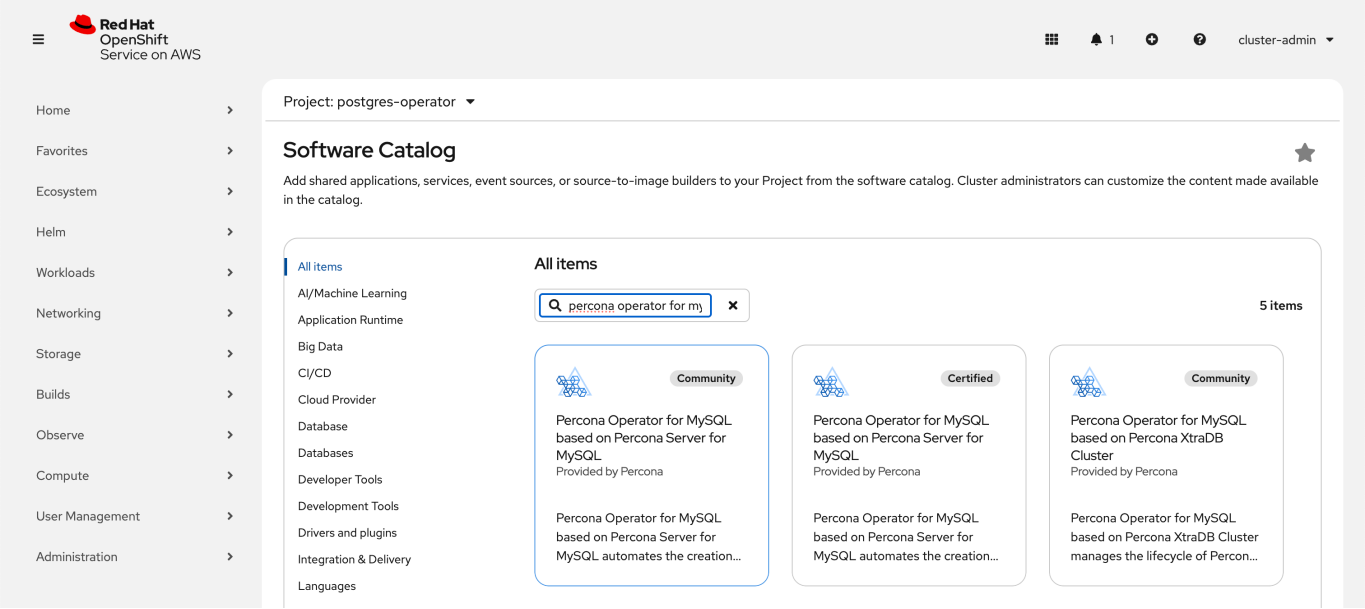
Before you start, ensure you have the following:

1. You can log in to the OpenShift console
2. You have the ARN role assigned to your OLM user (for OpenShift 4.20).

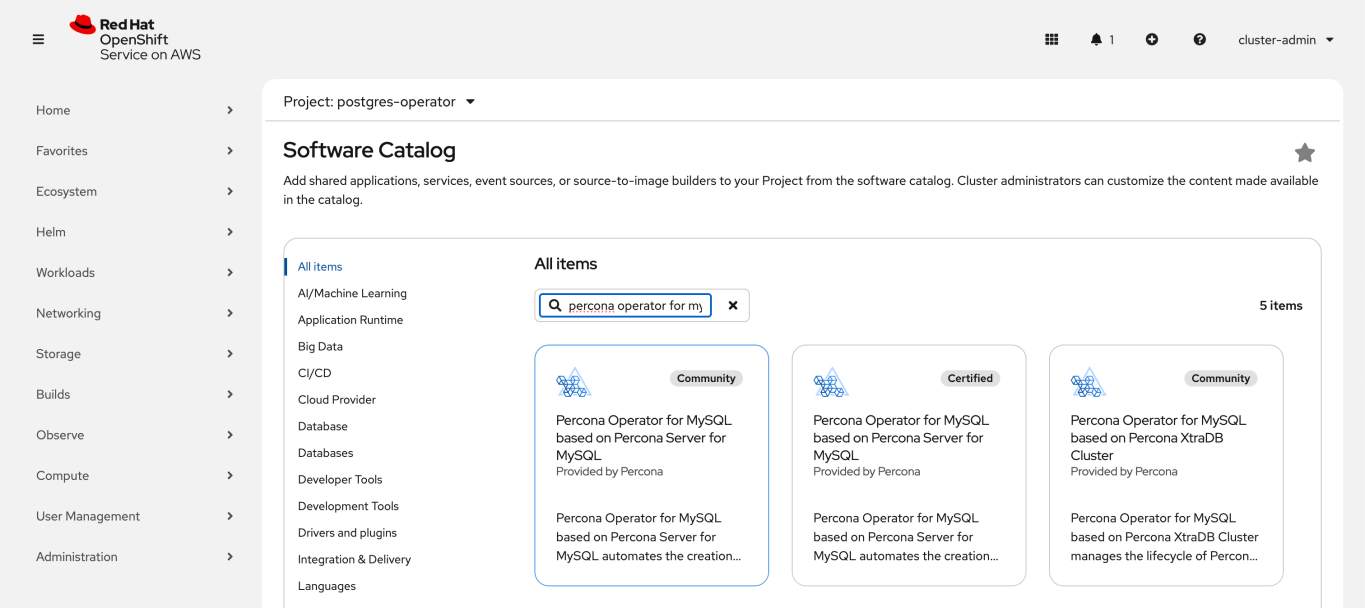
Install the Operator Deployment

Follow these steps to deploy the Operator and Percona Server for MySQL cluster:

1. Login to the OpenShift console.
2. Navigate to the Ecosystem -> Software Catalog.
3. Search for “Percona Operator for MySQL”, select “Percona Operator for MySQL based on Percona Server for MySQL”. You may need to change the project for your user:



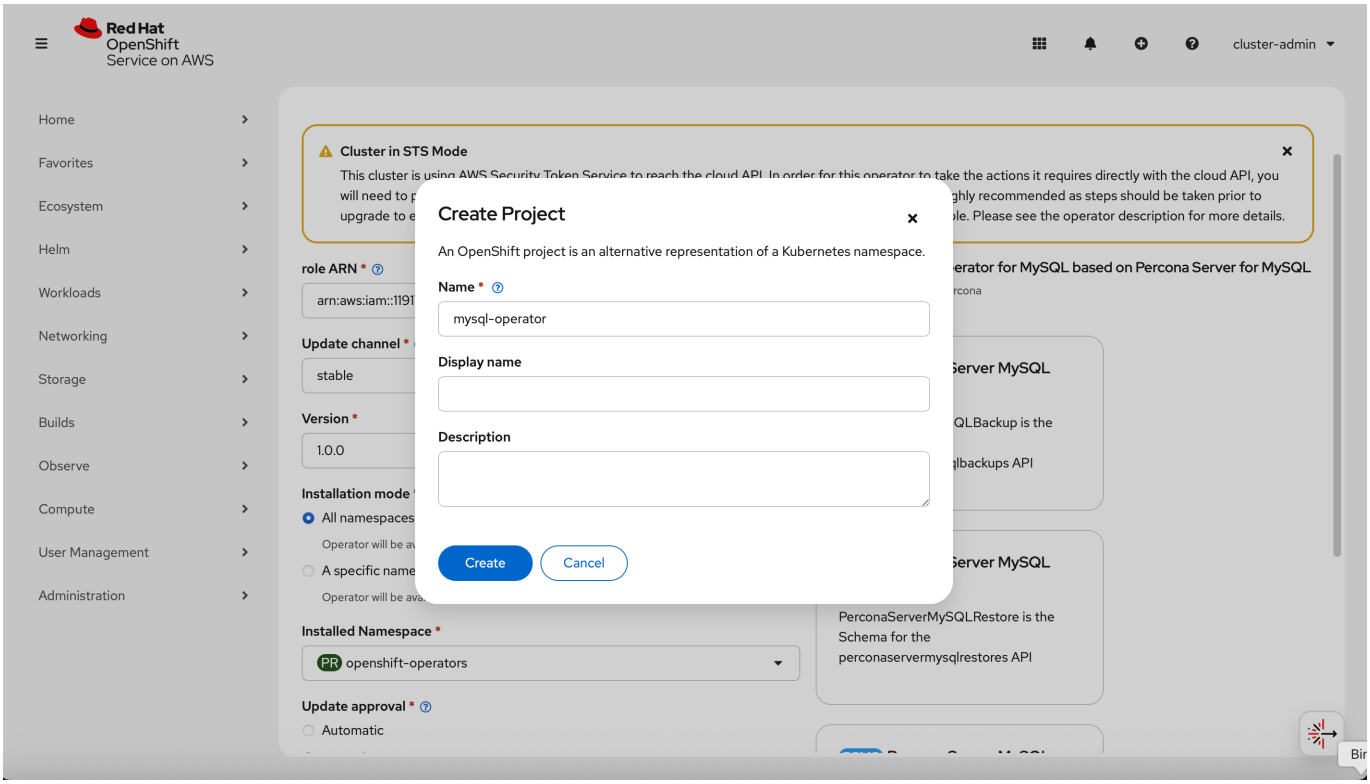
4. Then click “Continue”, and “Install”.



5. A new page opens where you choose the Operator version and the Namespace / OpenShift project you would like to install the Operator into. You can create a namespace (an OpenShift project) right away by clicking the **Create Project** and filling in project details like name, display name and description.

   For OpenShift 4.20, you also need to specify the ARN role assigned to your user.

6. Click “Install”

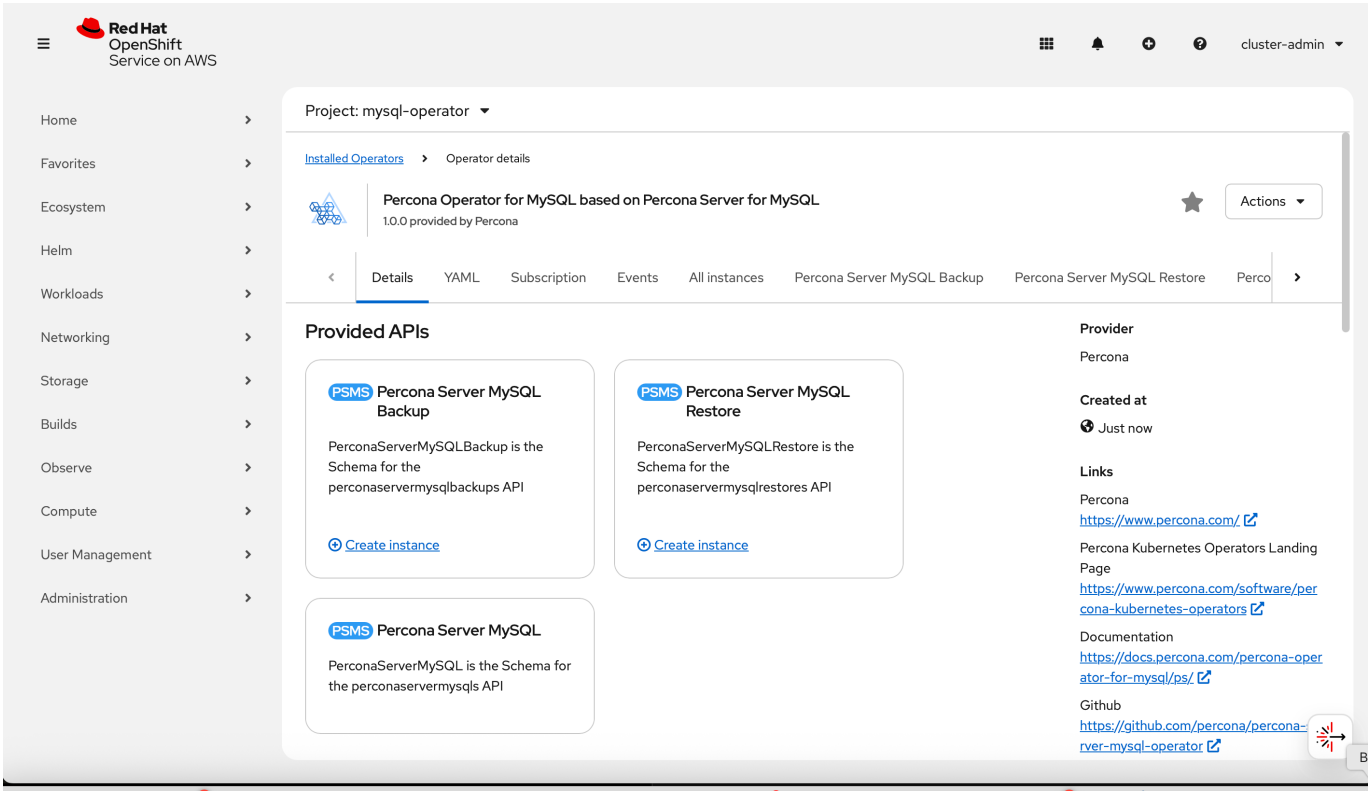


You can track the install process on the Installed Operators page. The Operator should report the **Succeeded** status.

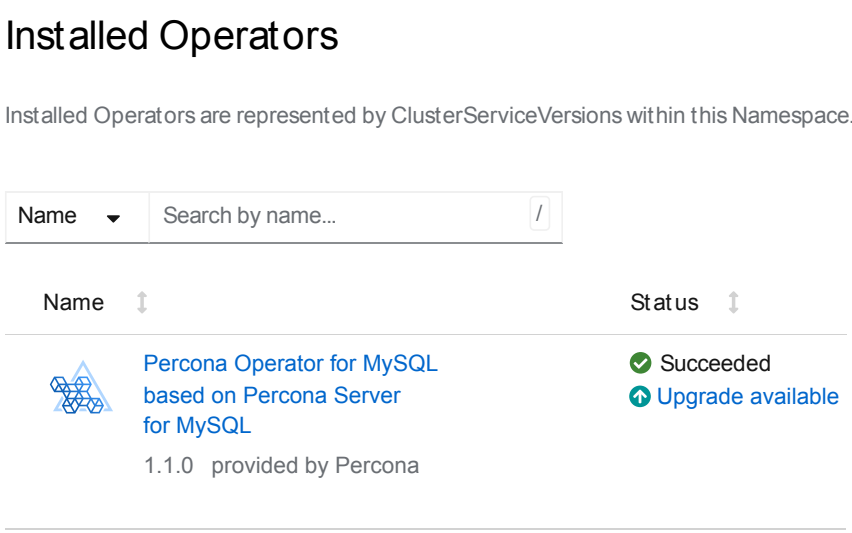
Deploy Percona Server for MySQL

Now you can deploy Percona Server for MySQL

1. Click the Operator you installed.
2. On the Details page, find the `PerconaServerMySQL` Custom Resource
3. Click “Create instance”
4. Edit the Custom Resource manifest to fine-tune your cluster configuration. Refer to Custom Resource reference for the description of available options
5. Click “Create”



6. Upon successful installation, you should see the “Ready” status for the database cluster.



Install the Operator via the command-line interface

The following steps install the latest version of the Operator with default parameters. To install a specific version, replace the `v1.2.0` tag with your value. See the full list of tags in the Operator repository on GitHub.

To install the Operator with customized parameters, see Install Percona Operator for MySQL with customized parameters.

Choose the approach that fits your needs:

- **Quick install** — Apply a single bundle file. Use this when you want to get started quickly with default settings.
- **Step-by-step install** — Run each installation step separately. Use this when you want more control over the installation process or you need to customize the installation.

Quick install

1. Clone the `percona-server-mysql-operator` repository and change the directory to `percona-server-mysql-operator`.

   Important

   You must specify the correct branch with the `-b` option while cloning the code on this step. Please be careful.

```
git clone -b v1.2.0 https://github.com/percona/percona-server-mysql-operator
cd percona-server-mysql-operator
```

2. Create the Kubernetes namespace for your cluster. It is a good practice to isolate workloads in Kubernetes by installing the Operator in a custom namespace. Replace the `<namespace>` placeholder with your value.

```
oc create namespace <namespace>
```

Expected output

```
namespace/<namespace> was created
```

3. A `bundle.yaml` is a Kubernetes manifest that packages Operator metadata and resources. By applying this file, Kubernetes creates the Custom Resource Definition, sets up role-based access control and installs the Operator in one single action. Replace the `<namespace>` placeholder with your value:

```
oc apply --server-side -f deploy/bundle.yaml -n <namespace>
```

Expected output

```
customresourcedefinition.apiextensions.k8s.io/perconaservermysqlbackups.ps.percona.
com serverside-applied
customresourcedefinition.apiextensions.k8s.io/perconaservermysqlrestores.ps.percona
.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/perconaservermysqls.ps.percona.com
serverside-applied
serviceaccount/percona-server-mysql-operator serverside-applied
role.rbac.authorization.k8s.io/percona-server-mysql-operator-leaderelection
serverside-applied
role.rbac.authorization.k8s.io/percona-server-mysql-operator serverside-applied
rolebinding.rbac.authorization.k8s.io/percona-server-mysql-operator serverside-
applied
rolebinding.rbac.authorization.k8s.io/percona-server-mysql-operator-leaderelection
serverside-applied
configmap/percona-server-mysql-operator-config serverside-applied
deployment.apps/percona-server-mysql-operator serverside-applied
```

Step-by-step installation

This section splits the installation flow into separate steps giving you more control over the process.

Step 1: Clone the repository

Use the following commands to clone the `percona-server-mysql-operator` repository and change the directory to `percona-server-mysql-operator`.

> **Important**
>
> You must specify the correct branch with the `-b` option while cloning the code on this step. Please be careful.

```
git clone -b v1.2.0 https://github.com/percona/percona-server-mysql-operator
cd percona-server-mysql-operator
```

Step 2: Create the Custom Resource Definition

At this step you must create the Custom Resource Definition for Percona Operator for MySQL from the `deploy/crd.yaml` file.

The Custom Resource Definition extends the standard set of resources which Kubernetes “knows” about with new items.

You create the Custom Resource Definition only once. It is not bound to a specific namespace and all other deployments will use this Custom Resource Definition.

Use the following command to create the Custom Resource Definition:

```
oc apply --server-side -f deploy/crd.yaml
```

> **Warning**
>
> This step requires cluster-admin privileges. If you’re using a non-privileged user, you’ll need to set up additional permissions.

Step 3: (optional) Set up user permissions

If you’re using a non-privileged user, grant the required permissions by applying the following clusterrole:

```
oc create clusterrole ps-admin --verb="*" --
resource=perconaservermysqls.ps.percona.com,perconaservermysqls.ps.percona.co
m/status,perconaservermysqlbackups.ps.percona.com,perconaservermysqlbackups.p
s.percona.com/status,perconaservermysqlrestores.ps.percona.com,perconaserverm
ysqlrestores.ps.percona.com/status
oc adm policy add-cluster-role-to-user ps-admin <some-user>
```

If you have a cert-manager installed, add these permissions to manage certificates with a non-privileged user:

```
oc create clusterrole cert-admin --verb="*" --
resource=issuers.certmanager.k8s.io,certificates.certmanager.k8s.io
oc adm policy add-cluster-role-to-user cert-admin <some-user>
```

Step 4: Create a project

A project in OpenShift corresponds to a Kubernetes namespace. When you create a new project, you isolate workloads in it.

```
oc new-project ps
```

Sample output

Now using project “ps” on server “https://api.openshift-4-15-my-cluster.example.com:6443”.

The command automatically sets context to this project so that all further resources are created in it.

Step 5: Configure RBAC

Role-Based Access Control (RBAC) manages resource access in OpenShift. The Operator needs specific permissions to run Percona Server for MySQL properly. These permissions are defined within roles.

```
oc apply -f deploy/rbac.yaml
```

Step 6: Deploy the Operator

Now you can deploy the Operator with the following command:

```
oc apply -f deploy/operator.yaml
```

Install Percona Server for MySQL

After installing the Operator, you can deploy Percona Server for MySQL. This section guides you through the process of setting up secrets, certificates, and creating your first cluster.

Step 1: Configure secrets (optional)

By default, the Operator generates users Secrets automatically, so you don't have to do anything. Yet if you wish to use your own Secrets, here's how:

1. Edit the `deploy/secrets.yaml` file to set up your MySQL users and passwords:

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: my-cluster-secrets
   type: Opaque
   stringData:
     root: your-root-password
     xtrabackup: your-xtrabackup-password
     monitor: your-monitor-password
     clustercheck: your-clustercheck-password
     proxyadmin: your-proxyadmin-password
     pmmserver: your-pmm-server-password
   ```

2. Apply the secrets:

   ```
   oc create -f deploy/secrets.yaml
   ```

Step 2: Configure certificates (optional)

The Operator handles certificate generation automatically so don't have to do anything. However, if you need custom certificates:

1. Generate your certificates
2. Create a secret with your certificates
3. Reference the secret in your cluster configuration

See TLS Configuration for detailed instructions.

Step 3: Deploy the database cluster

1. To deploy Percona Server for MySQL cluster means to create a Custom Resource for it in OpenShift. This Custom Resource uses the Percona Server for MySQL Operator, which automates the deployment, scaling, and management of MySQL clusters.

   The Custom Resource is described by the `deploy/cr.yaml` file. So to create it, you need to apply this file as follows:

   ```
   oc apply -f deploy/cr.yaml
   ```

   Expected output

   ```
   perconaservermysql.ps.percona.com/ps-cluster1 created
   ```

2. It may take up to 10 minutes to complete the cluster deployment. Use this command to monitor the deployment:

   ```
   oc get ps
   ```

   Expected output

   ```
   NAME       REPLICATION          ENDPOINT                     STATE    MYSQL
   ORCHESTRATOR   HAPROXY   ROUTER   AGE
   ps-cluster1   group-replication   ps-cluster1-haproxy.nastena1   ready   3
   3                  6m
   ```

   The `ready` status indicates that your cluster is fully operational.

Verify the cluster operation

To connect to Percona Server for MySQL, use the connection Secret that the Operator creates for the `root` user. It is named `<cluster_name>-psuser-root` (by default, `ps-cluster1-psuser-root`) and contains hostnames, ports, credentials, and ready-to-use connection URIs. Read Connection secrets for the full reference.

Here's how to connect:

1. Export the namespace, cluster name and the Secret name as environment variables:

```
export NAMESPACE=my-namespace
export CLUSTER_NAME=ps-cluster1
export SECRET_NAME=${CLUSTER_NAME}-psuser-root
```

Replace `ps-cluster1` with the name of your cluster if you changed it during installation.

2. List the Secrets objects.

```
$ oc get secrets -n $NAMESPACE
```

Look for the Secret named `<cluster_name>-psuser-root`. By default, it is `ps-cluster1-psuser-root`.

3. Retrieve the user credentials from the Secret:

`bash`

```
oc get secret "$SECRET_NAME" -n "$NAMESPACE" \
  -o jsonpath='{.data.user}' | base64 --decode && echo
oc get secret "$SECRET_NAME" -n "$NAMESPACE" \
  -o jsonpath='{.data.password}' | base64 --decode && echo
```

4. Run a container with the `mysql` tool and connect its console output to your terminal:

```
oc run -i --rm --tty percona-client --image=percona/percona-server:8.4 --
restart=Never -- bash -il
```

It may require some time to execute the command and deploy the corresponding Pod.

5. Now run `mysql` tool in the `percona-client` command shell using the password obtained from the Secret instead of the `<root password>` placeholder. The command will look different depending on whether the cluster uses load balancing with HAProxy (the default behavior) or uses MySQL Router (can be used with Group Replication clusters):

If using HAProxy (default)

```
mysql -h ps-cluster1-haproxy -uroot -p<root password>
```

If using MySQL Router

```
mysql -h ps-cluster1-router -uroot -p<root password>
```

Expected output

```
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1665
Server version: 8.4.10-10.1 Percona Server (GPL), Release 6, Revision dbba4396

Copyright (c) 2009-2026 Percona LLC and/or its affiliates
Copyright (c) 2000, 2026, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

The following example uses the MySQL prompt to check the `max_connections` variable:

```
SHOW VARIABLES LIKE "max_connections";
```

Expected output

```
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_connections | 158   |
+-----------------+-------+
1 row in set (0.02 sec)

mysql>
```

Next steps

Configure Backup and Restore | Set up monitoring | Scale your cluster

Install Percona Server for MySQL on Minikube

Minikube lets you run a Kubernetes cluster locally without a cloud provider. It works on Linux, Windows, and macOS using a hypervisor like VirtualBox, KVM/QEMU, VMware Fusion, Hyper-V, or Docker. This makes it perfect for testing the Operator before deploying it in a cloud or in production

Prerequisites

Before you begin, you need to install Minikube on your system. The installation includes three components:

1. **kubectl** - the Kubernetes command-line tool
2. **A hypervisor** - if you don't already have one installed
3. **Minikube** - the Minikube package itself

After installing Minikube, start it with increased resources to ensure the Operator runs smoothly:

```
minikube start --memory=4096 --cpus=3
```

This command downloads the necessary virtualized images, then initializes and starts your local Kubernetes cluster. The `--memory=4096` and `--cpus=3` parameters allocate more resources to the virtual machine, which helps the Operator run reliably.

Optional: Kubernetes Dashboard

You can optionally start the Kubernetes dashboard to visualize your cluster. Run:

```
minikube dashboard
```

This opens the dashboard in your default web browser, giving you a visual view of your cluster's state.

Install the Operator

1. Clone the repository and navigate into it:

```
git clone -b v1.2.0 https://github.com/percona/percona-server-mysql-operator
cd percona-server-mysql-operator
```

2. Deploy the Operator to your Minikube cluster

```
kubectl apply--server-side -f deploy/bundle.yaml
```

Expected output

```
customresourcedefinition.apiextensions.k8s.io/perconaservermysqlbackups.ps.percona.com created
customresourcedefinition.apiextensions.k8s.io/perconaservermysqlrestores.ps.percona.com created
customresourcedefinition.apiextensions.k8s.io/perconaservermysqls.ps.percona.com created
serviceaccount/percona-server-mysql-operator created
role.rbac.authorization.k8s.io/percona-server-mysql-operator-leaderelection created
role.rbac.authorization.k8s.io/percona-server-mysql-operator created
rolebinding.rbac.authorization.k8s.io/percona-server-mysql-operator-leaderelection created
rolebinding.rbac.authorization.k8s.io/percona-server-mysql-operator created
configmap/percona-server-mysql-operator-config created
deployment.apps/percona-server-mysql-operator created
```

Configure and deploy your MySQL cluster

Since Minikube runs on a single node, the default configuration doesn't fit. Use the minimal configuration adjusted for Minikube environment.

```
kubectl apply -f deploy/cr-minimal.yaml
```

Expected output

```
perconaservermysql.ps.percona.com/ps-cluster1 created
```

This creates a group-replication cluster with one Percona Server for MySQL instances and one HAProxy instance. For more configuration options, see the `deploy/cr.yaml` file and the Custom Resource Options reference.

Check the cluster status

The cluster creation process takes a few minutes. Monitor the status with:

```
kubectl get ps
```

Wait until the `STATE` column shows `ready`. This indicates your cluster is fully operational.

Expected output

```
NAME           REPLICATION         ENDPOINT                    STATE    MYSQL
ORCHESTRATOR   HAPROXY    ROUTER   AGE
ps-cluster1    group-replication   ps-cluster1-haproxy.default   ready    1
1                  5m50s
```

Verify the cluster operation

It typically takes about ten minutes for the cluster to start. Once `kubectl get ps` shows the cluster status as `ready`, you can connect to it and start using your MySQL database.

To connect to Percona Server for MySQL, use the connection Secret that the Operator creates for the `root` user. It is named `<cluster_name>-psuser-root` (by default, `ps-cluster1-psuser-root`) and contains hostnames, ports, credentials, and ready-to-use connection URIs. Read Connection secrets for the full reference.

Here's how to connect:

1. Export the namespace, cluster name and the Secret name as environment variables:

   ```
   export NAMESPACE=my-namespace
   export CLUSTER_NAME=ps-cluster1
   export SECRET_NAME=${CLUSTER_NAME}-psuser-root
   ```

   Replace `ps-cluster1` with the name of your cluster if you changed it during installation.

2. List the Secrets objects.

   ```
   $ kubectl get secrets -n $NAMESPACE
   ```

Look for the Secret named `<cluster_name>-psuser-root`. By default, it is `ps-cluster1-psuser-root`.

3. Retrieve the user credentials from the Secret:

   `bash`

   ```
   kubectl get secret "$SECRET_NAME" -n "$NAMESPACE" \
     -o jsonpath='{.data.user}' | base64 --decode && echo
   kubectl get secret "$SECRET_NAME" -n "$NAMESPACE" \
     -o jsonpath='{.data.password}' | base64 --decode && echo
   ```

4. Run a container with the `mysql` tool and connect its console output to your terminal:

   ```
   kubectl run -i --rm --tty percona-client --image=percona/percona-server:8.4 --restart=Never -- bash -il
   ```

   It may require some time to execute the command and deploy the corresponding Pod.

5. Now run `mysql` tool in the `percona-client` command shell using the password obtained from the Secret instead of the `<root password>` placeholder. The command will look different depending on whether the cluster uses load balancing with HAProxy (the default behavior) or uses MySQL Router (can be used with Group Replication clusters):

   If using HAProxy (default)

   ```
   mysql -h ps-cluster1-haproxy -uroot -p<root password>
   ```

   If using MySQL Router

   ```
   mysql -h ps-cluster1-router -uroot -p<root password>
   ```

Expected output

```
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1665
Server version: 8.4.10-10.1 Percona Server (GPL), Release 6, Revision dbba4396

Copyright (c) 2009-2026 Percona LLC and/or its affiliates
Copyright (c) 2000, 2026, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

The following example uses the MySQL prompt to check the `max_connections` variable:

```
SHOW VARIABLES LIKE "max_connections";
```

Expected output

```
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_connections | 158   |
+-----------------+-------+
1 row in set (0.02 sec)

mysql>
```

Install Percona Server for MySQL on Kubernetes

1. First of all, clone the percona-server-mysql-operator repository:

   ```
   git clone -b v1.2.0 https://github.com/percona/percona-server-mysql-operator
   cd percona-server-mysql-operator
   ```

   Note

   It is crucial to specify the right branch with `-b` option while cloning the code on this step. Please be careful.

2. Now Custom Resource Definition for Percona Server for MySQL should be created from the `deploy/crd.yaml` file. Custom Resource Definition extends the standard set of resources which Kubernetes "knows" about with the new items (in our case ones which are the core of the operator). Apply it as follows:

   ```
   kubectl apply --server-side -f deploy/crd.yaml
   ```

   This step should be done only once; it does not need to be repeated with the next Operator deployments, etc.

3. The next thing to do is to add the `mysql` namespace to Kubernetes, not forgetting to set the correspondent context for further steps:

   ```
   kubectl create namespace mysql
   kubectl config set-context $(kubectl config current-context) --namespace=mysql
   ```

   Note

   You can use different namespace name or even stay with the *Default* one.

4. Now RBAC (role-based access control) for Percona Server for MySQL should be set up from the `deploy/rbac.yaml` file. Briefly speaking, role-based access is based on specifically defined roles and actions corresponding to them, allowed to be done on specific Kubernetes resources (details about users and roles can be found in Kubernetes documentation).

```
kubectl apply -f deploy/rbac.yaml
```

> **Note**
>
> Setting RBAC requires your user to have cluster-admin role privileges. For example, those using Google Kubernetes Engine can grant user needed privileges with the following command: `$ kubectl create clusterrolebinding cluster-admin-binding --clusterrole=cluster-admin --user=$(gcloud config get-value core/account)`

Finally it's time to start the operator within Kubernetes:

```
kubectl apply -f deploy/operator.yaml
```

5. Now that's time to add the Percona Server for MySQL Users secrets to Kubernetes. They should be placed in the data section of the `deploy/secrets.yaml` file as logins and plaintext passwords for the user accounts (see Kubernetes documentation for details).

   After editing is finished, users secrets should be created using the following command:

   ```
   kubectl create -f deploy/secrets.yaml
   ```

   More details about secrets can be found in Users.

6. Now certificates should be generated. By default, the Operator generates certificates automatically, and no actions are required at this step. Still, you can generate and apply your own certificates as secrets according to the TLS instructions.

7. After the operator is started and user secrets are added, Percona Server for MySQL can be created at any time with the following command:

   ```
   kubectl apply -f deploy/cr.yaml
   ```

   Creation process will take some time. The process is over when both operator and replica set pod have reached their Running status. `kubectl get pods` output should look like this:

```
NAME                                                  READY   STATUS
RESTARTS        AGE
ps-cluster1-mysql-0                                     1/1     Running
0               7m6s
ps-cluster1-mysql-1                                     1/1     Running
1 (5m39s ago)   6m4s
ps-cluster1-mysql-2                                     1/1     Running
1 (4m40s ago)   5m7s
ps-cluster1-orc-0                                       2/2     Running
0               7m6s
percona-server-for-mysql-operator-54c5c87988-xfmlf   1/1     Running   0
7m42s
```

Verify the cluster operation

To connect to Percona Server for MySQL, use the connection Secret that the Operator creates for the `root` user. It is named `<cluster_name>-psuser-root` (by default, `ps-cluster1-psuser-root`) and contains hostnames, ports, credentials, and ready-to-use connection URIs. Read Connection secrets for the full reference.

Here's how to connect:

1. Export the namespace, cluster name and the Secret name as environment variables:

   ```
   export NAMESPACE=my-namespace
   export CLUSTER_NAME=ps-cluster1
   export SECRET_NAME=${CLUSTER_NAME}-psuser-root
   ```

   Replace `ps-cluster1` with the name of your cluster if you changed it during installation.

2. List the Secrets objects.

   ```
   $ kubectl get secrets -n $NAMESPACE
   ```

   Look for the Secret named `<cluster_name>-psuser-root`. By default, it is `ps-cluster1-psuser-root`.

3. Retrieve the user credentials from the Secret:

`bash`

```
kubectl get secret "$SECRET_NAME" -n "$NAMESPACE" \
  -o jsonpath='{.data.user}' | base64 --decode && echo
kubectl get secret "$SECRET_NAME" -n "$NAMESPACE" \
  -o jsonpath='{.data.password}' | base64 --decode && echo
```

4. Run a container with the `mysql` tool and connect its console output to your terminal:

```
kubectl run -i --rm --tty percona-client --image=percona/percona-server:8.4 --restart=Never -- bash -il
```

It may require some time to execute the command and deploy the corresponding Pod.

5. Now run `mysql` tool in the `percona-client` command shell using the password obtained from the Secret instead of the `<root password>` placeholder. The command will look different depending on whether the cluster uses load balancing with HAProxy (the default behavior) or uses MySQL Router (can be used with Group Replication clusters):

If using HAProxy (default)

```
mysql -h ps-cluster1-haproxy -uroot -p<root password>
```

If using MySQL Router

```
mysql -h ps-cluster1-router -uroot -p<root password>
```

Expected output

```
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1665
Server version: 8.4.10-10.1 Percona Server (GPL), Release 6, Revision dbba4396

Copyright (c) 2009-2026 Percona LLC and/or its affiliates
Copyright (c) 2000, 2026, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

The following example uses the MySQL prompt to check the `max_connections` variable:

```
SHOW VARIABLES LIKE "max_connections";
```

Expected output

```
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_connections | 158   |
+-----------------+-------+
1 row in set (0.02 sec)

mysql>
```

Daily operations

Users

MySQL user accounts within the Cluster can be divided into two different groups:

- *application-level users*: the unprivileged user accounts,
- *system-level users*: the accounts needed to automate the cluster deployment and management tasks, such as Percona Server for MySQL Health checks.

As these two groups of user accounts serve different purposes, they are considered separately in the following sections.

Unprivileged users

There are no unprivileged (general purpose) user accounts created by default. If you need general purpose users, run the following commands:

Start a temporary Percona MySQL client Pod and connect to MySQL in your cluster

```
kubectl run -it --rm percona-client --image=percona:8.4 --restart=Never --
mysql -hps-cluster1-mysql -uroot -proot_password
```

The following SQL command creates a new user `user1` with the password `password1`, and grants this user all privileges on all tables in the `database1` database. The `@'%'` means that the user can connect from any host.

```
CREATE USER 'user1'@'%' IDENTIFIED BY 'password1';
GRANT ALL PRIVILEGES ON database1.* TO 'user1'@'%';
FLUSH PRIVILEGES;
```

Note

MySQL password here should not exceed 32 characters due to the replication-specific limit introduced in MySQL 5.7.5.

Verify that the user was created successfully. If successful, the following command will let you successfully login to MySQL shell:

```
kubectl run -it --rm percona-client --image=percona:8.4 --restart=Never --
bash -il
percona-client:/$ mysql -h ps-cluster1-mysql-primary -uuser1 -ppassword1
mysql> SELECT * FROM database1.table1 LIMIT 1;
```

You may also try executing any simple SQL statement to ensure the permissions have been successfully granted.

Connection secrets

For the `root` user, the Operator creates a Secret named `<cluster_name>-psuser-root` that contains ready-to-use connection details: hostname, port, username, password, connection URIs, and proxy endpoints. The Operator updates this Secret automatically when the primary changes or when proxy configuration changes.

For example, the `root` user on a cluster named `ps-cluster1` gets a Secret called `ps-cluster1-psuser-root`. Application Pods can mount this Secret directly instead of assembling connection parameters manually.

See Connection secrets for the full field reference, read-only connection guidance, and application integration examples.

System Users

To automate the deployment and management of the cluster components, the Operator requires system-level Percona Server for MySQL users.

Credentials for these users are stored as a Kubernetes Secrets object. The Operator requires to be deployed before the Percona Server for MySQL is started.

Note

The Operator will either use existing Secrets, or create a new Secrets object with randomly generated passwords if it didn't exist. Also, starting from the Operator version 0.5, it will generate random passwords for system users not found in the existing Secrets object.

The name of the required Secrets (`ps-cluster1-secrets` by default) should be set in the `spec.secretsName` option of the `deploy/cr.yaml` configuration file.

The following table shows system users' names and purposes.

> **Warning**
>
> These users should not be used to run an application.

User Purpose	Username	Password Secret Key	Description
Admin	root	root	Database administrative user, can be used by the application if needed
Orchestrator	orchestrator	orchestrator	Orchestrator administrative user and credentials for read/write access to the Orchestrator HTTP API (from Operator version 1.2.0 when `spec.crVersion` is `1.2.0` or higher)
Backup	xtrabackup	xtrabackup	User to run backups
Cluster Check	clustercheck	clustercheck	User for liveness checks and readiness checks
Monitoring	monitor	monitor	User for internal monitoring purposes and PMM agent
Operator Admin	operator	operator	Database administrative user, should be used only by the Operator
Replication	replication	replication	Administrative user needed for replication
ClusterSet	clusterset	clusterset	Administrative user for InnoDB ClusterSet AdminAPI operations and cross-site replication; should be used only by the Operator's ClusterSet controller
PMM Server token		pmmservertoken	The service token used to access PMM Server

Orchestrator HTTP API authentication

Starting with Operator version 1.2.0, the Orchestrator HTTP API on port `3000` requires authentication when `spec.crVersion` is `1.2.0` or higher. This applies to asynchronous replication clusters where Orchestrator manages failover and topology.

The Operator authenticates its own API calls automatically using the `orchestrator` system user from the cluster Secret. You do not need to configure anything for normal cluster operation.

If you call the Orchestrator API yourself (for example from a custom script or when Orchestrator is exposed outside the cluster), use one of the following:

- **`orchestrator` user** — read and write access. Use the password stored under the `orchestrator` key in the cluster Secret (`spec.secretsName`).
- **`readonly` user** — read-only access. Orchestrator accepts any password for this built-in user.

Unauthenticated requests receive HTTP `401`. Upgrading the Operator alone does not enable API authentication on existing clusters; set `spec.crVersion` to `1.2.0` or higher to activate it.

YAML Object Format

The default name of the Secrets object for these users is `ps-cluster1-secrets` and can be set in the CR for your cluster in `spec.secretName` to something different. When you create the object yourself, it should match the following simple format:

```
apiVersion: v1
kind: Secret
metadata:
  name: ps-cluster1-secrets
type: Opaque
stringData:
  root: root_password
  xtrabackup: backup_password
  monitor: monitor_password
  pmmserverkey: my_pmm_server_key
  operator: operator_password
  replication: replication_password
  clusterset: clusterset_password
  orchestrator: orchestrator_password
  heartbeat: heartbeat_password
```

As you can see, because we use the `stringData` type when creating the Secrets object, all values for each key/value pair are stated in plain text format convenient from the user's point of view. But the resulting Secrets object contains passwords stored as `data` - i.e., base64-encoded strings. If you want to update any field, you'll need to encode the value into base64 format. To do this, you can run `echo -n "password" | base64 --wrap=0` (or just `echo -n "password" | base64` in case of Apple macOS) in your local shell to get valid values. For example, setting the Admin user's password to `new_password` in the `ps-cluster1-secrets` object can be done with the following command:

in Linux

```
kubectl patch secret/ps-cluster1-secrets -p '{"data":{"root": "'$(echo -n new_password | base64 --wrap=0)'"}}'
```

in macOS

```
kubectl patch secret/ps-cluster1-secrets -p '{"data":{"root": "'$(echo -n new_password | base64)'"}}'
```

Password rotation policies and timing

When there is a change in user secrets, the Operator creates the necessary transaction to change passwords. This rotation happens almost instantly (the delay can be up to a few seconds), and it's not needed to take any action beyond changing the password.

Warning

Please don't change `secretName` option in CR, make changes inside the secrets object itself.

Passwords with special characters

The Operator automatically generates passwords for user secrets with special characters to increase security. It uses the following set of characters:

- Uppercase letters (A–Z)
- Lowercase letters (a–z)
- Digits (0–9)
- Special symbols: `! $ % & ( ) * + , - . < = > ? @ [ ] ^ _ { } ~ #`

This character set has been carefully selected to ensure correct functioning of SQL, shell scripts, YAML files and connection strings.

To avoid issues in these contexts, the following characters are excluded: single quotes (‘), double quotes (“), backslashes (), forward slashes (/), colons (:), pipes (|), semicolons (;) and backticks (`).

You can define passwords for user secrets yourself. When doing so, be sure to stick to the approved character set to ensure your services run smoothly.

Marking System Users In MySQL

Starting with MySQL 8.0.16, a new feature called Account Categories has been implemented, which allows us to mark our system users as such. See the official documentation on this feature for more details.

Connection secrets

The Operator automatically creates and maintains a Kubernetes Secret for the `root` user that contains all connection details your applications need. This gives developers and administrators a single, reliable source of connection information without manual configuration.

The Secret is owned by the Operator and updated on every reconciliation, so values such as the primary hostname stay current after failover or configuration changes.

Secret naming

The connection Secret follows this naming pattern:

```
<cluster_name>-psuser-<username>
```

`<cluster_name>` is the name of your Percona Server for MySQL cluster. `<username>` is the MySQL user name. Currently, the Operator creates this Secret only for the `root` user.

For a cluster named `ps-cluster1`, the Secret name is `ps-cluster1-psuser-root`.

Secret structure

The following example shows the fields the Operator populates in the connection Secret:

```
stringData:
  host: ps-cluster1-mysql-primary.default.svc
  port: "3306"
  user: root
  password: "w3_3ijSd)wulc.b};xwvT?Rj"
  uri: "mysql://root:w3_3ijSd%29wulc.b%7D%3BxwvT%3FRj@ps-cluster1-mysql-
primary.default.svc:3306"
  proxy-host: ps-cluster1-haproxy.default.svc
  proxy-port: "3306"
  proxy-uri: "mysql://root:w3_3ijSd%29wulc.b%7D%3BxwvT%3FRj@ps-cluster1-
haproxy.default.svc:3306"
  proxy-readonly-host: ps-cluster1-haproxy.default.svc
  proxy-readonly-port: "3307"
  proxy-readonly-uri: "mysql://root:w3_3ijSd%29wulc.b%7D%3BxwvT%3FRj@ps-
cluster1-haproxy.default.svc:3307"
  proxy-external-host: 34.118.45.21
  proxy-external-port: "3306"
  proxy-external-uri:
"mysql://root:w3_3ijSd%29wulc.b%7D%3BxwvT%3FRj@34.118.45.21:3306"
```

The table below describes each field and when it is present.

Field	Present	Description
`host`	Always	Hostname of the direct MySQL primary Service. Updated to point to the current primary after failover.
`port`	Always	Port the primary MySQL instance listens on (typically `3306`).
`user`	Always	MySQL user name.
`password`	Always	Password for the user.
`uri`	Always	Ready-to-use connection URI for the primary MySQL instance. Special characters in the password are URL-encoded.
`proxy-host`	When a proxy is enabled	Hostname of the proxy Service for read/write connections. The same field is used with HAProxy or MySQL Router.
`proxy-port`	When a proxy is enabled	Port for read/write connections through the HAProxy or MySQL Router, depending on which proxy is enabled. Port `3306` is typically for HAProxy while `6446` is for MySQL Router.
`proxy-uri`	When a proxy is enabled	Ready-to-use connection URI for read/write connections

		through the proxy. The same field is used with HAProxy or MySQL Router.
`proxy-readonly-host`	When a proxy is enabled	Hostname of the proxy Service for read-only connections. The same field is used with HAProxy or MySQL Router.
`proxy-readonly-port`	When a proxy is enabled	Port for read-only connections through the proxy (typically `3307` for HAProxy or `6447` for MySQL Router).
`proxy-readonly-uri`	When a proxy is enabled	Ready-to-use connection URI for read-only connections through the proxy. The same field is used with HAProxy or MySQL Router.
`proxy-external-host`	When the proxy Service type is LoadBalancer and an external IP is assigned	External hostname or IP for read/write connections from outside the cluster. The same field is used with HAProxy or MySQL Router.
`proxy-external-port`	When the proxy Service type is LoadBalancer and an external IP is assigned	External port for read/write connections.
`proxy-external-uri`	When the proxy Service type is LoadBalancer and an external IP is assigned	Ready-to-use external connection URI for read/write connections.

The `proxy-*` and `proxy-readonly-*` fields are generic. They do not indicate whether HAProxy or MySQL Router is configured underneath.

Note

Passwords for system users used by the Operator itself (backup, monitoring, replication, and so on) remain in the separate Secrets object configured with `spec.secretsName` (for example, `ps-cluster1-secrets`). See System users for details.

View connection details

To inspect the connection Secret, list Secrets in your namespace and look for the `<cluster_name>-psuser-root` object:

```
kubectl get secrets -n <namespace>
kubectl get secret ps-cluster1-psuser-root -n <namespace> -o yaml
```

To decode individual fields:

```
kubectl get secret ps-cluster1-psuser-root -n <namespace> \
  -o jsonpath="{.data['proxy-uri']}" | base64 --decode && echo
```

For keys that contain hyphens (such as `proxy-host` or `proxy-readonly-port`), use bracket notation in the jsonpath expression, as shown above.

Connect to MySQL

Choose the connection fields based on how your application reaches the database.

Read/write through the proxy (default)

Most clusters run with HAProxy or MySQL Router enabled. For connections from inside the Kubernetes cluster, use the `proxy-*` fields or `proxy-uri`:

```
mysql -h $(kubectl get secret ps-cluster1-psuser-root -n <namespace> \
  -o jsonpath="{.data['proxy-host']}" | base64 --decode) \
  -u$(kubectl get secret ps-cluster1-psuser-root -n <namespace> \
  -o jsonpath='{.data.user}' | base64 --decode) \
  -p"$(kubectl get secret ps-cluster1-psuser-root -n <namespace> \
  -o jsonpath='{.data.password}' | base64 --decode)" \
  -P $(kubectl get secret ps-cluster1-psuser-root -n <namespace> \
  -o jsonpath="{.data['proxy-port']}" | base64 --decode)
```

Or parse the ready-made URI:

```
kubectl get secret ps-cluster1-psuser-root -n <namespace> \
  -o jsonpath="{.data['proxy-uri']}" | base64 --decode && echo
```

Read-only through the proxy

Use the `proxy-readonly-*` fields to send read queries to replica nodes. With HAProxy, read-only traffic is routed on port `3307`. See Configure HAProxy for details.

```
mysql -h $(kubectl get secret ps-cluster1-psuser-root -n <namespace> \
  -o jsonpath="{.data['proxy-readonly-host']}" | base64 --decode) \
  -u$(kubectl get secret ps-cluster1-psuser-root -n <namespace> \
  -o jsonpath='{.data.user}' | base64 --decode) \
  -p"$(kubectl get secret ps-cluster1-psuser-root -n <namespace> \
  -o jsonpath='{.data.password}' | base64 --decode)" \
  -P $(kubectl get secret ps-cluster1-psuser-root -n <namespace> \
  -o jsonpath="{.data['proxy-readonly-port']}" | base64 --decode)
```

Warning

Setting only `proxy-readonly-host` without `proxy-readonly-port` does **not** route your session to read-only nodes. If you omit the port, the client falls back to the default MySQL port (`3306`), which connects to the primary in read/write mode. Always set both `proxy-readonly-host` and `proxy-readonly-port` (or use `proxy-readonly-uri`) for read-only connections.

Direct connection to the primary

To bypass the proxy and connect directly to the current primary Pod, use the `host`, `port`, `user`, and `password` fields, or the `uri` field:

```
mysql -h $(kubectl get secret ps-cluster1-psuser-root -n <namespace> \
  -o jsonpath='{.data.host}' | base64 --decode) \
  -u$(kubectl get secret ps-cluster1-psuser-root -n <namespace> \
  -o jsonpath='{.data.user}' | base64 --decode) \
  -p"$(kubectl get secret ps-cluster1-psuser-root -n <namespace> \
  -o jsonpath='{.data.password}' | base64 --decode)"
```

The `host` value always points to the direct MySQL primary Service and is updated automatically when failover changes the primary.

External connections

If you have exposed the proxy with a LoadBalancer Service and an external IP is assigned, use the `proxy-external-*` fields:

```
kubectl get secret ps-cluster1-psuser-root -n <namespace> \
  -o jsonpath="{.data['proxy-external-uri']}" | base64 --decode && echo
```

These fields appear only after the LoadBalancer receives an external IP address.

Use in application Pods

Mount the connection Secret into your application Pod to inject connection details as environment variables. To avoid naming conflicts, add a `prefix`:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
  - name: app
    image: my-app:latest
    envFrom:
    - prefix: MYSQL_
      secretRef:
        name: ps-cluster1-psuser-root
```

All Secret keys are exposed as environment variables with the same names and your prefix. For example, `MYSQL_host`, `MYSQL_proxy-host`, `MYSQL_proxy-readonly-port` and so on. Your application can read them directly:

```
import os
import pymysql

connection = pymysql.connect(
    host=os.environ["MYSQL_proxy-host"],
    user=os.environ["MYSQL_user"],
    password=os.environ["MYSQL_password"],
    port=int(os.environ["MYSQL_proxy-port"]),
)
```

Read-only connections in application code

When connecting to read-only nodes, pass both the read-only host and port:

```
connection = pymysql.connect(
    host=os.environ["MYSQL_proxy-readonly-host"],
    user=os.environ["MYSQL_user"],
    password=os.environ["MYSQL_password"],
    port=int(os.environ["MYSQL_proxy-readonly-port"]),
)
```

Using only `proxy-readonly-host` without `proxy-readonly-port` connects on the default port and targets the primary. Write operations on that connection succeed; read-only routing requires the read-only port.

Alternatively, use the ready-made `proxy-readonly-uri` field and pass it to a client that accepts connection URIs.

Backup and restore

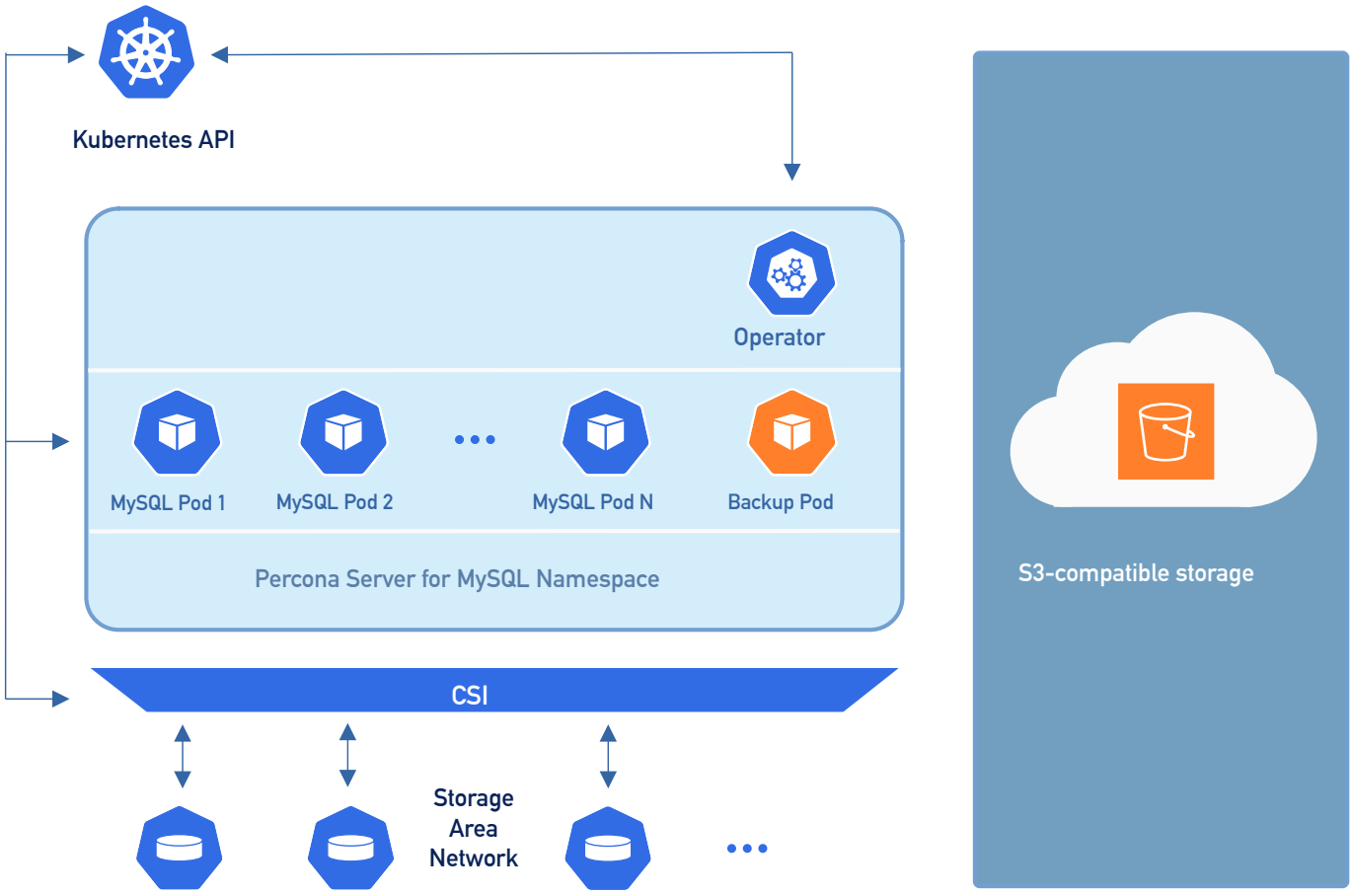
About backups

Backing up your database protects your data from loss and corruption, helps ensure business continuity, and lets you quickly recover if something goes wrong.

How backups work

The Operator stores your MySQL backups outside the Kubernetes cluster on cloud storage. You can use:

- Amazon S3 or S3-compatible storage
- Azure Blob Storage
- Google Cloud Storage



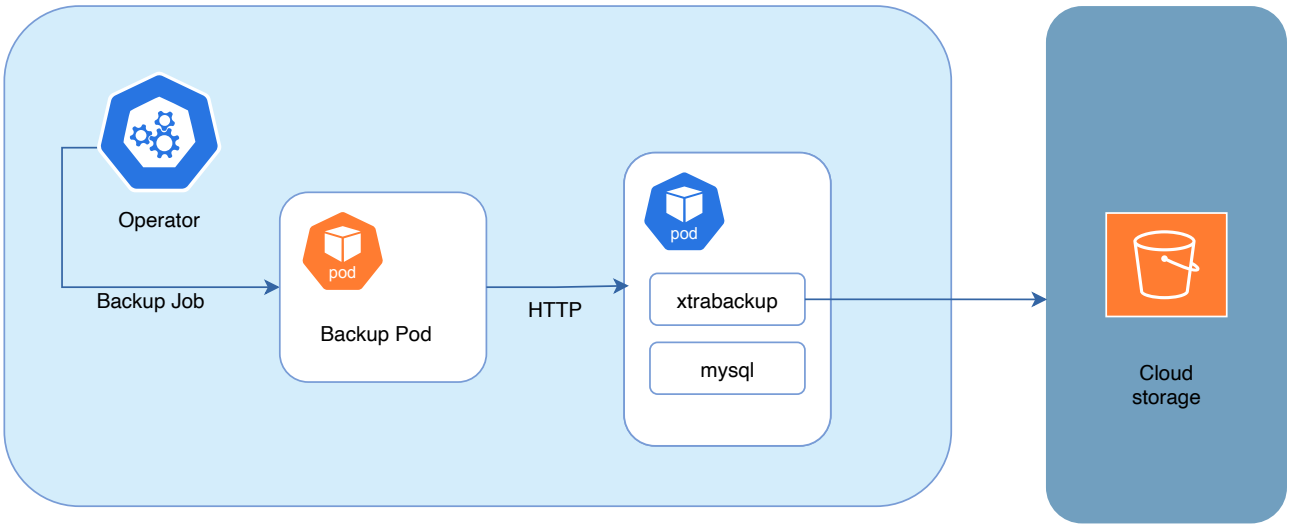
The Operator creates physical backups using Percona XtraBackup. Here's how it works:

1. Each database Pod includes a sidecar container called `xtrabackup` that runs an HTTP server.
2. When you create a backup, the Operator creates a Job that sends an HTTP request to the

backup source Pod.

3. The `xtrabackup` container receives the request and starts the backup process.
4. Backups are streamed to storage; the Operator does not keep a separate local copy of the backup on disk.

The following diagram outlines this workflow:



Backup types

You can make the following types of physical backups:

- Full backup - contains full data set
- An incremental backup - contains only the changes that occurred since the previous backup. To learn more, read Incremental backups.

Configuring backups

You configure backups in the `backup` section of your deploy/cr.yaml file. This section includes:

- The backup.enabled key, which you set to `true` to enable backups
- The backup.schedule.type key to set the type for scheduled backups.
- The PerconaServerMySQLBackup.spec.type field to set the type for on-demand backups.
- The `storages` subsection, where you configure access to your cloud storage

- Optional backup encryption settings to protect backup files in the object storage

Backup runs

You can create backups in two ways:

- **Scheduled backups**: Configure these in your deploy/cr.yaml file. The Operator runs them automatically at the times you specify.
- **On-demand backups**: Create these manually whenever you need them. You configure them in the deploy/backup/backup.yaml file.

Point-in-time recovery

Starting with Operator 1.1.0, you can combine a base backup with archived binary logs to restore to a specific GTID or timestamp. That flow uses a binlog server and object storage alongside your normal backup configuration.

Configure storage for backups

You configure backup storage in the `backup.storages` subsection of your Custom Resource using the deploy/cr.yaml file.

Before configuring storage, you need to create a Kubernetes Secret object that contains the credentials needed to access your storage.

To encrypt backups in object storage, create a separate Secret with your encryption key and reference it in `spec.backup.encryptionKeySecret` or under the storage entry.

Amazon S3 or S3-compatible storage

To use Amazon S3 or S3-compatible storage for backups, create a Secret object with your access credentials. Use the deploy/backup/backup-secret-s3.yaml file as an example. You must specify the following information:

- `name` is the name of the Kubernetes secret which you will reference in the Custom Resource
- `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` are base64-encoded keys to access S3 storage

  Use the following command to encode the keys:

  #### in Linux

  ```
  echo -n 'plain-text-string' | base64 --wrap=0
  ```

  #### in macOS

  ```
  echo -n 'plain-text-string' | base64
  ```

Here's the example configuration of the Secret file:

deploy/backup/backup-secret-s3.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: ps-cluster1-s3-credentials
type: Opaque
data:
  AWS_ACCESS_KEY_ID: UkVQTEFDRS1XSVRILUFXUy1BQ0NFU1MtS0VZ
  AWS_SECRET_ACCESS_KEY: UkVQTEFDRS1XSVRILUFXUy1TRUNSRVQtS0VZ
```

1. Create the Secret object with this file:

   ```
   kubectl apply -f deploy/backup/backup-secret-s3.yaml -n <namespace>
   ```

2. Configure the storage in the Custom Resource. Modify the deploy/cr.yaml file and define the following information:

   - `bucket` where the data will be stored
   - `region` - location of the bucket
   - `credentialsSecret` - the name of the Secret you created previously

   Here's the example:

   ```
   backup:
     enabled: true
     ...
     storages:
       s3-us-west:
         type: s3
         s3:
           bucket: S3-BACKUP-BUCKET-NAME-HERE
           region: us-west-2
           credentialsSecret: ps-cluster1-s3-credentials
   ```

   S3-compatible storage

   If you use S3-compatible storage instead of Amazon S3, add the `endpointUrl` option in the `s3` subsection. This points to your storage service and is specific to your cloud provider. For example, for MinIO:

   ```
   endpointUrl: https://minio-service:9000
   ```

Organizing backups

You can use the prefix option to specify a path (sub-folder) inside the S3 bucket where backups will be stored. If you don't set a prefix, backups are stored in the root directory.

3. Apply the configuration:

```
kubectl apply -f deploy/cr.yaml -n <namespace>
```

For more configuration options, see the Operator Custom Resource options.

Microsoft Azure Blob storage

To use Azure Blob Storage for storing backups, create a Secret object with your access credentials. Use the deploy/backup/backup-secret-azure.yaml file as an example. You must specify the following information:

- `name` is the name of the Kubernetes secret which you will reference in the Custom Resource
- `AZURE_STORAGE_ACCOUNT_NAME` and `AZURE_STORAGE_ACCOUNT_KEY` are base64-encoded credentials to access Azure Blob storage

  Use the following command to encode the credentials:

  in Linux

  ```
  echo -n 'plain-text-string' | base64 --wrap=0
  ```

  in macOS

  ```
  echo -n 'plain-text-string' | base64
  ```

Here's the example configuration of the Secret file:

deploy/backup/backup-secret-azure.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: ps-cluster1-azure-credentials
type: Opaque
data:
  AZURE_STORAGE_ACCOUNT_NAME: UkVQTEFDRS1XSVRILUFXUy1BQ0NFU1MtS0VZ
  AZURE_STORAGE_ACCOUNT_KEY: UkVQTEFDRS1XSVRILUFXUy1TRUNSRVQtS0VZ
```

1. Create the Secret object with this file:

   ```
   kubectl apply -f deploy/backup/backup-secret-azure.yaml -n <namespace>
   ```

2. Configure the storage in the Custom Resource. Modify the deploy/cr.yaml file and define the following information:

   - `container` where the data will be stored
   - `credentialsSecret` - the name of the Secret you created previously

   Here's the example:

   ```
   backup:
     enabled: true
     ...
     storages:
       azure-blob:
         type: azure
         azure:
           container: <your-container-name>
           credentialsSecret: ps-cluster1-azure-credentials
   ```

3. Apply the configuration:

   ```
   kubectl apply -f deploy/cr.yaml -n <namespace>
   ```

For more configuration options, see the Operator Custom Resource options.

Google Cloud Storage

To use Google Cloud Storage for storing backups, create a Secret object with your access credentials. Use the deploy/backup/backup-secret-gcp.yaml file as an example. You must specify the following information:

- `name` is the name of the Kubernetes secret which you will reference in the Custom Resource
- `ACCESS_KEY_ID` and `SECRET_ACCESS_KEY` are base64-encoded keys to access GCS storage

  Use the following command to encode the keys:

  in Linux

  ```
  echo -n 'plain-text-string' | base64 --wrap=0
  ```

  in macOS

  ```
  echo -n 'plain-text-string' | base64
  ```

Here's the example configuration of the Secret file:

deploy/backup/backup-secret-gcp.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: ps-cluster1-gcp-credentials
type: Opaque
data:
  ACCESS_KEY_ID: Z2NwLWFjY2Vzcy1rZXkK
  SECRET_ACCESS_KEY: Z2NwLXNlY3JldC1rZXkK
```

1. Create the Secret object with this file:

   ```
   kubectl apply -f deploy/backup/backup-secret-gcp.yaml -n <namespace>
   ```

2. Configure the storage in the Custom Resource. Modify the deploy/cr.yaml file and define the following information:

   - `bucket` where the data will be stored
   - `credentialsSecret` - the name of the Secret you created previously

   Here's the example:

```
backup:
  enabled: true
  ...
  storages:
    gcp-cs:
      type: gcs
      gcs:
        bucket: GCS-BACKUP-BUCKET-NAME-HERE
        credentialsSecret: ps-cluster1-gcp-credentials
```

3. Apply the configuration:

```
kubectl apply -f deploy/cr.yaml -n <namespace>
```

For more configuration options, see the Operator Custom Resource options.

Incremental backups

Version added: 1.1.0

This feature is in the tech preview stage. The behavior can change in future releases.

If your database grows quickly or sees heavy write traffic, you may want frequent backups. Taking a full backup every time costs more storage, takes longer to upload, and adds load to the cluster.

Incremental backups copy only what changed since the previous backup in the chain. They are usually smaller, faster to transfer, and cheaper to keep in the backup storage.

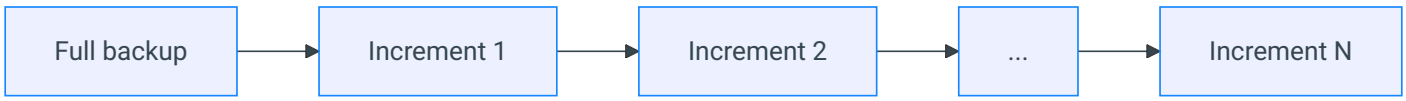
What you need first

Every incremental backup belongs to a **chain** that starts with one **full** backup on the **same** storage. The Operator checks that a valid full backup exists before it starts an incremental one. Using the same storage for the whole chain enables the Operator to reuse the same credentials and paths, and simplifies restore.

You can use any supported storage type.

What the backup chain looks like

Incremental backups build on the full backup and on each other in order:



By default, the Operator uses the **latest full backup** as the base for both **scheduled** and **on-demand** incremental backups. If you want more control, you can explicitly specify the base backup in the configuration for on-demand backups. The Operator validates the specified backup and derives the incremental chain from it.

How an incremental backup runs

1. You create a `PerconaServerMySQLBackup` object with type **incremental**. Or, you configure the

backup schedule in the cluster Custom Resource that creates incremental backups.

2. The Operator confirms that a **full** backup exists on the same storage and is valid. Unless you specify another backup, it uses the **most recent** full backup.
3. If the backup is valid, the Operator sets the `percona.com/base-backup-name` annotation on it so that it serves as the base for the incremental backup chain.
4. If there are already incremental backups derived from the base, the Operator takes the `to_lsn` value from the previous increment and sets it as the `from_lsn` value for the new increment so the chain stays continuous.
5. The Operator streams the new incremental to the same storage.
6. The Operator records the backup type on the backup object.



Important

The `percona.com/base-backup-name` annotation is internal and serves to correctly link incremental backups to the base one. Removing or editing it will lead to unpredictable results and data corruption. Don't remove or edit this annotation.

How restore from an incremental backup works

The restore flow is unified for both full and incremental backups. The Operator identifies the backup type by name or destination. To identify the increments and reconstruct the chain, the backup destination has now the `.incr` path segment. The Operator downloads the full backup and all related increments and sorts them in the correct order. Then it restores the full backup first and applies each incremental backup.

If you make a point-in-time recovery, it also applies binlogs on top, after restoring all backups. To learn more how it works, check the point-in-time recovery workflow.

Here's how it works in detail:

1. You create a `PerconaServerMySQLRestore` object and reference the incremental backup with `backupName` (same cluster) or `backupSource` (remote path / another environment).
2. The Operator detects that the target is incremental and resolves the chain.
3. It connects to the storage and lists full backup plus all increments up to and including the one you chose, using paths that include the `.incr` segment so increments are unambiguous.
4. The Operator sorts increments in the right order.

5. The Operator pauses the cluster for the restore.
6. The restore Job downloads and applies the full backup, then each incremental one in order.
7. The cluster is unpaused when the restore completes.

Path layout

Incremental destinations use an `.incr` segment in the path so the Operator can tell full and incremental artifacts apart. A typical pattern resembles `prefix/<cluster>-<timestamp>-full.incr/<cluster>-<timestamp>-incr`; exact layout follows your `prefix` and storage settings.

Here's the example:

```
s3://bucket/prefix/
my-cluster-2026-04-06-full/           # base full backup
my-cluster-2026-04-06-full.incr/   # incremental chain directory
    my-cluster-2026-04-07T000000-incr/  # Monday's incremental
    my-cluster-2026-04-08T000000-incr/    # Tuesday's incremental
    my-cluster-2026-04-09T000000-incr/    # Wednesday's incremental
```

Why you need incremental backups

With incremental backups, you gain the following benefits:

- strengthen your backup strategy by creating multiple restore points
- increase storage efficiency by avoiding duplication of unchanged data
- lower system load, since smaller backups require fewer compute resources and reduce impact on your cluster
- reduce both storage and data-transfer costs

Implementation specifics and rules

Backup chain rules

1. A full backup is required to start the incremental chain. If none exists, the incremental backup fails.
2. A base full backup and incremental backups derived from it must be **on the same storage**.
3. By default, the Operator uses the most recent full backup to start the incremental chain. You can explicitly specify the base full backup in the `spec.incrementalBaseBackupName` option in the

backup configuration file. If the specified full backup is valid, the Operator starts the incremental backup chain from it.

4. If the base backup already has the incremental backup chain, the Operator uses the most recent increment to continue the chain.
5. Retention applies to the chain as a unit: deleting the **base** full backup removes the **entire** incremental chain that depends on it, so you do not leave orphaned increments. Specifying the retention policy for increments is not supported.
6. You cannot delete an increment in the middle of a chain as it would break its continuity. You can delete only the **last** increment in the chain or the base backup, which removes the whole chain.
7. The Operator prevents to run two concurrent incremental backups against the **same** chain to avoid ambiguous ordering. The Operator runs increments one by one.

Restore rules

1. You can make either an in-place restore pointing at an incremental backup object in the `backupName` option, or make a cross-cluster restore specifying the incremental backup path for the `backupSource.destination` option.
2. Restores that use `backupSource` work across clusters and namespaces when the storage destination is reachable; incremental paths remain discoverable because of the `.incr` layout.
3. Restore always needs the full chain: full backup first, then increments in order, up to the backup you selected.

Known limitations

- If a backup in your chain fails but some data was already uploaded to storage, your restore still includes that failed backup, so the restore fails. Whenever any backup in the chain fails, start a new chain.
- If the checkpoint file is missing from a backup directory, your next incremental backup can hang. Start a new chain.

Making scheduled backups

Scheduled backups run automatically at times you specify. You configure them in the `backup` section of your Custom Resource using the deploy/cr.yaml file.

Before setting up scheduled backups, make sure you have at least one configured storage in the `backup.storages` subsection.

Configure a backup schedule

To schedule backups, specify the following configuration in your Custom Resource:

1. Enable backup by setting `backup.enabled` to `true`.
2. Add entries to the `backup.schedule` subsection in your Custom Resource. Each schedule entry requires the following:
   - `name` - a unique name for this backup schedule
   - `schedule` - the backup schedule in crontab format. For example, `"0 0 * * 6"` runs every Saturday at midnight.
   - `storageName` - the name of your configured storage where backups will be stored
   - `type` (optional) - the backup type: `full` (default) or incremental
   - `keep` (optional) - the number of backups to keep in storage. Older backups are automatically deleted when this limit is reached. Note that this option is ignored for incremental backups.

Here's an example configuration that creates a **full backup** every Sunday at 2:00 AM and incremental backups **every day** at 3:00 AM. The configuration keeps the last 3 full backups and incremental chains derived from them:

```
...
backup:
  enabled: true
  storages:
    s3-us-west:
      type: s3
      s3:
        bucket: S3-BACKUP-BUCKET-NAME-HERE
        region: us-west-2
        credentialsSecret: ps-cluster1-s3-credentials
  schedule:
    - name: "weekly-full-backup"
      schedule: "0 2 * * 0"              # Every Sunday at 2:00 AM
      type: "full"
      keep: 3
      storageName: s3-us-west
    - name: "daily-incremental-backup"
      schedule: "0 3 * * 1-6"            # Monday through Saturday at 3:00 AM
      type: "incremental"
      storageName: s3-us-west
  ...
```

Managing multiple backup schedules in the same storage

You can define multiple backup schedules to meet different recovery and compliance needs. For example, you might want daily backups for quick recovery from recent changes and monthly backups for long-term retention or audit purposes.

When you use the same storage location for multiple schedules, be aware of these important limitations:

1. **Retention policy conflicts**: The Operator only applies retention policies to the first schedule in your configuration. For example, if you set daily backups to keep 5 copies and monthly backups to keep 3 copies, the Operator will only keep 5 total backups in storage, not 8 as you might expect. However, all backup objects will still appear in `kubectl get ps-backup` output.
2. **Concurrent backup conflicts**: When multiple schedules run at the same time and write to the same storage path, backups can overwrite each other, resulting in incomplete or corrupted data.

To avoid these issues and ensure each schedule maintains its own retention policy, configure separate storage locations for each schedule. Here's how:

1. Create separate storage configurations in your Custom Resource with different names
2. Specify unique buckets or prefixes for each storage configuration
3. Assign different storage names to specific schedules

Here's an example configuration that uses separate storage locations for daily and monthly backups:

```
storages:
  minio1:
    type: s3
    s3:
      credentialsSecret: minio-secret
      bucket: my-bucket
      prefix: minio1
      endpointUrl: "http://minio.minio.svc.cluster.local:9000/"
    verifyTLS: false
  minio2:
    type: s3
    s3:
      credentialsSecret: minio-secret
      bucket: my-bucket
      prefix: minio2
      endpointUrl: "http://minio.minio.svc.cluster.local:9000/"
    verifyTLS: false

....

backup:
  schedule:
  - name: "daily-backup"
    schedule: "0 2 * * *"
    keep: 2
    storageName: minio1
  - name: "monthly-backup"
    schedule: "0 2 1 * *"
    keep: 5
    storageName: minio2
```

In this example, daily backups are stored in `minio1` and monthly backups are stored in `minio2`, each with their own retention policy.

Monitoring scheduled backups

When a scheduled backup runs, the Operator creates a Kubernetes Job to execute the backup. You can view these jobs to monitor backup execution:

```
kubectl get jobs -n <namespace>
```

Example output

```
NAME                                                    COMPLETIONS   DURATION   AGE
xb-cron-ps-cluster2-gcp-20251107152047-9mgo5-gcp        1/1           9s         21s
```

To find all backups created by a specific schedule, use the `percona.com/backup-ancestor` label. The label format is `percona.com/backup-ancestor: <hash>-<schedule-name>`, where `<hash>` is a unique identifier and `<schedule-name>` is the name you specified in your schedule configuration.

For example, to find all backups created by the `sat-night-backup` schedule:

```
kubectl get ps-backup -l percona.com/backup-ancestor -n <namespace>
```

Or to filter for a specific schedule:

```
kubectl get ps-backup -l percona.com/backup-ancestor=0ed1d-sat-night-backup -
n <namespace>
```

Troubleshooting

If you face issues with backups, refer to our Backup troubleshooting guide for help.

Making on-demand backup

Before you begin

1. Export the namespace as an environment variable. Replace the `<namespace>` placeholder with your value:

   ```
   export NAMESPACE = <namespace>
   ```

2. Check the configuration of the `PerconaServerMySQL` object:

   - Check that the `backup.enabled` key is set to `true`. Use the following command:

     ```
     kubectl get ps <cluster-name> -n $NAMESPACE -o
     jsonpath='{.spec.backup.enabled}'
     ```

   - Verify that you have configured backup storage and specified its configuration in the `backup.storages` subsection of the Custom Resource.

3. For incremental backups, check implementation specifics and rules.

Backup steps

To make an on-demand backup, use *a special backup configuration YAML file*. The example of such file is deploy/backup/backup.yaml.

You can check available options in the Backup resource reference

Specify the following keys:

- Set the `metadata.name` key to assign a name to the backup.
- Set the `spec.clusterName` key to the name of your cluster.
- Set the `spec.storageName` key to a storage configuration defined in your `deploy/cr.yaml` file.
- Set the `spec.type` key to make an incremental backup. When unset, a full backup is made by default.
- Optionally, specify the base backup for the `spec.incrementalBaseBackupName`. This option is only valid for the `spec.type=incremental`.

- Optionally, add the `percona.com/delete-backup` entry under `metadata.finalizers` to enable deletion of backup files from a cloud storage when the backup object is removed (manually or by schedule).

Pass this information to the Operator:

via the YAML manifest

1. Edit the `deploy/backup/backup.yaml` file:

```
apiVersion: ps.percona.com/v1
kind: PerconaServerMySQLBackup
metadata:
  name: backup1
  finalizers:
    - percona.com/delete-backup
spec:
  clusterName: ps-cluster1
  storageName: s3-us-west
```

2. Start the backup process:

```
kubectl apply -f deploy/backup/backup.yaml -n $NAMESPACE
```

via the command line

Instead of storing backup settings in a separate file, you can pass them directly to the `kubectl apply` command as follows:

```
cat <<EOF | kubectl apply -n $NAMESPACE -f-
apiVersion: ps.percona.com/v1
kind: PerconaServerMySQLBackup
metadata:
  name: backup1
  finalizers:
    - percona.com/delete-backup
spec:
  clusterName: ps-cluster1
  storageName: s3-us-west
EOF
```

List backups with this command:

```
kubectl get ps-backup -n $NAMESPACE
```

Specifying the backup source

When you create a backup object, the Operator selects a Pod to take the backup from. You can see the backup source pod in the backup object's status:

```
kubectl get ps-backup backup1 -o yaml
```

Sample output

```
apiVersion: ps.percona.com/v1
kind: PerconaServerMySQLBackup
metadata:
  name: backup1
  ...
status:
  backupSource: cluster1-mysql-1.cluster1-mysql.<namespace>
  ...
```

You can specify the source pod in the backup object to run the backup on this specific pod:

```
apiVersion: ps.percona.com/v1
kind: PerconaServerMySQLBackup
metadata:
  name: backup1
  finalizers:
    - percona.com/delete-backup
spec:
  clusterName: ps-cluster1
  storageName: s3-us-west
  sourcePod: ps-cluster1-mysql-2
```

Troubleshooting

If you face issues with backups, refer to our Backup troubleshooting guide for help.

Point-in-time recovery

This feature is in the tech preview stage. The behavior can change in future releases.

A base backup captures your data at a single moment in time. Restoring from such a backup is enough in many cases. However, if you need to undo a bad migration, recover right before someone dropped the wrong table, or meet a tighter recovery point, a base backup alone won't give you that level of precision.

To address these use cases, you must be able to restore the database to a specific point in time or to a specific transaction. To do that means to apply the binary logs generated after the backup up to that particular point/transaction target, as they contain all subsequent database changes leading to it. This process is called point-in-time recovery and is available in the Operator starting with version 1.1.0.

This feature is in the tech preview stage. We don't recommend using it in production yet, but we encourage you to try it out and share your feedback.

How the Operator performs point-in-time recovery

The Operator implements point-in-time recovery in two stages:

1. Restores a base physical backup taken with `PerconaXtraBackup`
2. Replays binary logs on top of the base backup to bring the database to the desired point in time or transaction. A binary log records all changes made to the database, such as updates, inserts, and deletes.

The Operator decodes binary logs using the `mysqlbinlog` client and applies them sequentially.

Recovery modes

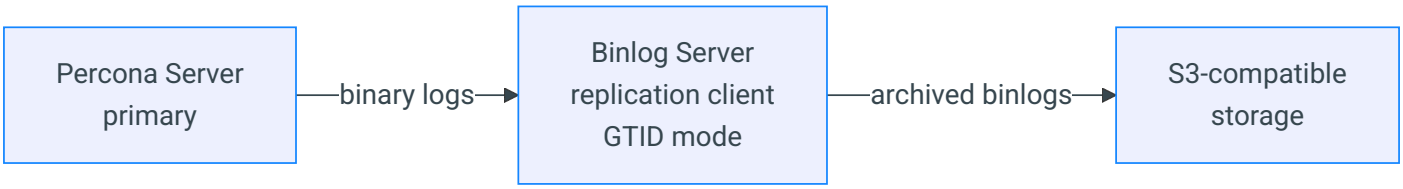
You can restore the database up to a specific moment in time or up to a specific transaction. You specify the desired mode in the `PerconaServerMySQLRestore` object:

Mode	What you specify	How it works	Typical use case

GTID	A GTID set	The Operator restores the database up to that transaction.	Precise, replication-friendly recovery when you know the GTID to stop before
Date / time	A timestamp	The Operator restores the database up to the specified timestamp	Use when wall-clock time is easier than GTIDs

Binary log collection

The Operator requires binary logs (binlogs) for point-in-time recovery. The Operator collects them using the Percona Binlog Server command-line utility. The Binlog Server connects to MySQL as a replication client and uploads binlogs to a dedicated object storage location. It can resume collection automatically after being interrupted or stopped.



To enable binlog collection, configure the `spec.backup.pitr` section in your Custom Resource:

```
spec:
  backup:
    pitr:
      enabled: true
      binlogServer:
        storage:
          s3:
            bucket: my-binlogs
            credentialsSecret: my-s3-secret
            region: us-east-1
            prefix: binlogs
            endpointURL: https://s3.amazonaws.com
```

After you apply the changes, the Operator starts the Percona Binary log Server Pod, which continuously collects binlogs and stores them in the configured location.

Point-in-time recovery workflow

Let's have a closer look at how point-in-time recovery works:

When you trigger a restore with a point-in-time recovery target, the Operator does the following:

1. Execs into the Binlog Server Pod and searches for the required binlogs
2. Pauses the cluster
3. Identifies the closest base backup leading to that point and restores it
4. Runs the point-in-time recovery Job that:
    - Starts a temporary `mysqld` instance
    - Retrieves binlogs from the storage and applies them to the `mysql` client
    - Shuts down the temporary `mysqld` instance
5. Rebootstraps the cluster based on the restored dataset.

For how to configure and start a point-in-time recovery, refer to the Make a restore with point-in-time recovery on the same cluster and Make a restore with point-in-time recovery on the new cluster tutorials.

With point-in-time recovery, you get finer control over when you come back online, which improves recovery point objective (RPO)

Implementation specifics

1. Point-in-time recovery is supported for both asynchronous and group replication topologies.
2. The Binlog Server is deployed with the number of Pods restricted to 1. This is because it connects to MySQL as a replication client with a specific server ID and only one instance can connect to the database with the same ID. Any number you set for the `size` in the Custom Resource will be ignored.
3. Enabling the Binlog Server requires the GTID mode to be enabled on the cluster for replaying binary logs. This mode is enabled by default.
4. The cluster is paused during point-in-time recovery. The Operator starts a temporary `mysqld` pod to perform the restore operation using the data PVC.
5. The Binlog Server must be running before the restore begins. The reconciler locates the required binlogs before pausing the cluster.
6. The Binlog Server stores binlogs in a dedicated folder. Therefore, you must specify the folder name (`prefix`) when configuring storage for the Binlog Server.

Known limitations

- The Binlog Server currently supports only AWS S3 and S3-compatible storage services for streaming binlogs. Use the same practices as backups: provide credentials via Kubernetes Secrets, set the endpoint URL (including scheme if required), region, and TLS options to match your environment.
- If the Operator user password is different from the password saved in the base backup, point-in-time recovery will fail. You must take a new full backup after changing the Operator user password to ensure point-in-time recovery works.
- During point-in-time recovery, updates made by `mysql-shell` can result in replication errors such as:

```
2026-04-15T10:58:08.911878Z 11 [ERROR] [MY-010584] [Repl] Replica SQL: Could not execute Update_rows event on table mysql_innodb_cluster_metadata.instances; Can't find record in 'instances', Error_code: 1032; handler error HA_ERR_KEY_NOT_FOUND; the event's source log FIRST, end_log_pos 162397, Error_code: MY-001032
```

If you want to ignore such errors, add `force: true` to the PITR section of your restore spec:

```
apiVersion: ps.percona.com/v1
kind: PerconaServerMySQLRestore
metadata:
  name: restore1
spec:
  clusterName: cluster1
  backupName: backup1
  pitr:
    force: true
    type: date
    date: "2026-04-16 21:12:00"
```

The `force: true` option enables the `--force` flag with the MySQL client and will silently ignore all SQL errors during binlog replay. **Warning:** This might result in data loss if underlying replication or data integrity errors are ignored.

- Binlog storage for point-in-time recovery is configured separately from base backup storage. The Operator uses the settings under `spec.backup.pitr.binlogServer` of the cluster Custom Resource during the in-place restore on the same cluster. For cross-cluster restore, specify the Binlog storage settings under `spec.pitr.backupSource.binlogServer` in the restore object.

- Point-in-time recovery job retries are not idempotent. If recovery fails after the base backup is restored, a retry will not restore the full backup again to reset the state. We recommend setting `spec.backup.backoffLimit=0` in your `cr.yaml` to prevent automatic job retries.
- If something fails mid-restore, use the same discipline as with any restore: inspect `PerconaServerMySQLRestore` **status**, the **restore and PITR jobs**, and refer to our Restore troubleshooting guide.\
- Data at rest encryption is not supported with point-in-time recovery.
- You cannot change the prefix for the Binlog Server.
- The Binlog Server may encounter issues if the number of objects in the point-in-time recovery bucket becomes too high. In such situations, the Binlog Server can get stuck and enter a CrashLoopBackOff state, unable to proceed with point-in-time recovery operations. This is due to how the Binlog Server processes or lists these objects internally. To recover from this state, you need to delete old objects from the bucket manually. If the issue isn't caught within the binlog's expiration period, restoring the database becomes significantly more difficult. To reduce the risk of getting into this situation, monitor the object count in your point-in-time recovery bucket and clean up binlogs regularly.

Backup compression support in Percona Operator for MySQL

Percona Operator for MySQL uses Percona XtraBackup (PXB) to create backups. The Operator exposes this capability too, supporting compression using the `zstd` compression algorithm.

Compression works for both full and incremental backups, and you can configure it for scheduled and on-demand backups.

With this ability to compress backups, you can reduce the size of your backups, and lower storage and data-transfer costs.

Configure global compression

You can configure compression using the MySQL configuration file. This setting applies to both scheduled and on-demand backups for all backup storages that you use.

```
spec:
  mysql:
    image: percona/percona-server:8.4.10-10.1
    configuration: |
      ...
      [xtrabackup]
      compress=zstd
      ...
```

You can override the compression for scheduled backups per specific storage or for a particular on-demand backup.

Configure compression for scheduled backups

In the cluster Custom Resource, add compression flags under the storage entry your schedules use:

```
spec:
  backup:
    storages:
      <storage-name>:
        containerOptions:
          args:
            xtrabackup:
              - "--compress"
```

Configure compression for on-demand backups

For on-demand backups, define the compression settings directly in the configuration file for the `PerconaServerMySQLBackup` object:

```
apiVersion: ps.percona.com/v1
kind: PerconaServerMySQLBackup
metadata:
  name: backup1-compressed
  finalizers:
    - percona.com/delete-backup
spec:
  clusterName: ps-cluster1
  storageName: s3-us-west
  containerOptions:
    args:
      xtrabackup:
        - "--compress"
```

It's important to understand how the Operator prioritizes compression and other backup tool settings when they're defined both globally in your cluster's Custom Resource (CR) and individually for a specific backup or restore job. Percona Operator gives precedence to the settings in an individual `PerconaServerMySQLBackup` or `PerconaServerMySQLRestore` object, allowing you to override cluster-wide defaults on a per-job basis. For a detailed explanation of how these options apply, see Fine-tuning backup and restore operations.

Restore from a compressed backup

You can make a restore either using the `backupName` or the `backupSource` options.

The Operator detects if this backup is compressed and automatically decompresses during the preparation stage for the restore.

Encrypted backups

Version added: 1.2.0

Percona Operator for MySQL uses Percona XtraBackup to create physical backups. By default, backups are unencrypted. To add an extra security layer, you can encrypt backup files before they are stored in your cloud object storage. Encryption is applied inline during the backup stream, so files in the storage stay protected at rest even if the storage account is shared or accessed without authorization. This helps you meet compliance requirements without adding external encryption tools or changing how you run scheduled and on-demand backups.

Encryption is available for the Group Replication and asynchronous replication cluster types.

Backup flow

You can encrypt both full and incremental backups, whether they are scheduled or on-demand. When encrypting incremental backups, the key must be the same for every backup in an incremental chain.

You can enable encryption globally for all backups, or set different encryption options for each storage. The setting you configure for a specific storage overrides the global setting. This flexibility lets you tailor backup security to your unique requirements.

Restore flow

During the in-place restore to the same cluster, the Operator recognizes the encrypted backup and uses the same encryption key Secret to decrypt the data.

For a restore on a new cluster or in a different Kubernetes environment, provide the same encryption key in the Restore resource. This is required because the Operator has no access to the encryption key Secrets from the source cluster.

The key used for the restore must match the key used to encrypt the backup. If the key is lost or rotated, the backup cannot be recovered.

Configuration

Create an encryption key Secret

Create a Kubernetes Secret that holds the encryption key. The encryption key must be base64-encoded.

Generate the encryption key using the following command:

```
openssl rand -base64 24
```

Create a Secret configuration file and specify the generated string within. Here's the example of the file `encryption-key.yaml`:

encryption-key.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: my-encryption-key
stringData:
  encryptionKey: <encryption-key-string>
```

Create the Secret object:

```
kubectl apply -f encryption-key.yaml -n <namespace>
```

The Operator encodes the key when it creates the Secret.

Store this Secret securely and back it up separately from your database backups. If you lose the key, encrypted backups cannot be restored.

Enable encryption for all backups

1. Edit the `deploy/cr.yaml` cluster Custom Resource and reference the Secret under `spec.backup.encryptionKeySecret`. Specify the name. The `key` field is optional and defaults to `encryptionKey`. The Operator never stores the key value, it only references it via the Secret.

Here's the sample configuration:

```
spec:
  backup:
    enabled: true
    encryptionKeySecret:
      name: my-encryption-key
      key: encryptionKey
    storages:
      s3-us-west:
        type: s3
        s3:
          bucket: S3-BACKUP-BUCKET-NAME-HERE
          region: us-west-2
          credentialsSecret: ps-cluster1-s3-credentials
```

1. Apply the new configuration:

   ```
   kubectl apply -f deploy/cr.yaml -n <namespace>
   ```

This configuration applies to all backup storages unless a storage entry defines its own key.

Enable encryption for a specific storage

To use a different key for a particular storage location, create a new Secret that stores this key.

Next, configure the `encryptionKeySecret` subsection in the Custom Resource under that storage entry.

This example configuration shows how to override encryption settings for the S3 storage. The new encryption key is stored within the Secret named `my-encryption-key-s3`:

```
spec:
  backup:
    encryptionKeySecret:
      name: my-encryption-key
      key: encryptionKey
    storages:
      s3-us-west:
        encryptionKeySecret:
          name: my-encryption-key-s3
          key: encryptionKey
        type: s3
        s3:
          bucket: S3-BACKUP-BUCKET-NAME-HERE
          region: us-west-2
          credentialsSecret: ps-cluster1-s3-credentials
```

After you apply the configuration, the storage-level encryption settings take precedence over the global settings.

Tune the encryption algorithm and performance

The Operator uses **AES256** by default. To use a different cipher or adjust encryption performance, pass Percona XtraBackup options through `containerOptions.args.xtrabackup` in the cluster Custom Resource or in a specific backup or restore manifest. See Fine-tuning backup and restore operations for how global and per-job settings interact.

The example configuration shows how to tune Percona XtraBackup globally on the cluster level.

```
spec:
  backup:
    storages:
      s3-us-west:
        containerOptions:
          args:
            xtrabackup:
              - "--encrypt=AES192"
              - "--encrypt-threads=3"
              - "--encrypt-chunk-size=64K"
```

Make an encrypted backup

You don’t need any additional configuration to make an encrypted backup. The Operator uses the encryption settings from the Custom Resource for both on-demand and scheduled backups.

Restore from an encrypted backup on the same cluster

When you restore using `backupName` on the same cluster where the backup was created, the Operator reads the encryption key from the cluster Custom Resource automatically. No extra configuration is required as long as the same `encryptionKeySecret` is still defined in the cluster.

See Restore the cluster from a previously saved backup.

Restore from an encrypted backup on a new cluster

When you restore from a backup on a different cluster or in a different Kubernetes environment, you must provide the encryption key in the restore resource. The Operator has no access to Secrets from the source cluster.

The key used for restore **must match** the key that was used to encrypt the backup.

See Restore from a backup to a new Kubernetes-based environment for the full cross-cluster restore workflow.

Keep your backup encryption keys safe

To restore from an encrypted backup, you **must have the original encryption key**. If the key is lost or rotated, your backups are irrecoverable. Always ensure you have a secure and reliable process for managing and backing up encryption keys separately from your database backups.

Restore from a backup

Restore the cluster from a previously saved backup

You can restore from a backup as follows:

- On the same Kubernetes cluster and in the same namespace where you made the backup
- On a new cluster deployed in a different Kubernetes-based environment.

This document focuses on the restore to the same cluster.

Restore scenarios

Select how you wish to restore:

- Without point-in-time recovery
- Make a point-in-time recovery

To restore from a backup, you create a Restore object using a special restore configuration file. The example of such file is deploy/backup/restore.yaml.

You can check available options in the restore options reference.

Before you begin

To restore from a backup on the same cluster and namespace, do the following:

1. Export the namespace as an environment variable. Replace the `<namespace>` placeholder with your value:

   ```
   export NAMESPACE = <namespace>
   ```

2. Make sure that the cluster is running. Use this command to check it:

   ```
   kubectl get ps <cluster-name> -n $NAMESPACE
   ```

3. List backups with the following command:

```
kubectl get ps-backup -n $NAMESPACE
```

Restore from a backup without point-in-time recovery

When the correct names for the backup and the cluster are known,configure the `PerconaServerMySQLRestore` Custom Resource. Specify the following keys:

- set `spec.clusterName` key to the name of the target cluster to restore the backup on
- set `spec.backupName` key to the name of your backup. This is the value from the output of the `kubectl get ps-backup` command.

Pass this information to the Operator

via the YAML manifest

1. Edit the `deploy/backup/restore.yaml` file:

```
apiVersion: ps.percona.com/v1
kind: PerconaServerMySQLRestore
metadata:
  name: restore1
spec:
  clusterName: ps-cluster1
  backupName: backup1
```

2. Start the restore process:

```
kubectl apply -f deploy/backup/restore.yaml -n $NAMESPACE
```

via the command line

Instead of storing restore settings in a separate file, you can pass them directly to the `kubectl apply` command as follows:

```
cat <<EOF | kubectl apply -n $NAMESPACE  -f-
apiVersion: "ps.percona.com/v1"
kind: "PerconaServerMySQLRestore"
metadata:
  name: "restore1"
spec:
  clusterName: "ps-cluster1"
  backupName: "backup1"
EOF
```

Restore with point-in-time recovery

Before performing a point-in-time recovery, ensure you have:

- Enabled binlog collection in your cluster configuration
- Identified a base backup to restore from
- Determined either the GTID set (for restoring up to a specific transaction) or the exact timestamp

(for restoring up to a specific time)

- Configured the backup storage appropriately

Also check the implementation specifics and known limitations

Example 1. Restore to a specific time

1. Edit the deploy/backup/restore.yaml and specify the following options:
   - `spec.clusterName` - the name of your cluster
   - `spec.backupName` - the name of a backup
   - configure point-in-time recovery in the `spec.pitr` subsection:
     - `type` - specify `time`
     - `date` - specify the timestamp to restore the database to

   Here's the example configuration:

   ```
   apiVersion: ps.percona.com/v1
   kind: PerconaServerMySQLRestore
   metadata:
     name: restore-pitr-date
   spec:
     clusterName: ps-cluster1
     backupName: backup1
     pitr:
       type: date
       date: "2026-03-20 09:15:00"
   ```

2. Apply the configuration:

   ```
   kubectl apply -f deploy/backup/restore.yaml -n <namespace>
   ```

Example 2. Restore to a specific transaction

1. Edit the deploy/backup/restore.yaml and specify the following options:
   - `spec.clusterName` - the name of your cluster
   - `spec.backupName` - the name of a backup
   - configure point-in-time recovery in the `spec.pitr` subsection:

- `type` - specify `gtid`
- `gtid` - specify the GTID set to restore the database to. It has the format `source_id:transaction_id`

Here's the example configuration:

```
apiVersion: ps.percona.com/v1
kind: PerconaServerMySQLRestore
metadata:
  name: restore-pitr-gtid
spec:
  clusterName: ps-cluster1
  backupName: backup1
  pitr:
    type: gtid
    gtid: "cc5e06e7-241e-11f1-a165-522d36bd0c5e:225"
```

2. Apply the configuration:

```
kubectl apply -f deploy/backup/restore.yaml -n <namespace>
```

View restore details

When you start the restore, the restore job is created. You can check the job details using these commands:

```
kubectl get job -n <namespace>
```

Sample output

```
xb-restore-restore2                                Running    0/1
0s
xb-restore-restore2                                Complete             1/1
25s        25s
```

You can check the restore progress with this command:

```
kubectl get ps-restore -n $NAMESPACE
```

Troubleshooting

If you face issues with restore, refer to our Restore troubleshooting guide for help.

Restore from a backup to a new Kubernetes-based environment

You can restore from a backup as follows:

- On the same Kubernetes cluster where you made the backup
- On a new cluster deployed in a different Kubernetes-based environment.

This document focuses on the restore on a new cluster deployed in a different Kubernetes environment.

Restore scenarios

Select how you wish to restore:

- Restore from a backup to a new Kubernetes-based environment
- Restore scenarios
- Preconditions
- Before you begin
- Restore from a backup without point-in-time recovery
- Restore with point-in-time recovery
- Restore from an encrypted backup
- View restore details
- Post-restore steps
- Troubleshooting

To restore from a backup, you create a Restore object using a special restore configuration file. The example of such file is deploy/backup/restore.yaml.

You can check available options in the restore options reference.

Preconditions

When restoring to a new Kubernetes-based environment, make sure it has a Secrets object with the same user passwords as in the source cluster.

If the backup was encrypted, the Operator requires the Secret with the same encryption key that was used for the backup. Reference this Secret in the restore resource under `spec.backupSource.storage.encryptionKeySecret`.

You can export the user Secret from the source cluster and create a Secrets object on the target one. Here's how to do it:

1. Find the secret name. Run this command on the **cluster where you made the backup**:

   ```
   kubectl get ps ps-cluster1 -n $NAMESPACE -o jsonpath='{.spec.secretsName}
   ```

   Expected output

   ```
   ps-cluster1-secrets
   ```

2. Export the secret to a file:

   ```
   kubectl get secrets -n $NAMESPACE ps-cluster1-secrets -o yaml > ps-cluster1-secrets.yaml
   ```

3. Remove the `annotations`, `labels`, `creationTimestamp`, `resourceVersion`, `selfLink`, `uid` and `namespace` metadata fields from the resulting file to make it ready for the source cluster. Use the following script to do it:

   ```
   yq eval 'del(.metadata.ownerReferences, .metadata.annotations, .metadata.labels, .metadata.creationTimestamp, .metadata.resourceVersion, .metadata.selfLink, .metadata.uid, .metadata.namespace)' ps-cluster1-secrets.yaml > ps-cluster1-secrets-target.yaml
   ```

Expected output

```
apiVersion: v1
data:
  heartbeat: MUBra1Z2IWRNVjNFe0lJM3o=
  monitor: b1k3IyxyWns8eGxVSmFfTiU=
  operator: XT8oYWVhQnd3Sk5FeVdbJg==
  orchestrator: ZF8yXy04PGNTTSl4Qk5ZdFU=
  replication: JFZxVUF4XjBvOFJoI0hPbjZS
  root: QGckTSQtMF9ZeUlYWzV3UWIp
  xtrabackup: REZtVGNbTW5fKUVMfTUqRSEo
kind: Secret
metadata:
  name: my-db-ps-db-secrets
type: Opaque
```

4. **On the target cluster**, create the Secrets object. Replace the `<namespace>` with your value:

```
kubectl apply -f ps-cluster1-secrets-target.yaml -n <namespace>
```

Before you begin

1. Export the namespace as an environment variable. Replace the `<namespace>` placeholder with your value:

```
export NAMESPACE = <namespace>
```

2. Make sure that the cluster is running. Use this command to check it:

```
kubectl get ps <cluster-name> -n $NAMESPACE
```

3. List backups on the source cluster with the following command:

```
kubectl get ps-backup -n $NAMESPACE
```

Restore from a backup without point-in-time recovery

Configure the `PerconaServerMySQLRestore` Custom Resource. Specify the following keys:

- set `spec.clusterName` key to the name of the target cluster to restore the backup on
- configure the `spec.backupSource` subsection to point to the cloud storage where the backup is stored. This subsection should include:
  - a destination key. Take it from the output of the `kubectl get ps-backup` command
  - the necessary storage configuration keys, just like in the `deploy/cr.yaml` file of the source cluster.

S3-compatible storage

```
apiVersion: ps.percona.com/v1
kind: PerconaServerMySQLRestore
metadata:
  name: restore1
spec:
  clusterName: ps-cluster1
  backupSource:
    destination: s3://S3-BUCKET-NAME/BACKUP-NAME
    storage:
      type: s3
      s3:
        bucket: S3-BUCKET-NAME
        credentialsSecret: ps-cluster1-s3-credentials
        region: us-west-2
        endpointUrl: https://URL-OF-THE-S3-COMPATIBLE-STORAGE
        prefix: <PREFIX-WHERE-BACKUP-IS-STORED>
        ...
```

Google Cloud Storage

```
apiVersion: ps.percona.com/v1
kind: PerconaServerMySQLRestore
metadata:
  name: restore1
spec:
  clusterName: ps-cluster1
  backupSource:
    destination: gs://BUCKET-NAME/BACKUP-NAME
    storage:
      gcs:
        bucket: operator-testing
        credentialsSecret: ps-cluster1-gcp-credentials
        prefix: <PREFIX-WHERE-BACKUP-IS-STORED>
      type: gcs
```

Start the restore:

```
kubectl apply -f deploy/backup/restore.yaml -n $NAMESPACE
```

Restore with point-in-time recovery

For point-in-time recovery to a new cluster, you need the following:

- a binlog storage configured on the target cluster with a different path for binlog collection than on the source cluster. You can reuse the storage settings from the source cluster but specify a different `prefix` value.
- a base backup that will be used for the restore
- the GTID set to restore up to a specific transaction, or a timestamp to restore up to a specific time
- the backup storage configured either on the target cluster or in the restore object
- the binlog server storage from the source cluster configured in the restore object. The Operator uses it to apply binlogs on top of the backup
- the Secret with the user credentials. Refer to the Preconditions for how to create it.

When restoring to a new cluster, the Operator starts a temporary Binlog Server Pod using the settings you defined in the restore object, uses this server to locate and fetch the required binlogs, and removes it when the restore completes. Copy the binlog storage settings from the source cluster's `spec.backup.pitr.binlogServer` configuration, including the `prefix` that points at the binlog folder.

Binlog storage currently supports only AWS S3 and S3-compatible services, even when the base backup is stored elsewhere. Read more in the Point-in-time recovery documentation.

Edit the deploy/backup/restore.yaml manifest.

1. Specify the following keys:
   - Set `spec.clusterName` key to the name of the target cluster to restore the backup on
   - Configure the `spec.backupSource` subsection to point to the cloud storage where the backup is stored. This subsection should include:
   - A destination key. Take it from the output of the `kubectl get ps-backup` command on the source cluster

- The necessary storage configuration keys, just like in the `deploy/cr.yaml` file of the source cluster. Make sure to set the `prefix` in the restore object to the exact same value used for the backup, so the Operator can find the correct backup location.
- Configure the `pitr` subsection:
- `type` - specify one of the following:
  - `date` - to restore up to a specific time
  - `gtid` - to restore up to a specific transaction
- For the `type=date` option, set the `date` key in the datetime format.
- For the `type=gtid` option, set the `gtid` to the GTID set to restore the database to. It has the format `source_id:transaction_id`
- `pitr.backupSource.binlogServer` - configure access to the binlog storage on the source cluster. Use the same settings as in the source cluster's `spec.backup.pitr.binlogServer`, including the `prefix` for the binlog folder.

Restore to a timestamp

S3-compatible storage

```
apiVersion: ps.percona.com/v1
kind: PerconaServerMySQLRestore
metadata:
  name: restore-timestamp
spec:
  clusterName: ps-cluster1
  backupSource:
    destination: s3://S3-BUCKET-NAME/BACKUP-NAME
    storage:
      s3:
        bucket: S3-BUCKET-NAME
        credentialsSecret: ps-cluster1-s3-credentials
        region: us-west-2
        endpointUrl: https://URL-OF-THE-S3-COMPATIBLE-STORAGE
        prefix: <PREFIX-WHERE-BACKUP-IS-STORED>
        ...
    type: s3
  pitr:
    backupSource:
      binlogServer:
        storage:
          s3:
            bucket: S3-BINLOG-BUCKET-NAME
```

```
            credentialsSecret: ps-cluster1-s3-credentials
            region: us-west-2
            endpointUrl: https://URL-OF-THE-S3-COMPATIBLE-STORAGE
#Optional for AWS S3
            prefix: binlogs
    type: date
    date: "2026-03-20 09:15:00"
```

Google Cloud Storage

```
apiVersion: ps.percona.com/v1
kind: PerconaServerMySQLRestore
metadata:
  name: restore-timestamp
spec:
  clusterName: ps-cluster1
  backupSource:
    destination: gs://BUCKET-NAME/BACKUP-NAME
    storage:
      gcs:
        bucket: operator-testing
        credentialsSecret: ps-cluster1-gcp-credentials
        prefix: <PREFIX-WHERE-BACKUP-IS-STORED>
      type: gcs
  pitr:
    backupSource:
      binlogServer:
        storage:
          s3:
            bucket: S3-BINLOG-BUCKET-NAME
            credentialsSecret: ps-cluster1-s3-credentials
            region: us-west-2
            endpointUrl: https://URL-OF-THE-S3-COMPATIBLE-STORAGE
#Optional for AWS S3
            prefix: binlogs
    type: date
    date: "2026-03-20 09:15:00"
```

Restore to a transaction

S3-compatible storage

```
apiVersion: ps.percona.com/v1
kind: PerconaServerMySQLRestore
metadata:
  name: restore-transaction
spec:
```

```
  clusterName: ps-cluster1
  backupSource:
    destination: s3://S3-BUCKET-NAME/BACKUP-NAME
    storage:
      s3:
        bucket: S3-BUCKET-NAME
        credentialsSecret: ps-cluster1-s3-credentials
        region: us-west-2
        endpointUrl: https://URL-OF-THE-S3-COMPATIBLE-STORAGE
        prefix: <PREFIX-WHERE-BACKUP-IS-STORED>
        ...
    type: s3
  pitr:
    backupSource:
      binlogServer:
        storage:
          s3:
            bucket: S3-BINLOG-BUCKET-NAME
            credentialsSecret: ps-cluster1-s3-credentials
            region: us-west-2
            endpointUrl: https://URL-OF-THE-S3-COMPATIBLE-STORAGE
            prefix: binlogs
    type: gtid
    gtid: "cc5e06e7-241e-11f1-a165-522d36bd0c5e:225"
```

Google Cloud Storage

```
apiVersion: ps.percona.com/v1
kind: PerconaServerMySQLRestore
metadata:
  name: restore-transaction
spec:
  clusterName: ps-cluster1
  backupSource:
    destination: gs://BUCKET-NAME/BACKUP-NAME
    storage:
      gcs:
        bucket: operator-testing
        credentialsSecret: ps-cluster1-gcp-credentials
        prefix: <PREFIX-WHERE-BACKUP-IS-STORED>
      type: gcs
  pitr:
    backupSource:
      binlogServer:
        storage:
          s3:
            bucket: S3-BINLOG-BUCKET-NAME
            credentialsSecret: ps-cluster1-s3-credentials
```

```
            region: us-west-2
            endpointUrl: https://URL-OF-THE-S3-COMPATIBLE-STORAGE
            prefix: binlogs
    type: gtid
    gtid: "cc5e06e7-241e-11f1-a165-522d36bd0c5e:225"
```

Start the restore:

```
kubectl apply -f deploy/backup/restore.yaml -n $NAMESPACE
```

Restore from an encrypted backup

Configure the `PerconaServerMySQLRestore` Custom Resource. Specify the following keys:

- set `spec.clusterName` key to the name of the target cluster to restore the backup on
- configure the `spec.backupSource` subsection to point to the cloud storage where the backup is stored. This subsection should include:
- a destination key. Take it from the output of the `kubectl get ps-backup` command on the source cluster
- the necessary storage configuration keys, just like in the `deploy/cr.yaml` file of the source cluster.
- `encryptionKeySecret` referencing the Secret with the encryption key. The encryption key must be the same that was used to encrypt the backup.
- If you need point-in-time recovery, define the `pitr` subsection in your restore configuration. Refer to the Restore with point-in-time recovery section for supported keys and detailed guidance.

Without point-in-time recovery

This configuration restores an encrypted backup **without** point-in-time recovery.

```
apiVersion: ps.percona.com/v1
kind: PerconaServerMySQLRestore
metadata:
  name: restore1
spec:
  clusterName: ps-cluster1
  backupSource:
    destination: s3://S3-BUCKET-NAME/BACKUP-NAME
    storage:
      encryptionKeySecret:
        name: my-encryption-key
        key: encryptionKey
      type: s3
      s3:
        bucket: S3-BUCKET-NAME
        credentialsSecret: ps-cluster1-s3-credentials
        region: us-west-2
```

With point-in-time recovery

This configuration shows how to restore an encrypted backup **up to a specific timestamp**.

See the Point-in-time recovery options section for all point-in-time recovery configuration possibilities.

```
apiVersion: ps.percona.com/v1
kind: PerconaServerMySQLRestore
metadata:
  name: restore1-pitr
spec:
  clusterName: ps-cluster1
  backupSource:
    destination: s3://S3-BUCKET-NAME/BACKUP-NAME
    storage:
      encryptionKeySecret:
        name: my-encryption-key
        key: encryptionKey
      type: s3
      s3:
        bucket: S3-BUCKET-NAME
        credentialsSecret: ps-cluster1-s3-credentials
        region: us-west-2
  pitr:
    backupSource:
      binlogServer:
        storage:
          s3:
            bucket: S3-BINLOG-BUCKET-NAME
            credentialsSecret: ps-cluster1-s3-credentials
            region: us-west-2
            endpointUrl: https://URL-OF-THE-S3-COMPATIBLE-STORAGE
            prefix: binlogs
    type: date
    date: "2026-05-01T15:30:00Z"
```

Start the restore:

```
kubectl apply -f deploy/backup/restore.yaml -n <namespace>
```

View restore details

When you start the restore, the restore job is created. You can check the job details using these commands:

```
kubectl get job
```

Sample output

```
xb-restore-restore2                                Running    0/1
0s
xb-restore-restore2                                Complete             1/1
25s        25s
```

You can check the restore progress with this command:

```
kubectl get ps-restore -n $NAMESPACE
```

Post-restore steps

- Enable point-in-time recovery to start binlog collection on the restored cluster:

```
spec:
  backup:
    pitr:
      enabled: true
```

- Make a fresh base backup to start a new timeline for subsequent restores.

Troubleshooting

If you face issues with restore, refer to our Restore troubleshooting guide for help.

Fine-tuning backup and restore operations

When you run Percona Server for MySQL on Kubernetes, the Operator uses Percona XtraBackup to handle all backup and restore tasks. This process involves a few key binaries: `xtrabackup`, `xbcloud`, and `xbstream`. The Operator sets sensible defaults for these tools, ensuring the smooth flow for backups and restores. However, you might want to customize their behavior for specific scenarios, such as performance optimization.

You can configure `xtrabackup`, `xbcloud`, and `xbstream` tools in two main ways:

- **Globally**, by setting options in the main `deploy/cr.yaml` Custom Resource manifest. These settings apply to all backups and restores within the cluster.
- **Individually**, by defining options in the specific `PerconaServerMySQLBackup` or `PerconaServerMySQLRestore` manifests for on-demand operations.

The Operator applies backup and restore settings in a defined order: any options set in a `PerconaServerMySQLBackup` or `PerconaServerMySQLRestore` manifest will override those set globally in `deploy/cr.yaml`. For example:

- If an option is present in both the global and individual manifest, the Operator uses the value from the specific backup or restore manifest.
- If you set `xtrabackup` arguments globally in `deploy/cr.yaml` and set `xbcloud` options for a particular `PerconaServerMySQLBackup`, only the `xbcloud` configuration from the `PerconaServerMySQLBackup` manifest is applied for that job.

This hierarchy gives you maximum flexibility: you can define consistent default settings for the entire cluster, but still tailor individual backup or restore operations as needed. This way you can optimize performance, troubleshoot, or customize specific scenarios without affecting the global configuration.

Configuring through the Custom Resource

To apply settings globally, modify the `containerOptions` subsection within the `backup.storages` section of your `deploy/cr.yaml` file. This is useful for defining a consistent baseline for all backup and restore jobs.

You can set custom command-line arguments and environment variables for the backup binaries.

```
spec:
  backup:
    storages:
      <storage-name>:
        containerOptions:
          args:
            xtrabackup: ["--compress"]
            xbcloud: ["--max-retries=5"]
            xbstream: ["--parallel=4"]
          env:
            - name: MY_CUSTOM_VAR
              value: "some-value"
```

- **`xtrabackup`**: This tool handles the core backup process, interacting directly with the database. You can pass arguments like `--compress` to override the default compression mechanism, or `--encrypt`, `--encrypt-threads`, and `--encrypt-chunk-size` to tune encrypted backups. Read more about xtrabackup options in The xtrabackup command-line options reference
- **`xbcloud`**: This binary is used for transferring backup data to and from cloud storage. An example configuration is to define the number of retries after the failure. Read more about available options in The xbcloud command-line options reference
- **`xbstream`**: This binary is for streaming and decompressing backup data during restore. Arguments here, like `--parallel`, can help accelerate the process. Read more about available options in The xbstream command-line options reference.

The settings you define here are used for both backup and restore jobs, although `xbstream` and `xbcloud` arguments are primarily used during the restore process.

Overriding configuration for Specific Jobs

For on-demand backups and restores, you can provide specific `containerOptions` for the `PerconaServerMySQLBackup` or `PerconaServerMySQLRestore` objects in the respective `deploy/backup/backup.yaml` and `deploy/backup/restore.yaml` custom resources. This is helpful when you need a one-off task with unique settings that you don't want to apply globally.

Example for an on-demand backup

To apply specific settings for a single backup job, edit the `deploy/backup/backup.yaml` manifest.

```
apiVersion: ps.percona.com/v1alpha1
kind: PerconaServerMySQLBackup
metadata:
  name: my-special-backup
spec:
  clusterName: ps-cluster1
  storageName: s3-us-west
  containerOptions:
    args:
      xtrabackup: ["--binlog-info=off"]
      xbcloud: ["--retries=10"]
    env:
      - name: CUSTOM_BACKUP_ENV
        value: "extra-value"
```

You can see here that only `xtrabackup` and `xbcloud` arguments are specified. The `xbstream` binary is not used for backups and its configuration will be ignored, even if you specify it.

Note that the settings you specify here are used, overriding any global configuration from the `deploy/cr.yaml` file.

Example for a Restore

Similarly, for a restore operation, you can define options in the `deploy/backup/restore.yaml` manifest.

```
apiVersion: ps.percona.com/v1alpha1
kind: PerconaServerMySQLRestore
metadata:
  name: my-special-restore
spec:
  clusterName: cluster1
  backupName: backup1
  containerOptions:
    args:
      xtrabackup: ["--apply-log-only"]
      xbcloud: ["--retries=5"]
      xbstream: ["--parallel=8"]
    env:
      - name: CUSTOM_RESTORE_ENV
        value: "restore-value"
```

When this restore job runs, it will use the `xtrabackup`, `xbcloud`, and `xbstream` arguments defined here, ignoring any settings in the main `cr.yaml`.

The Percona Operator for MySQL provides a flexible and powerful way to manage backup and restore operations using `xtrabackup`, `xbcloud`, and `xbstream`. By leveraging the layered configuration system, you can set general policies in `deploy/cr.yaml` while retaining the ability to use specialized options for individual jobs. This ensures your operations remain efficient, reliable, and tailored to your specific needs.

Implementation specifics

1. The `xbstream` settings apply only to restores.
2. If you pass storage-specific parameters (like `--s3-region`) directly to the `xtrabackup`, `xbcloud`, and `xbstream` binaries, they will be overridden by the configuration in the storage section of the Custom Resource.
3. If you define environment variables for a backup job, they are passed to both the `xtrabackup` and `xbcloud` processes running within the `xtrabackup` sidecar container. You can verify this by inspecting the process environment variables inside the container during a backup.
4. Environment variables for a restore job are set for the `xtrabackup` container in the restore Pod.
5. The `VERIFY_TLS` environment variable currently applies only to restore operations and is ignored for backups.
6. If the same binary is configured globally and individually for backups or restores, individual settings take precedence.

Delete unneeded backups

You can delete backups in the following ways:

- Configure the retention policy for full backups and have the Operator delete them according to this policy rules.
- Delete backups of any type manually. However, deleting incremental backups has rules. To learn more, see Deleting incremental backup chains.

How the Operator handles backup deletion

The `percona.com/delete-backup` Kubernetes finalizer defined within a `PerconaServerMySQLBackup` backup object controls its deletion lifecycle. This is done to prevent orphaned backup data.

How it works:

- To start a backup, the Operator creates a backup object. Scheduled backups include the `percona.com/delete-backup` finalizer in the `metadata.finalizers` section by default. For on-demand backups, you can choose to add or remove this finalizer during configuration. This finalizer controls whether the Operator cleans up the backup data from the storage before deletion.
- While the finalizer is present, Kubernetes does not fully remove the `PerconaServerMySQLBackup` object after a delete request. The resource stays until every finalizer is cleared.
- After you run `kubectl delete ps-backup …`, the Operator marks the specified backup object for deletion by setting the `metadata.deletionTimestamp` value. The Operator's backup controller sees this and, if `percona.com/delete-backup` finalizer is listed, deletes backup data in a remote storage.
- When cleanup finishes, the Operator removes the `percona.com/delete-backup` finalizer. Kubernetes then completes deletion of the backup resource.

You can force the deletion of a backup by manually removing the `percona.com/delete-backup` finalizer from the backup object. Be aware that doing this bypasses the Operator's cleanup steps and may leave backup data or resources orphaned. Only do this if you fully understand the consequences and are certain it is safe.

Configure retention for full scheduled backups

In the cluster Custom Resource, each entry under `backup.schedule` with the type `full` can include an optional `keep` field. This field specifies how many most recent successful backups of that schedule to keep. When the count exceeds the `keep` value, the Operator deletes older backup objects.

Setting	Behavior
`keep: N` (positive integer)	Keep the **N** newest backups for that schedule; older ones are removed automatically.
`keep: 0` or **omit** `keep`	**No automatic deletion** for that schedule. Backups accumulate until you delete them manually.

Configure schedules in the `deploy/cr.yaml` as described in Make scheduled backups.

This example configuration instructs the Operator to keep the last three full backups from a weekly job:

```
backup:
  schedule:
    - name: "weekly-full-backup"
      schedule: "0 2 * * 0"
      type: "full"
      keep: 3
      storageName: s3-us-west
```

See also `backup.schedule.keep` in the Custom Resource reference.

Full vs incremental backup schedules

- Retention policy applies to **scheduled full backups** in the usual way: the Operator keeps only the specified number of newest full backups for that schedule. Note that retention is subject to the limitations when multiple schedules share one storage.
- **For scheduled incremental backups**, `keep` is ignored. There is no automatic retention that trims old increments. You can delete **only the latest increment** or **the whole chain** by deleting the base full backup.

Deleting incremental backup chains

Incremental backups depend on a **chain**: one **full** backup plus **increments** on the same storage. See Incremental backups for how chains are built.

Retention works differently than for standalone full backups, if the `percona.com/delete-backup` finalizer exists in the backup object:

- **Deleting the base full backup** removes the **entire** incremental chain that derives from it (the Operator does not leave orphaned increments).
- You **cannot** delete an increment **in the middle** of a chain as that would break continuity. You may delete only:
- the **last** increment in the chain (the tip), or
- the **full** backup at the base (which removes the whole chain).

If you use scheduled incremental jobs, plan cleanup: either rotate by deleting the **latest** increment when appropriate, or delete the **base** full backup when you want to drop the whole chain and start fresh (after taking a new full backup, if you still need recovery points).

Delete a backup manually

1. List backups in the namespace:

```
kubectl get ps-backup -n <namespace>
```

2. When you know the name, delete the `PerconaServerMySQLBackup` object:

```
kubectl delete ps-backup <backup-name> -n <namespace>
```

The Operator reconciles storage and related objects for that backup. For **incremental** backups, follow the chain rules above: prefer deleting `ps-backup` objects that represent the last increment or the base full backup.

Monitoring

Monitor database with Percona Monitoring and Management (PMM)

In this section you will learn how to monitor Percona Server for MySQL cluster with Percona Monitoring and Management (PMM).

PMM is a client/server application. It includes the PMM Server and the number of PMM Clients running on each node with the database you wish to monitor.

A PMM Client collects server metrics, general system metrics, query analytics and sends it to the server As a user, you connect to the PMM Server to see database metrics on a number of dashboards.

PMM Server and PMM Client are installed separately.

Considerations

1. Starting with the version 0.10.0, the Operator supports only PMM 3.x versions. The support for PMM 2.x is dropped.
2. You must run the Operator version 0.10.0 and later to monitor your database with PMM 3.x. Check the Upgrade the Operator tutorial for the update steps.
3. To use PMM3, PMM Server version must be equal to or newer than the PMM Client.

Install PMM Server

You must have PMM Server up and running. You can run PMM Server as a *Docker container*, a *virtual appliance*, or on an *AWS instance*. Please refer to the official PMM documentation for the installation instructions.

For Kubernetes environment, we recommend to install PMM from the Helm chart.

Install PMM Client

PMM Client is installed as a sidecar container in the database Pods in your Kubernetes-based environment. To install PMM Client, do the following:

1. Authorize PMM Client within PMM Server.

    1. PMM3 uses Grafana service accounts to control access to PMM server components and resources. To authenticate in PMM server, you need a service account token. Use PMM documentation to generate a service account with the **Admin** role and token.

        The token must have the format `glsa_**********_9e35351b`.

        > **Warning**
        >
        > When you create a service account token, you can select its lifetime: it can be either a permanent token that never expires or the one with the expiration date. PMM server cannot rotate service account tokens after they expire. So you must take care of reconfiguring PMM Client in this case.

    2. Add the service account token to the `pmmservertoken` option in the users Secrets object. Use the following command and replace the `<my-token>` placeholder with your value:

        in Linux

        ```
        kubectl patch secret/ps-cluster1-secrets -p "$(echo -n '{"data":
        {"pmmservertoken":"'$(echo -n <my-token> | base64 --wrap=0)'"}}')"
        ```

        in macOS

        ```
        kubectl patch secret/ps-cluster1-secrets -p "$(echo -n '{"data":
        {"pmmservertoken":"'$(echo -n new_key | base64)'"}}')"
        ```

2. Update the `pmm` section in the deploy/cr.yaml file:

    - Set `pmm.enabled` = `true`.
    - Specify your PMM Server hostname or an IP address for the `pmm.serverHost` option. The PMM Server IP address should be resolvable and reachable from within your cluster.

    ```
    pmm:
      enabled: true
      image: percona/pmm-client:3.8.1
      serverHost: monitoring-service
    ```

3. Apply the changes:

```
kubectl apply -f deploy/cr.yaml -n <namespace>
```

This triggers the Operator to restart your cluster Pods.

4. Check that corresponding Pods are not in a cycle of stopping and restarting. This cycle occurs if there are errors on the previous steps:

```
kubectl get pods -n <namespace>
kubectl logs  <cluster-name>-mysql-0 -c pmm-client -n <namespace>
```

Check the metrics

Let's see how the collected data is visualized in PMM.

1. Log in to PMM Server.
2. Click MySQL **MySQL** from the left-hand navigation menu.
3. Select your cluster from the **Clusters** drop-down menu and the desired time range on the top of the page. You should see the metrics.
4. Click MySQL **MySQL** → Other dashboards to see the list of available dashboards that allow you to drill down to the metrics you are interested in.
5. Click **Explore** from the left-hand navigation menu. In the **Metric** drop-down, start typing `mysql` to see the list of available metrics.
6. To see the data for the selected metric, click **Run query**.

Check PMM Client health and status

A probe is a diagnostic mechanism in Kubernetes which helps determine whether a container is functioning correctly and whether it should continue to run, accept traffic, or be restarted.

PMM Client has the following probes:

- **Readiness probe** determines when a PMM Client is available and ready to accept traffic
- **Liveness probe** determines when to restart a PMM Client

To configure probes, use the `spec.pmm.readinessProbe` and `spec.pmm.livenessProbe` Custom Resource options.

Monitor Kubernetes

Monitoring the state of the database is crucial to timely identify and react to performance issues. Percona Monitoring and Management (PMM) solution enables you to do just that.

However, the database state also depends on the state of the Kubernetes cluster itself. Hence it's important to have metrics that can depict the state of the Kubernetes cluster.

This document describes how to set up monitoring of the Kubernetes cluster health. This setup has been tested with the PMM server as the centralized data storage and the Victoria Metrics Operator as the statistics collector, though the steps should also apply to other monitoring applications.

Considerations

In this setup we use Victoria Metrics kubernetes monitoring stack Helm chart. When customizing the chart's values, consider the following:

- Since we use the PMM server for monitoring, there is no need to store the data in Victoria Metrics Operator. Therefore, the Victoria Metrics Helm chart is installed with the `vmsingle.enabled` and `vmcluster.enabled` parameters set to `false` in this setup.
- The Prometheus node exporter is not installed by default since it requires privileged containers with the access to the host file system. If you need the metrics for Nodes, enable the Prometheus node exporter by setting the `prometheus-node-exporter.enabled` flag in the Victoria Metrics Helm chart to `true`.
- Check all the role-based access control (RBAC) rules of the `victoria-metrics-k8s-stack` chart and the dependencies chart, and modify them based on your requirements.

Pre-requisites

To set up monitoring of Kubernetes, you need the following:

1. PMM Server up and running. You can run PMM Server in Kubernetes, as a Docker image, a virtual appliance, or on an AWS instance. Please refer to the official PMM documentation for the installation instructions.
2. Helm

Procedure

Install the Victoria Metrics Kubernetes monitoring stack

Quick install

Use the quick install script and the set of flags that describe your PMM Server and Kubernetes environment to install Victoria Metrics Kubernetes monitoring stack with a single command. The script installs Victoria Metrics Kubernetes monitoring stack with default parameters.

Replace the following placeholders with your values:

- `PMM-SERVER-TOKEN` - The PMM Server service account token
- `PMM-SERVER-URL` - The URL to access the PMM Server
- `UNIQUE-K8s-CLUSTER-IDENTIFIER` - Identifier for the Kubernetes cluster. It can be the name you defined during the cluster creation.

  You should use a unique identifier for each Kubernetes cluster. The use of the same identifier for more than one Kubernetes cluster will result in the conflicts during the metrics collection.
- `NAMESPACE` - The namespace where the Victoria metrics Kubernetes stack will be installed. If you haven't created the namespace before, it will be created during the command execution.

  We recommend to use a separate namespace like `monitoring-system`.

Without Prometheus node-exporter

By default, Prometheus node exporter is not installed with this stack because it requires privileged access to the host system, which may not be desirable in some Kubernetes environments. Also, you may already have a solution in place to scrape and analyze hardware- and kernel-level metrics.

The following command installs the Victoria Metrics Kubernetes monitoring stack without Prometheus node exporter. To install it with the node-exporter, refer to the adjacent tab (“With Prometheus node-exporter”) for instructions.

```
curl -fsL https://raw.githubusercontent.com/Percona-Lab/k8s-
monitoring/refs/tags/v0.1.2/vm-operator-k8s-stack/quick-install.sh \
| bash -s -- \
  --api-key <PMM-SERVER-TOKEN> \
  --pmm-server-url <PMM-SERVER-URL> \
  --k8s-cluster-id <UNIQUE-K8s-CLUSTER-IDENTIFIER> \
  --namespace $NAMESPACE
```

With Prometheus node-exporter

To install Victoria Metrics Kubernetes monitoring stack with the metrics for Nodes, add the `--node-exporter-enabled` flag as follows:

```
curl -fsL https://raw.githubusercontent.com/Percona-Lab/k8s-monitoring/
refs/tags/v0.1.2/vm-operator-k8s-stack/quick-install.sh \
| bash -s -- \
  --api-key <PMM-SERVER-TOKEN> \
  --pmm-server-url <PMM-SERVER-URL> \
  --k8s-cluster-id <UNIQUE-K8s-CLUSTER-IDENTIFIER> \
  --namespace $NAMESPACE \
  --node-exporter-enabled
```

Install manually

You may need to customize the default parameters of the Victoria metrics Kubernetes stack.

- Since we use the PMM Server for monitoring, there is no need to store the data in Victoria Metrics Operator. Therefore, the Victoria Metrics Helm chart is installed with the `vmsingle.enabled` and `vmcluster.enabled` parameters set to `false` in this setup.

- Check all the role-based access control (RBAC) rules of the `victoria-metrics-k8s-stack` chart and the dependencies chart, and modify them based on your requirements.

Prepare your environment

1. Create the Namespace where you want to set up monitoring. The following command creates the Namespace `monitoring-system`. You can specify a different name. In the following steps, specify your namespace instead of the `<namespace>` placeholder.

   ```
   kubectl create namespace monitoring-system
   ```

2. Export the namespace as the environment variable to simplify further configuration

   ```
   export NAMESPACE=monitoring-system
   ```

Set up authentication in PMM Server

To access the PMM Server resources and perform actions on the server, configure authentication.

1. Encode the PMM server token with base64.

   in Linux

   ```
   echo -n <pmm-token> | base64 --wrap=0
   ```

   in macOS

   ```
   echo -n <pmm-token>| base64
   ```

2. Create the YAML file for the Kubernetes Secrets and specify the base64-encoded PMM server token value within. Change the `namespace` option value to your namespace. Let's name this file `pmm-api-vmoperator.yaml`.

   pmm-api-vmoperator.yaml

```
apiVersion: v1
data:
  api_key: <base-64-encoded-pmm-server-token>
kind: Secret
metadata:
  name: pmm-token-vmoperator
  namespace: monitoring-system
type: Opaque
```

3. Create the Secrets object using the YAML file you created previously.

```
kubectl apply -f pmm-api-vmoperator.yaml -n $NAMESPACE
```

4. Check that the secret is created. The following command checks the secret for the resource named `pmm-token-vmoperator` (as defined in the `metadata.name` option in the secrets file). If you defined another resource name, specify your value.

```
kubectl get secret pmm-token-vmoperator -n $NAMESPACE
```

Create a ConfigMap to mount for `kube-state-metrics`

The `kube-state-metrics` (KSM) is a simple service that listens to the Kubernetes API server and generates metrics about the state of various objects - Pods, Deployments, Services and Custom Resources.

To define what metrics the `kube-state-metrics` should capture, create the ConfigMap and mount it to a container.

Use the example `configmap.yaml` configuration file to create the ConfigMap.

```
kubectl apply -f https://raw.githubusercontent.com/Percona-Lab/k8s-monitoring/refs/tags/v0.1.2/vm-operator-k8s-stack/ksm-configmap.yaml -n $NAMESPACE
```

As a result, you have the `customresource-config-ksm` ConfigMap created.

Install the Victoria Metrics Kubernetes monitoring stack

1. Add the dependency repositories of victoria-metrics-k8s-stack chart.

```
helm repo add grafana https://grafana.github.io/helm-charts
helm repo add prometheus-community https://prometheus-
community.github.io/helm-charts
```

2. Add the Victoria Metrics Kubernetes monitoring stack repository.

```
helm repo add vm https://victoriametrics.github.io/helm-charts/
```

3. Update the repositories.

```
helm repo update
```

4. Install the Victoria Metrics Kubernetes monitoring stack Helm chart. You need to specify the following configuration:

   - the URL to access the PMM server in the `externalVM.write.url` option in the format `<PMM-SERVER-URL>/victoriametrics/api/v1/write`. The URL can contain either the IP address or the hostname of the PMM server.
   - the unique name or an ID of the Kubernetes cluster in the `vmagent.spec.externalLabels.k8s_cluster_id` option. Ensure to set different values if you are sending metrics from multiple Kubernetes clusters to the same PMM Server.
   - The Namespace must be the same as the Namespace for the Secret and ConfigMap

```
helm install vm-k8s vm/victoria-metrics-k8s-stack \
-f https://raw.githubusercontent.com/Percona-Lab/k8s-
monitoring/refs/tags/v0.1.2/vm-operator-k8s-stack/values.yaml \
--set externalVM.write.url=<PMM-SERVER-URL>/victoriametrics/api/v1/write \
--set vmagent.spec.externalLabels.k8s_cluster_id=<UNIQUE-CLUSTER-
IDENTIFIER/NAME> \
-n $NAMESPACE
```

   To illustrate, say your PMM Server URL is `https://pmm-example.com`, the cluster ID is `test-cluster` and the Namespace is `monitoring-system`. Then the command would look like this:

```
helm install vm-k8s vm/victoria-metrics-k8s-stack \
-f https://raw.githubusercontent.com/Percona-Lab/k8s-
monitoring/refs/tags/v0.1.2/vm-operator-k8s-stack/values.yaml \
--set externalVM.write.url=https://pmm-
example.com/victoriametrics/api/v1/write \
--set vmagent.spec.externalLabels.k8s_cluster_id=test-cluster \
-n monitoring-system
```

Validate the successful installation

Check the Pods.

```
kubectl get pods -n $NAMESPACE
```

What Pods are running depends on the configuration chosen in values used while installing `victoria-metrics-k8s-stack` chart.

Without Prometheus node-exporter

Sample output

```
NAME                                                   READY   STATUS    RESTARTS
AGE
vm-k8s-grafana-5f6bdb8c7c-d5bw5                        3/3     Running   0
90m
vm-k8s-kube-state-metrics-57c5977d4f-6jtbj             1/1     Running   0
81m
vm-k8s-victoria-metrics-operator-6b7f4f786d-sctp8      1/1     Running   0
90m
vmagent-vm-k8s-victoria-metrics-k8s-stack-fbc86c9db-rz8wk   2/2     Running   0
90m
```

With Prometheus node-exporter

Sample output

```
NAME                                                   READY   STATUS    RESTARTS
AGE
vm-k8s-grafana-5f6bdb8c7c-d5bw5                        3/3     Running   0
90m
vm-k8s-kube-state-metrics-57c5977d4f-6jtbj             1/1     Running   0
81m
vm-k8s-prometheus-node-exporter-kntfk                  1/1     Running   0
90m
vm-k8s-prometheus-node-exporter-mjrvj                  1/1     Running   0
90m
vm-k8s-prometheus-node-exporter-v98c8                  1/1     Running   0
90m
vm-k8s-victoria-metrics-operator-6b7f4f786d-sctp8      1/1     Running   0
90m
vmagent-vm-k8s-victoria-metrics-k8s-stack-fbc86c9db-rz8wk   2/2     Running   0
90m
```

Verify metrics capture

The Victoria Metrics kubernetes chart captures the following metrics:

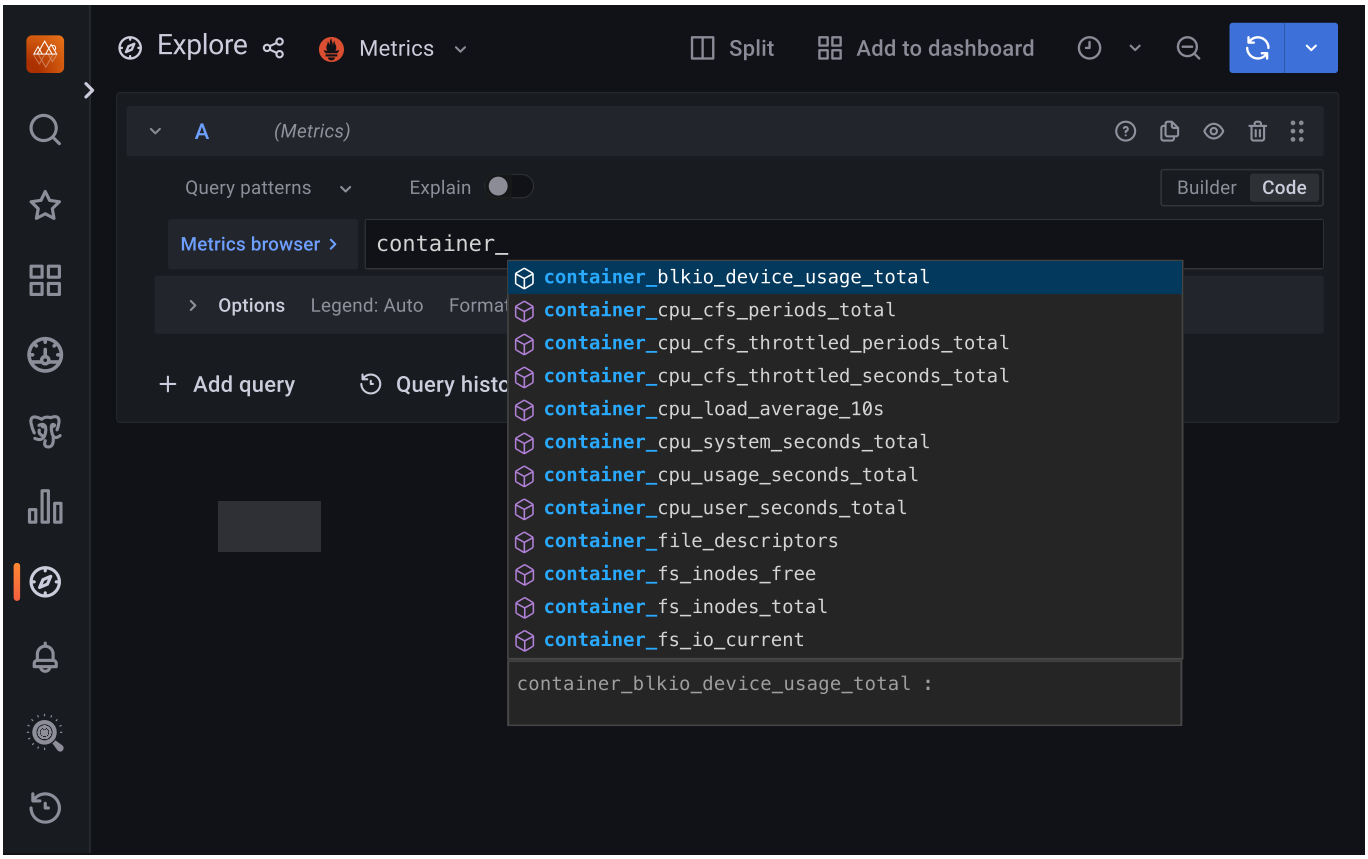
1. **cAdvisor (Container Advisor):** Embedded in the kubelet on each node. It observes resource usage and performance characteristics of running containers.
2. **kubelet:** The node agent that runs pods, interacts with the container runtime, mounts volumes,

and reports node and pod status to the control plane.

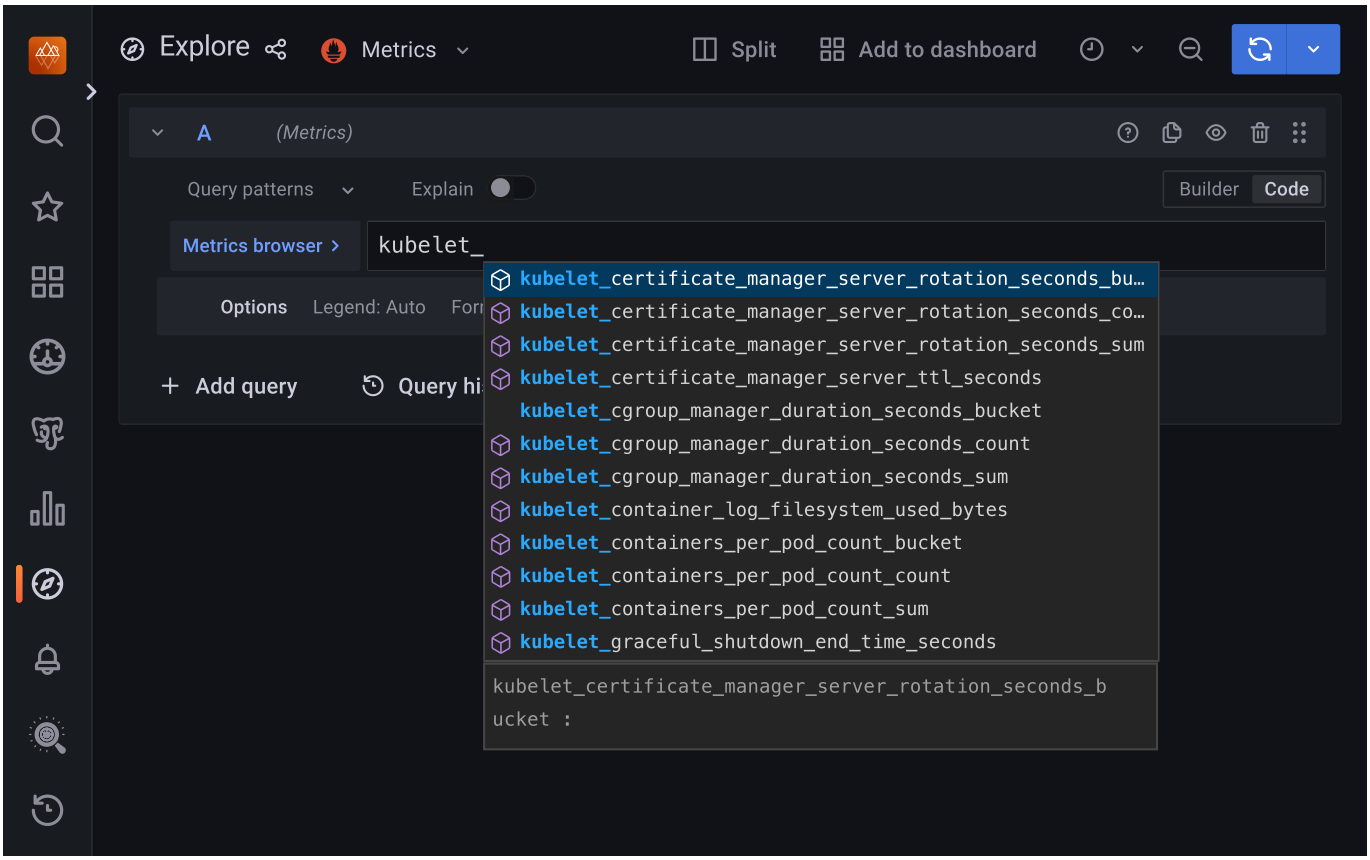
3. **CoreDNS:** The cluster DNS server that resolves service names and external names for pods.
4. **kube-state-metrics (KSM):** Listens to the Kubernetes API and emits metrics about object state (Deployments, Pods, PVCs, etc.), not low-level container performance.
5. **Node exporter:** A daemon that exposes host-level hardware and OS metrics (CPU, memory, disk, filesystem, network stack statistics) from /proc and /sys.
6. **Kubernetes API server (kube-apiserver):** Exposes metrics about requests to the API, admission, authentication, etcd interactions (from the API server's perspective), and control-plane load.
7. **Control plane components:** When running as pods: `kube-controller-manager`, `etcd`, `kube-scheduler`. `
   - `kube-controller-manager`: Runs controllers that reconcile desired state (ReplicaSet, endpoints, service accounts, garbage collection, etc.).
   - `kube-scheduler`: Assigns pods to nodes based on filters and scoring.
   - `etcd`: The consistent key-value store for all Kubernetes state.

Here's how to check the metrics:

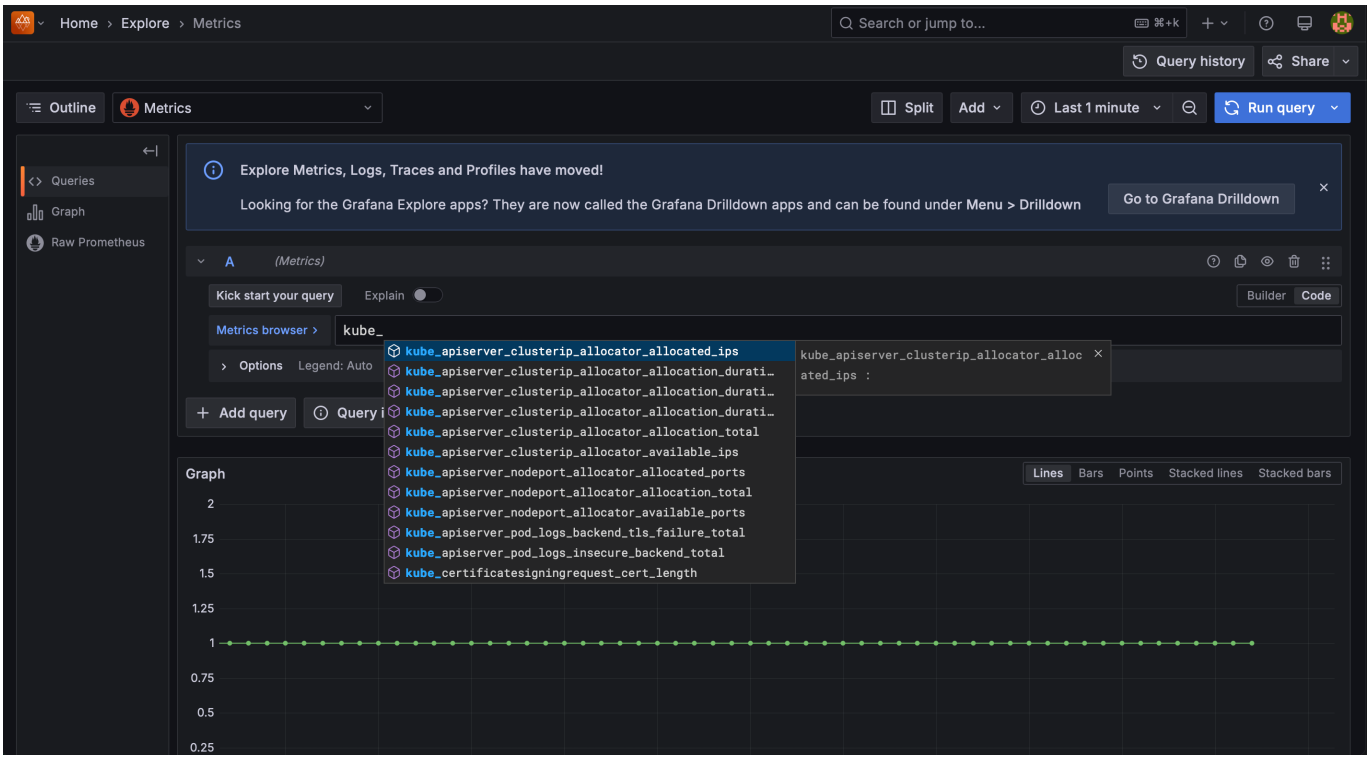
1. Connect to the PMM server.
2. Click **Explore** and switch to the **Code** mode.
3. Check that the required metrics are captured, type the following in the Metrics browser dropdown:
   - cadvisor :



- `kubelet`:



- kube-state-metrics metrics that also include Custom resource metrics for the Operator and database deployed in your Kubernetes cluster:



Uninstall Victoria metrics Kubernetes stack

To remove Victoria metrics Kubernetes stack used for Kubernetes cluster monitoring, use the cleanup script. By default, the script removes all the Custom Resource Definitions(CRD) and Secrets associated with the Victoria metrics Kubernetes stack. To keep the CRDs, run the script with the `--keep-crd` flag.

Remove CRDs

Replace the `<NAMESPACE>` placeholder with the namespace you specified during the Victoria metrics Kubernetes stack installation:

```
bash <(curl -fsL https://raw.githubusercontent.com/Percona-Lab/k8s-monitoring/refs/tags/v0.1.2/vm-operator-k8s-stack/cleanup.sh) --namespace <NAMESPACE>
```

Keep CRDs

Replace the `<NAMESPACE>` placeholder with the namespace you specified during the Victoria metrics Kubernetes stack installation:

```
bash <(curl -fsL https://raw.githubusercontent.com/Percona-Lab/k8s-monitoring/refs/tags/v0.1.2/vm-operator-k8s-stack/cleanup.sh) --namespace <NAMESPACE> --keep-crd
```

Check that the Victoria metrics Kubernetes stack is deleted:

```
helm list -n $NAMESPACE
```

The output should provide the empty list.

If you face any issues with the removal, uninstall the stack manually:

```
helm uninstall vm-k8s-stack -n $NAMESPACE
```

Scale Percona Distribution for MySQL on Kubernetes

One of the great advantages brought by Kubernetes platform is the ease of an application scaling. Scaling an application results in adding or removing the Pods and scheduling them to available Kubernetes nodes.

Scaling can be vertical and horizontal. Vertical scaling adds more compute or storage resources to MySQL nodes; horizontal scaling is about adding more nodes to the cluster. High availability looks technically similar, because it also involves additional nodes, but the reason is maintaining liveness of the system in case of server or network failures.

Vertical scaling

Scale compute resources

The Operator deploys and manages multiple components, such as Percona Server for MySQL, Orchestrator, HAProxy or MySQL Router and others. You can manage CPU or memory for every component separately by editing corresponding sections in the Custom Resource. We follow the structure for `requests` and `limits` that Kubernetes provides.

For example, you can add more resources to your HAProxy nodes by editing the following section in the Custom Resource:

```
spec:
  ...
  proxy:
    haproxy:
      ...
      resources:
        requests:
          memory: 4G
          cpu: 2
        limits:
          memory: 4G
          cpu: 2
```

Use our reference documentation for the Custom Resource options for more details about other components.

Scale storage

Kubernetes manages storage with the following components:

- a PersistentVolume (PV) - a segment of storage supplied by the Kubernetes administrator,
- a PersistentVolumeClaim (PVC) - a request for storage from a user.

Starting with Kubernetes v1.11, you can increase the size of an existing PVC object (considered stable since Kubernetes v1.24). Note that you **cannot shrink** the size of an existing PVC object.

Use storage scaling to keep up with growing data while keeping the cluster online. The Operator supports the following scaling options:

- automatic scaling - Starting with Operator version 1.2.0, the Operator monitors storage usage and scales the storage automatically
- storage resizing with Volume Expansion capability - Starting with the Operator version 0.11.0, you can instruct the Operator to scale the storage by updating the Custom Resource manifest
- manual scaling - scale the storage manually.

You can also use an external autoscaler with the Operator. Enabling an external autoscaler disables the Operator's internal logic for automatic storage resizing. Choose one method based on your environment and requirements; using both simultaneously is not supported.

For either option, the volume type must support PVCs expansion.

Check expansion capability for your volume type

Certain volume types support PVCs expansion by default. You can run the following command to check if your storage supports the expansion capability:

```
kubectl describe sc <storage class name> | grep AllowVolumeExpansion
```

Expected output

```
AllowVolumeExpansion: true
```

Find exact details about PVCs and the supported volume types in Kubernetes documentation.

Automatic storage resizing

Starting with version 1.2.0, the Operator can automatically resize Persistent Volume Claims (PVCs) for Percona Server for MySQL Pods based on your configured thresholds. The Operator monitors storage usage of MySQL PVCs. When the usage exceeds the defined threshold, the Operator triggers resizing until the storage size reaches the maximum limit.

This feature gives you:

- fewer outages from full disks because storage grows with demand
- less guesswork on capacity planning and fewer last-minute fixes
- lower operational effort for developers and platform engineers
- cost control by expanding only when needed
- a more predictable environment so teams can focus on delivery

Important

Keep in mind that some storage providers may have specific limitations when it comes to expanding storage. Review your provider's documentation and sett thresholds that work well for your environment to get the best results.

To enable automatic storage resizing, edit the `deploy/cr.yaml` Custom Resource manifest as follows:

1. Make sure the MySQL component has a storage size set.

   Example:

   ```
   spec:
     mysql:
       volumeSpec:
         persistentVolumeClaim:
           resources:
             requests:
               storage: 6Gi
   ```

2. Configure autoscaling thresholds in the `storageScaling` subsection:

   - `enableVolumeScaling` - set to `true`.
   - `autoscaling.enabled` - set to `true`
   - `autoscaling.triggerThresholdPercent` - specify the usage percentage (50–95, default

`80`). When usage exceeds this threshold, autoscaling is triggered

- `autoscaling.growthStep` - specify how much storage to add on each resize (default `2Gi`)
- `autoscaling.maxSize` - specify the upper limit for storage growth (minimum `1Gi`). When this limit is reached, scaling stops

Example configuration:

```
spec:
  storageScaling:
    enableVolumeScaling: true
    autoscaling:
      enabled: true
      triggerThresholdPercent: 80
      growthStep: 2Gi
      maxSize: 10Gi
```

3. Apply the configuration:

```
kubectl apply -f deploy/cr.yaml -n <namespace>
```

When the Operator changes the storage size, it updates the Custom Resource status as follows:

- adds the `pvc-resize-in-progress` annotation. The annotation contains the timestamp of the resize start and indicates that the resize operation is running. After the resize finishes, the Operator deletes this annotation
- records the new size in the `currentSize` field
- updates the `resizeCount` field
- records any errors in the `lastError` field
- Check the current cluster state:

```
kubectl get ps <cluster-name> -o yaml -n <namespace>
```

Sample output

```
storageAutoscaling:
  datadir-ps-cluster1-mysql-0:
    currentSize: 4194304Ki
    lastResizeTime: "2026-01-23T15:08:59Z"
    resizeCount: 2
```

The `storageAutoscaling` section appears under the `.status` in the Custom Resource.

When the storage size reaches the limit, no further resizing is done and this event is recorded in the logs. You can either clean up the data or set a new limit based on your organization's policies and requirements. For help with common issues, see Operator logs.

Storage resizing with Volume Expansion capability

In this document we're using the default Percona Server for MySQL cluster name `ps-cluster1`. If you have a different name, replace `ps-cluster1` with it in the commands.

To enable storage resizing via volume expansion, do the following:

1. Set the `storageScaling.enableVolumeScaling` Custom Resource option to `true` (it is turned off by default). When enabled, the Operator automatically expands storage when you increase the size in the Custom Resource
2. Change the `mysql.volumeSpec.persistentVolumeClaim.resources.requests.storage` option in the `deploy/cr.yaml` file to the desired storage size.

   Here's the example configuration:

   ```
   spec:
     ...
     storageScaling:
       enableVolumeScaling: true
     mysql:
       ...
       volumeSpec:
         ...
         persistentVolumeClaim:
           resources:
             requests:
               storage: <NEW STORAGE SIZE>
   ```

3. Apply the new configuration:

```
kubectl apply -f cr.yaml
```

The storage size change takes some time. When it starts, the Operator automatically adds the `pvc-resize-in-progress` annotation to the `PerconaServerMySQL` Custom Resource. The annotation contains the timestamp of the resize start and indicates that the resize operation is running. After the resize finishes, the Operator deletes this annotation.

Manual resizing

To increase the storage size manually, do the following:

1. Extract and backup the cluster configuration

   ```
   kubectl get ps ps-cluster1 -o yaml > CR_backup.yaml
   ```

2. Now you should delete the cluster.

   You can use the following command to delete the cluster:

   ```
   kubectl delete ps ps-cluster1
   ```

3. For each node, edit the yaml to resize the PVC object.

   ```
   kubectl edit pvc datadir-ps-cluster1-mysql-0
   ```

   In the yaml, edit the spec.resources.requests.storage value.

   ```
   spec:
     accessModes:
     - ReadWriteOnce
     resources:
       requests:
         storage: 6Gi
   ```

   Perform the same operation on the other nodes.

   ```
   kubectl edit pvc datadir-ps-cluster1-mysql-1
   kubectl edit pvc datadir-ps-cluster1-mysql-2
   ```

4. In the CR configuration file, use vim or another text editor to edit the PVC size.

```
vim CR_backup.yaml
```

5. Set the new storage size for the `mysql.volumeSpec.persistentVolumeClaim.resources.requests.storage` option:

```
mysql:
volumeSpec:
  persistentVolumeClaim:
    resources:
      requests:
        storage: 6Gi
```

The size of the storage must match the size you defined for the PVC object on the cluster nodes.

6. Apply the updated configuration to the cluster.

```
kubectl apply -f CR_backup.yaml
```

The storage size change takes some time. When it starts, the Operator automatically adds the `pvc-resize-in-progress` annotation to the `PerconaServerMySQL` Custom Resource. The annotation contains the timestamp of the resize start and indicates that the resize operation is running. After the resize finishes, the Operator deletes this annotation.

Horizontal scaling

Size of the cluster is controlled by a size key in the Custom Resource options configuration. That's why scaling the cluster needs nothing more but changing this option and applying the updated configuration file. This may be done in a specifically saved config, or on the fly, using the following command:

```
kubectl patch ps ps-cluster1 --type='json' -p='[{"op": "replace", "path": "/spec/mysql/size", "value": 5 }]'
```

In this example we have changed the size of the Percona Server for MySQL Cluster to `5` instances.

Production configuration

TLS/SSL encryption

Transport Layer Security (TLS)

The Percona Operator for MySQL uses Transport Layer Security (TLS) cryptographic protocol for the communication between the client application and the cluster.

You can configure TLS security in several ways.

- By default, the Operator **generates long-term certificates** automatically during the cluster creation if there are no certificate secrets available. The Operator's self-signed issuer is local to the Operator Namespace. This self-signed issuer is created because Percona Distribution for MySQL requires all certificates issued by the same source.
- The Operator can use a *cert-manager*, which will automatically **generate and renew short-term TLS certificates**. You must explicitly install cert-manager for this scenario.

  The *cert-manager* acts as a self-signed issuer and generates certificates allowing you to deploy and use the Percona Operator without a separate certificate issuer.
- You can generate TLS certificates manually or obtain them from some other issuer and provide to the Operator.

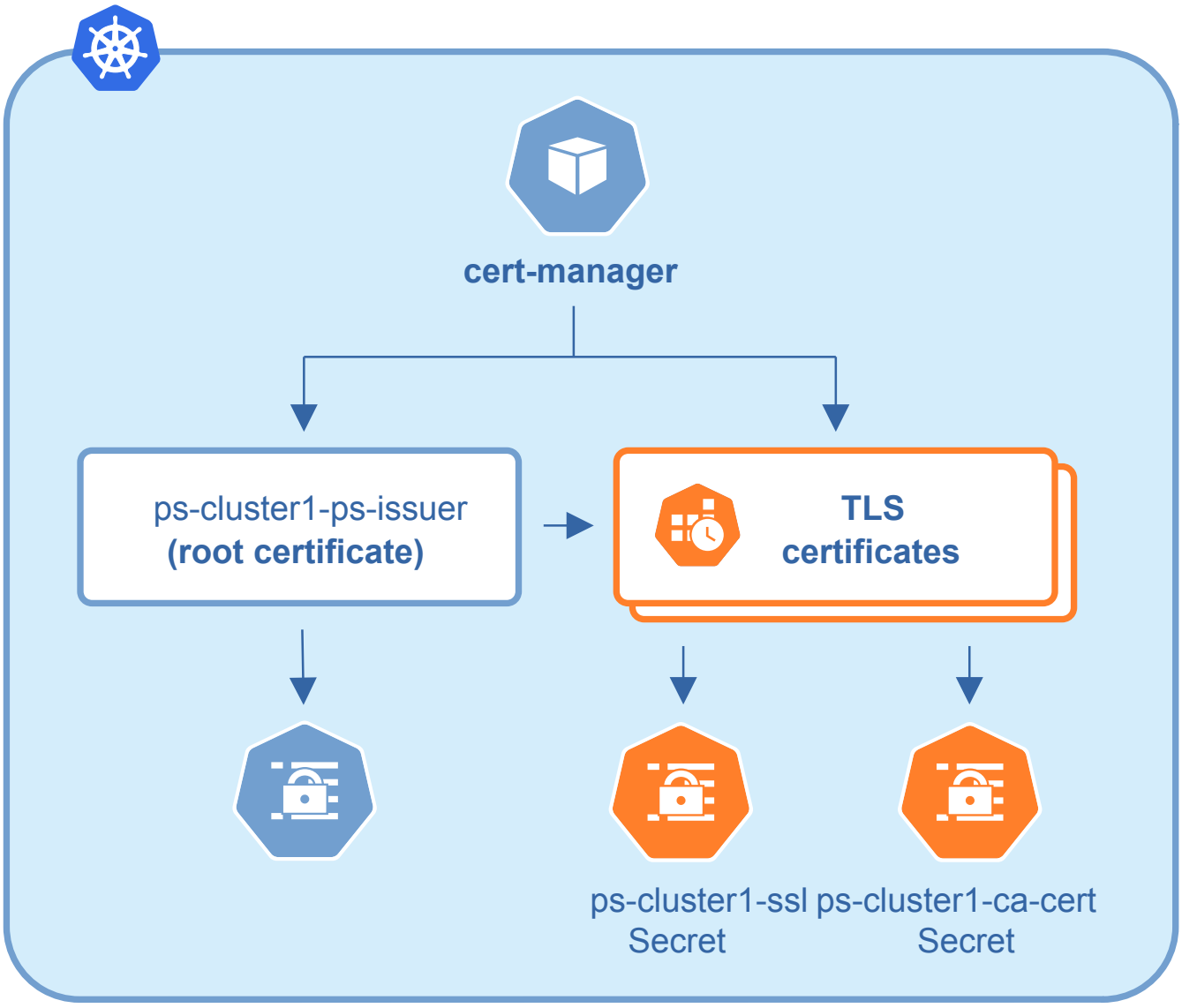
Configure TLS/SSL encryption using cert-manager

About the *cert-manager*

A cert-manager is a Kubernetes certificate management controller which is widely used to automate the management and issuance of TLS certificates. It is community-driven, and open source.

When you have already installed *cert-manager*, nothing else is needed: just deploy the Operator, and the Operator will request a certificate from the *cert-manager*.

The cert-manager generates short-term certificates valid for 3 months.



Install *cert-manager*

The cert-manager requires its own namespace

The steps to install the *cert-manager* are the following:

1. Create a namespace:

```
$ kubectl create namespace cert-manager
```

2. Disable resource validations on the cert-manager namespace:

```
$ kubectl label namespace cert-manager certmanager.k8s.io/disable-validation=true
```

3. Install the cert-manager:

```
$ kubectl apply -f https://github.com/jetstack/cert-manager/releases/download/v1.20.2/cert-manager.yaml
```

4. Verify the *cert-manager* by running the following command:

```
$ kubectl get pods -n cert-manager
```

   The result should display the *cert-manager* and webhook active and running:

Expected output

```
NAME                                       READY   STATUS    RESTARTS   AGE
cert-manager-69f748766f-6chvt              1/1     Running   0          65s
cert-manager-cainjector-7cf6557c49-l2cwt   1/1     Running   0          66s
cert-manager-webhook-58f4cff74d-th4pp      1/1     Running   0          65s
```

Once you create the database with the Operator, it will automatically trigger the cert-manager to create certificates. Whenever you check certificates for expiration, you will find that they are valid and short-term.

Generate certificates manually

You can generate TLS certificates manually instead of using the Operator's automatic certificate generation. This approach gives you full control over certificate properties and is useful for production environments with specific security requirements.

What you'll create

When you follow the steps from this guide, you'll generate these certificate files:

- `server.pem` - Server certificate for MySQL nodes
- `server-key.pem` - Private key for the server certificate
- `ca.pem` - Certificate Authority certificate
- `ca-key.pem` - Certificate Authority private key

Next, create the server TLS certificates using the CA keys, certs, and server details and then reference this Secret in the Custom Resource.

Prerequisites

Before you start, make sure you have:

- `cfssl` and `cfssljson` tools installed on your system
- Your cluster name and namespace ready
- Access to your Kubernetes cluster

Generate certificates

1. Replace `ps-cluster1` and `my-namespace` with your actual cluster name and namespace in the commands below:

   ```
   CLUSTER_NAME=ps-cluster1
   NAMESPACE=my-namespace
   ```

2. Generate a Certificate Authority (CA). You will use it to sign your server certificates.

```
cat <<EOF | cfssl gencert -initca - | cfssljson -bare ca
{
  "CN": "Root CA",
  "key": {
    "algo": "rsa",
    "size": 2048
  }
}
EOF
```

The output is two files: `ca.pem` (the CA certificate) and `ca-key.pem` (the CA private key).

3. Generate the Server Certificate using the CA. This command generates a server certificate and key, signed by your newly-created CA. The certificate will be valid for all hosts required by your cluster components.

```
cat <<EOF | cfssl gencert -ca=ca.pem  -ca-key=ca-key.pem - | cfssljson -
bare server
{
  "hosts": [
    "*.${CLUSTER_NAME}-mysql",
    "*.${CLUSTER_NAME}-mysql.${NAMESPACE}",
    "*.${CLUSTER_NAME}-mysql.${NAMESPACE}.svc",
    "*.${CLUSTER_NAME}-orchestrator",
    "*.${CLUSTER_NAME}-orchestrator.${NAMESPACE}",
    "*.${CLUSTER_NAME}-orchestrator.${NAMESPACE}.svc",
    "*.${CLUSTER_NAME}-router",
    "*.${CLUSTER_NAME}-router.${NAMESPACE}",
    "*.${CLUSTER_NAME}-router.${NAMESPACE}.svc"
  ],
  "CN": "${CLUSTER_NAME}-mysql",
  "key": {
    "algo": "rsa",
    "size": 2048
  }
}
EOF
```

The outputs are `server.pem` (the server certificate) and `server-key.pem` (the server private key).

Create the Kubernetes Secret from the generated certificates

This command packages the generated certificate and key files into a Kubernetes secret named `my-cluster-ssl` in your chosen namespace.

```
kubectl create secret generic my-cluster-ssl -n $NAMESPACE \
  --from-file=tls.crt=server.pem \
  --from-file=tls.key=server-key.pem \
  --from-file=ca.crt=ca.pem \
  --type=kubernetes.io/tls
```

Configure your cluster

After creating the Secret, reference it in your cluster configuration in the deploy/cr.yaml.

```
spec:
  sslSecretName: my-cluster-ssl
```

Apply the configuration to update the cluster:

```
kubectl apply -f deploy/cr.yaml -n $NAMESPACE
```

This triggers your Pods to restart.

Update certificates

How your TLS certificates are updated depends on how they were created:

- Certificates generated by the Operator are long-term. If you need to rotate them, you must do it manually.
- Certificates issued by the cert-manager are short-term. They are valid for 3 months. The cert-manager automatically reissues the certificates on schedule and without downtime.
- Certificates manually generated by you are not renewed automatically. It is your responsibility to timely update them. Use the steps in the following sections for how to do it.

Check your certificates for expiration

If you use cert-manager:

1. Check the necessary secrets names (`ps-cluster1-ssl` and `ps-cluster1-ca-cert` by default):

   ```
   kubectl get certificate -n $NAMESPACE
   ```

   Sample output

   ```
   ps-cluster1-ca-cert   True    ps-cluster1-ca-cert   45m
   ps-cluster1-ssl       True    ps-cluster1-ssl       43m
   ```

2. Optionally you can also check that the certificates issuer is up and running:

   ```
   kubectl get issuer -n $NAMESPACE
   ```

   The response should be as follows:

   ```
   NAME                              READY   AGE
   ps-cluster1-ps-ca-issuer   True    40m
   ps-cluster1-ps-issuer      True    38m
   ```

Note

If you don't use cert-manager, list your secrets:

```
kubectl get secrets -n $NAMESPACE
```

Then either use the default ones or the one you created

3. Use the following command to find out the certificates validity dates, substituting Secrets names if necessary:

```
{
kubectl get secret/ps-cluster1-ca-cert -n ps -o
jsonpath='{.data.tls\.crt}' | base64 --decode | openssl x509 -noout -dates
kubectl get secret/ps-cluster1-ssl -o jsonpath='{.data.ca\.crt}' | base64
--decode | openssl x509 -noout -dates
}
```

Sample output

```
notBefore=Nov  7 10:54:00 2025 GMT
notAfter=Nov  7 10:54:00 2026 GMT
```

Update certificates without downtime

If you don't use cert-manager and have *created certificates manually*, you can follow the next steps to perform a no-downtime update of these certificates *if they are still valid*.

Note

For already expired certificates, follow the alternative way.

Having non-expired certificates, you can roll out new certificates (both CA and TLS) with the Operator as follows:

1. Generate a new CA certificate (`ca.pem`), a new TLS certificate (server.pem) and a key for it (server-key.pem).

2. Get the current CA (`ca.pem.old`) and TLS (`tls.pem.old`) certificates and the TLS certificate key (`tls.key.old`):

```
kubectl get secret/ps-cluster1-ssl -o jsonpath='{.data.ca\.crt}' | base64
--decode > ca.pem.old
kubectl get secret/ps-cluster1-ssl -o jsonpath='{.data.tls\.crt}' | base64
--decode > tls.pem.old
kubectl get secret/ps-cluster1-ssl -o jsonpath='{.data.tls\.key}' | base64
--decode > tls.key.old
```

3. Combine new and current `ca.pem` into a `ca.pem.combined` file:

```
cat ca.pem ca.pem.old >> ca.pem.combined
```

4. Create a new Secrets object with the *old* TLS certificate (`tls.pem.old`) and key (`tls.key.old`), but a *new combined* `ca.pem` (`ca.pem.combined`):

```
kubectl create secret generic ps-cluster1-ssl \
--from-file=tls.crt=server.pem.old \
--from-file=tls.key=server-key.pem.old \
--from-file=ca.crt=ca.pem.combined \
--type=kubernetes.io/tls -o yaml --dry-run=client | kubectl apply -f -
```

5. The cluster will go through a rolling restart. This process will not cause issues, because every node has the old TLS certificate/key, and both new and old CA certificates.
6. Create a new Secrets object again. This time use a new TLS certificate (`server.pem` in the example) and a new TLS key (`server-key.pem`), and again the combined CA certificate (`ca.pem.combined`):

```
kubectl create secret generic ps-cluster1-ssl \
--from-file=tls.crt=server.pem \
--from-file=tls.key=server-key.pem \
--from-file=ca.crt=ca.pem.combined \
--type=kubernetes.io/tls -o yaml --dry-run=client | kubectl apply -f -
```

7. The cluster will go through a rolling restart. This process will not cause issues, because every node already has a new CA certificate (as a part of the combined CA certificate), and can successfully allow joiners with new TLS certificate to join. A joiner node also has a combined CA certificate, so it can authenticate against older TLS certificate.
8. Create a final Secrets object: use the new TLS certificate (`server.pmm`) and its key (`server-key.pem`), and only the new CA certificate (`ca.pem`):

```
kubectl create secret generic ps-cluster1-ssl \
--from-file=tls.crt=server.pem \
--from-file=tls.key=server-key.pem \
--from-file=ca.crt=ca.pem \
--type=kubernetes.io/tls -o yaml --dry-run=client | kubectl apply -f -
```

9. The cluster will go through a rolling restart, but it will do it without problems: the old CA certificate is removed, and every node is already using new TLS certificate and no nodes rely on the old CA certificate any more.

Data-at-rest encryption

Data-at-rest encryption

Data-at-rest encryption ensures that data stored on disk remains protected even if the underlying storage is compromised. This process is transparent to your applications, meaning you don't need to change your application code. If an unauthorized user gains access to the storage, they can't read the data files.

Percona Operator for MySQL uses the `keyring_vault` plugin, shipped with Percona Server for MySQL, to encrypt tablespaces, backups and binlogs. It also uses HashiCorp Vault to securely store and manage master encryption keys, enabling automatic key rotation, audit logging, and compliance with enterprise security standards. This setup enhances the overall security posture of your MySQL cluster.

Encryption flow

The encryption mechanism uses a two-tiered key architecture to secure your data:

- Each database instance has a master encryption key to encrypt tablespaces and binlogs. Master encryption key is stored separately from tablespace keys, in an external key management service like HashiCorp Vault.
- Each tablespace has a unique tablespace key to encrypt the data files (tables and indexes).

The data is encrypted before being written to disk. When you need to read the data, it's decrypted in memory for use and then re-encrypted before being written back to disk.

Key rotation

Key rotation is replacing the old master encryption key with the new one. When a new master encryption key is created, it is stored in Vault and tablespace keys are re-encrypted with it. The entire dataset is not re-encrypted and this makes the key rotation a fast and lightweight operation.

Read more about key rotation in the Rotate the master key.

Backups and encryption

Percona Operator for MySQL uses Percona XtraBackup for backups and fully supports backing up encrypted tablespaces. When your database uses the `keyring_vault` plugin, backup files retain that encryption layer as part of the physical backup data.

Starting with Operator 1.2.0, you can also encrypt backup files in the object storage independently of database encryption. This protects backup data even when live database encryption is not enabled.

Keep your encryption keys safe

To restore the encrypted data you must have the original key that was used to encrypt it. For data-at-rest encryption, this is the **the original Vault master encryption key**. For encrypted backups, this is the **backup encryption key** from the Kubernetes Secret.

If either key is lost or rotated, your backups will be irrecoverable. Always ensure you have a secure and reliable process for managing and backing up encryption keys separately from your database backups.

Next steps

Configure data-at-rest encryption

Configure data-at-rest encryption with HashiCorp Vault

This guide walks you through deploying and configuring HashiCorp Vault to work with Percona Operator for MySQL to enable data-at-rest encryption.

Create the namespace

It is a good practice to isolate workloads in Kubernetes using namespaces. Create a namespace with the following command:

```
kubectl create namespace vault
```

Export the namespace as an environment variable to simplify further configuration and management

```
NAMESPACE="vault"
```

Install Vault

For this setup, we install Vault in Kubernetes using the Helm 3 package manager. However, Helm is not required — any supported Vault deployment (on-premises, in the cloud, or a managed Vault service) works as long as the Operator can reach it.

1. Add and update the Vault Helm repository.

   ```
   helm repo add hashicorp https://helm.releases.hashicorp.com
   helm repo update
   ```

2. Install Vault

   ```
   helm upgrade --install vault hashicorp/vault --namespace $NAMESPACE
   ```

Sample output

```
NAME: vault
LAST DEPLOYED: Wed Aug 20 12:55:38 2025
NAMESPACE: vault
STATUS: deployed
REVISION: 1
NOTES:
Thank you for installing HashiCorp Vault!

Now that you have deployed Vault, you should look over the docs on using
Vault with Kubernetes available here:

https://developer.hashicorp.com/vault/docs
```

3. Retrieve the Pod name where Vault is running:

```
$(kubectl -n $NAMESPACE get pod -l app.kubernetes.io/name=vault -o
jsonpath='{.items[0].metadata.name}')
```

Sample output

```
vault-0
```

4. After Vault is installed, you need to initialize it. Run the following command:

```
kubectl exec -it pod/vault-0 -n $NAMESPACE -- vault operator init -key-
shares=1 -key-threshold=1 -format=json > /tmp/vault-init
```

   The command does the following:

   - Connects to the Vault Pod
   - Initializes Vault server
   - Creates 1 unseal key share which is required to unseal the server
   - Outputs the init response in JSON format to a local file `/tmp/vault-init`. It includes unseal keys and root token.

5. Vault is started in a sealed state. In this state Vault can access the storage but it cannot decrypt data. In order to use Vault, you need to unseal it.

   Retrieve the unseal key from the file:

```
unsealKey=$(jq -r ".unseal_keys_b64[]" < /tmp/vault-init)
```

Now, unseal Vault. Run the following command on every Pod where Vault is running:

```
kubectl exec -it pod/vault-0 -n $NAMESPACE -- vault operator unseal
"$unsealKey"
```

Sample output

```
Key             Value
---             -----
Seal Type       shamir
Initialized     true
Sealed          false
Total Shares    1
Threshold       1
Version         1.20.1
Build Date      2025-07-24T13:33:51Z
Storage Type    file
Cluster Name    vault-cluster-55062a37
Cluster ID      37d0c2e4-8f47-14f7-ca49-905b66a1804d
HA Enabled      false
```

Configure Vault

At this step you need to configure Vault and enable secrets within it. To do so you must first authenticate in Vault.

When you started Vault, it generates and starts with a root token that provides full access to Vault. Use this token to authenticate.

Note

For the purposes of this tutorial we use the root token in further sections. For security considerations, the use of root token is not recommended. Refer to the Create token in Vault documentation how to create user tokens.

1. Extract the Vault root token from the file where you saved the init response output:

```
cat /tmp/vault-init | jq -r ".root_token"
```

Sample output

```
hvs.*************Jg9r
```

2. Connect to Vault Pod:

```
kubectl exec -it vault-0 -n $NAMESPACE -- /bin/sh
```

3. Authenticate in Vault with this token:

```
vault login hvs.*************Jg9r
```

4. Enable the secrets engine at the mount path. The following command enables KV secrets engine v2 at the `ps-secret` mount-path:

```
vault secrets enable --version=2 -path=ps-secret kv
```

Sample output

```
Success! Enabled the kv secrets engine at: ps-secret/
```

Create a Secret for Vault

To enable Vault for the Operator, create a Secret object for it. To do so, create a YAML configuration file and specify the following information:

- A token to access Vault
- A Vault server URL
- The secrets mount path
- Path to TLS certificates if you deployed Vault with TLS
- Contents of the ca.cert certificate file

Depending on Percona Server for MySQL version, you must specify the Vault configuration as follows:

- For Percona Server for MySQL 8.0 - as key=value pairs
- For Percona Server for MySQL 8.4 - as a JSON object

You can modify the example `deploy/vault-secret.yaml` configuration file:

Percona Server for MySQL 8.0

HTTP access without TLS

```
apiVersion: v1
kind: Secret
metadata:
  name: ps-cluster1-vault
type: Opaque
stringData:
  keyring_vault.cnf: |-
    token = hvs.CvmS4c0DPTvHv5eJgXWMJg9r
    vault_url = http://vault.vault.svc.cluster.local:8200
    secret_mount_point = ps-secret
```

HTTPS access with TLS

```
apiVersion: v1
kind: Secret
metadata:
  name: ps-cluster1-vault
type: Opaque
stringData:
  keyring_vault.cnf: |-
    token = hvs.CvmS4c0DPTvHv5eJgXWMJg9r
    vault_url = https://vault.vault.svc.cluster.local:8200
    secret_mount_point = ps-secret
    vault_ca = /etc/mysql/vault-keyring-secret/ca.cert
  ca.cert: |-
    -----BEGIN CERTIFICATE-----
    MIIEczCCA1ugAwIBAgIBADANBgkqhkiG9w0BAQQFAD..AkGA1UEBhMCR0Ix
    EzARBgNVBAgTClNvbWUtU3RhdGUxFDASBgNVBAoTC0..0EgTHRkMTcwNQYD
    7vQMfXdGsRrXNGRGnX+vWDZ3/zWI0joDtCkNnqEpVn..HoX
    -----END CERTIFICATE-----
```

Percona Server for MySQL 8.4

HTTP access without TLS

```
apiVersion: v1
```

```
kind: Secret
metadata:
  name: ps-cluster1-vault-84
type: Opaque
stringData:
  keyring_vault.cnf: |-
    {
      "token": "hvs.CvmS4c0DPTvHv5eJgXWMJg9r",
      "vault_url": "http://vault.vault.svc.cluster.local:8200",
      "secret_mount_point": "ps-secret"
    }
```

HTTPS access with TLS

```
apiVersion: v1
kind: Secret
metadata:
  name: ps-cluster1-vault-84
type: Opaque
stringData:
  keyring_vault.cnf: |-
    {
      "token": "hvs.CvmS4c0DPTvHv5eJgXWMJg9r",
      "vault_url": "https://vault.vault.svc.cluster.local:8200",
      "secret_mount_point": "ps-secret",
      "vault_ca": "/etc/mysql/vault-keyring-secret/ca.cert"
    }
  ca.cert: |-
    -----BEGIN CERTIFICATE-----
    MIIEczCCA1ugAwIBAgIBADANBgkqhkiG9w0BAQQFAD..AkGA1UEBhMCR0Ix
    EzARBgNVBAgTClNvbWUtU3RhdGUxFDASBgNVBAoTC0..0EgTHRkMTcwNQYD
    7vQMfXdGsRrXNGRGnX+vWDZ3/zWI0joDtCkNnqEpVn..HoX
    -----END CERTIFICATE-----
```

Note that you must either specify the certificate value or don't declare it at all. Having a commented `#ca.cert` field in the Secret configuration file is not allowed.

Now create a Secret object. Replace the `<namespace>` placeholder with the namespace where your database cluster is deployed:

```
kubectl apply -f deploy/vault-secret.yaml -n <namespace>
```

If your deployment uses Group Replication as the cluster type, you must pause the cluster before patching to enable encryption.

After you add the required secret, unpause the cluster to resume normal operation.

Reference the Secret in your Custom Resource manifest

Now, reference the Vault Secret in the Operator Custom Resource manifest. Note that the Secret name is the one you specified in the `metadata.name` field when you created a Secret.

1. Export the namespace where the cluster is deployed as an environment variable:

   ```
   export ps-cluster-namespace = <cluster-namespace>
   ```

2. Update the cluster configuration. Since this is a running cluster, we will apply a patch.

Group replication

a. Pause the cluster:

```
kubectl patch ps ps-cluster1 \
  --namespace $<ps-cluster-namespace> \
  --type=merge \
  --patch '{"spec": {"pause": true}}'
```

b. Apply the patch referencing your Secret. Note for MySQL 8.0 the default Secret name is `ps-cluster1-vault` and for MySQL 8.4 - `ps-cluster1-vault-84`. Use the following command as an example and specify the Secret name for the MySQL version you're using:

```
kubectl patch ps ps-cluster1 \
  --namespace $<ps-cluster-namespace> \
  --type=merge \
  --patch '{"spec":{"mysql":{"vaultSecretName":"ps-cluster1-vault"}}}'
```

c. Unpause the cluster:

```
kubectl patch ps ps-cluster1 \
  --namespace $<ps-cluster-namespace> \
  --type=merge \
  --patch '{"spec": {"pause": false}}'
```

Asynchronous replication

Apply the patch referencing your Secret. Note for MySQL 8.0 the default Secret name is `ps-cluster1-vault` and for MySQL 8.4 - `ps-cluster1-vault-84`. Use the following command as an example and specify the Secret name for the MySQL version you're using:

```
kubectl patch ps ps-cluster1 \
  --namespace $<ps-cluster-namespace> \
  --type=merge \
  --patch '{"spec":{"mysql":{"vaultSecretName":"ps-cluster1-vault"}}}'
```

Use data-at-rest encryption

To use encryption, you can:

- turn it on for every table you create with the `ENCRYPTION='Y' clause in your SQL statement. For example,

```
CREATE TABLE t1 (c1 INT, PRIMARY KEY pk(c1)) ENCRYPTION='Y';
CREATE TABLESPACE foo ADD DATAFILE 'foo.ibd' ENCRYPTION='Y';
```

- turn on default encryption of a schema or a general tablespace. Then all tables you create will have encryption enabled. To turn on default encryption, use the following SQL statement:

```
SET default_table_encryption=ON;
```

Verify encryption

Refer to the Percona Server for MySQL documentation for guidelines how to verify encryption in your database.

External access

Expose cluster

The Operator provides different ways to access your MySQL database cluster. Each way uses Kubernetes Service objects to expose the cluster to client applications. These Service objects are configured by the Operator.

This document shows you how to configure cluster exposure using options in the Custom Resource manifest. The available options depend on the replication type of your cluster.

For a cluster with Asynchronous replication, your options are:

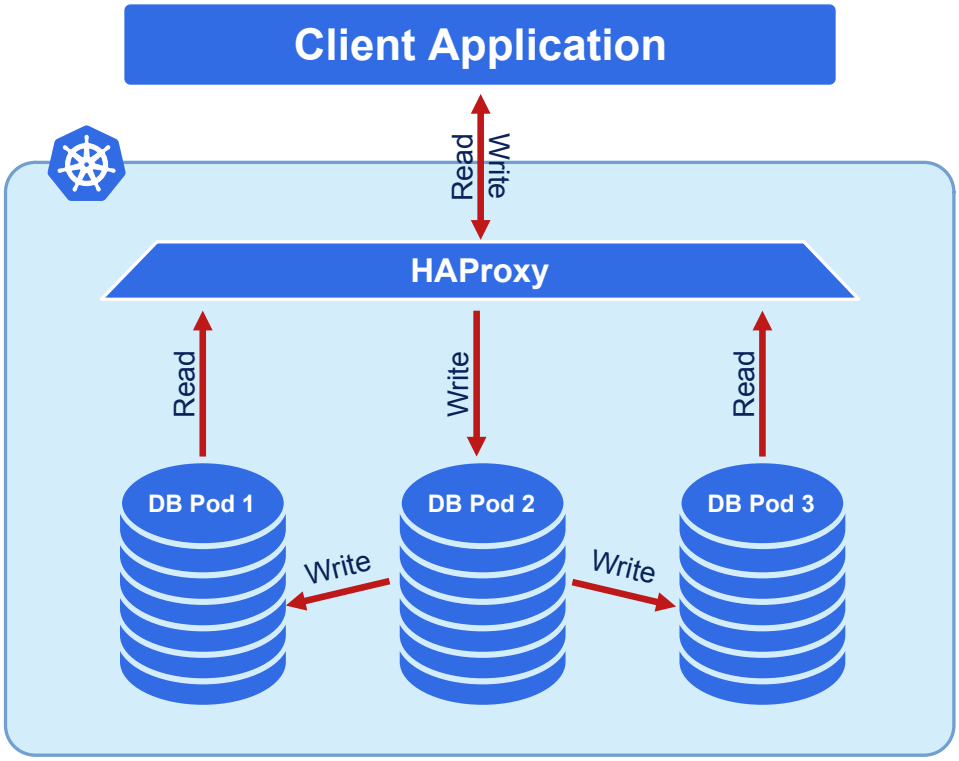
- Use HAProxy
- Use the Primary Service

For a cluster with Group Replication, your options are:

- Use HAProxy - recommended
- Use MySQL Router
- Use the Primary Service

Use HAProxy

HAProxy provides load balancing and proxy service for your cluster. It's enabled by default and works with both replication types.



To enable HAProxy, set the following in your `deploy/cr.yaml` manifest:

Asynchronous replication (tech preview)

```
mysql:
  clusterType: async
  ....
  haproxy:
    enabled: true
    size: 3
    image: perconalab/percona-server-mysql-operator:1.2.0-haproxy
```

Group replication

```
mysql:
  clusterType: group-replication
  ....
  haproxy:
    enabled: true
    size: 3
    image: perconalab/percona-server-mysql-operator:1.2.0-haproxy
```

The created HAProxy service (`ps-cluster1-haproxy`) listens on the following ports:

- `3306`: MySQL primary

- `3307`: MySQL replicas
- `3309`: Proxy protocol

To find your HAProxy endpoint, run:

```
kubectl get service ps-cluster1-haproxy
```

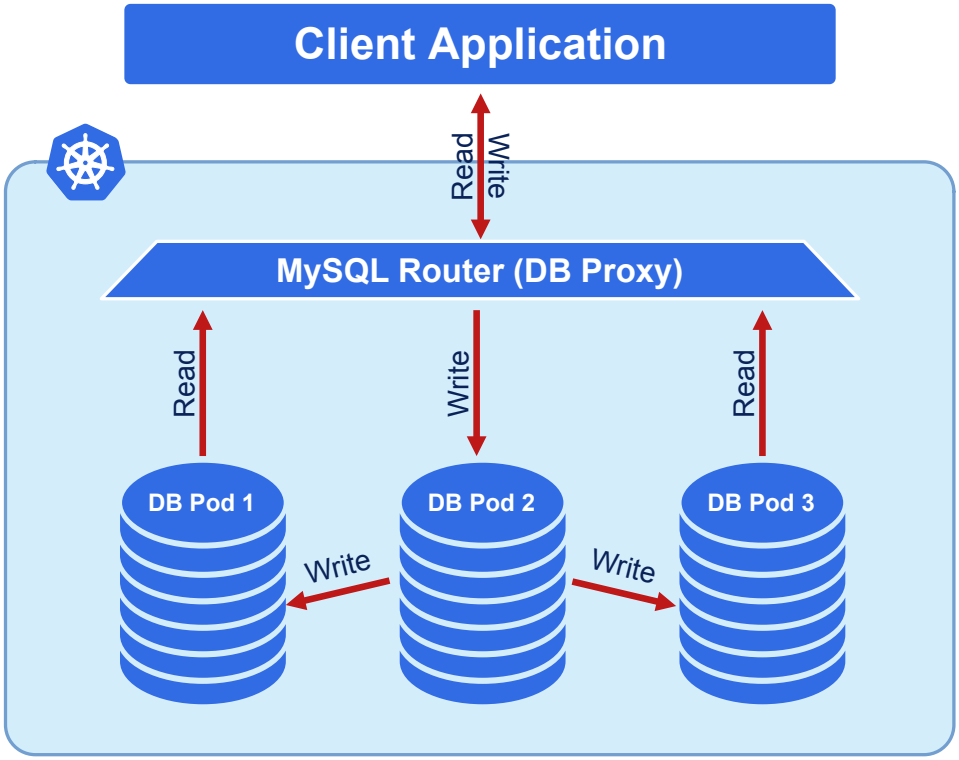
Sample output

```
NAME                TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE
ps-cluster1-haproxy   ClusterIP   10.76.2.102   <none>        3306/TCP,3307/TCP,3309/TCP
2m32s
```

Use MySQL Router

MySQL Router provides intelligent routing for group replication clusters. This option is left for backward compatibility. We recommend to Use HAProxy for exposing the cluster.

You can expose the cluster through a `<CLUSTER_NAME>-router` Kubernetes Service.



To configure MySQL Router with the LoadBalancer expose type, modify the `spec.router` section in `deploy/cr.yaml` manifest:

```
mysql:
  clusterType: group-replication
  ...
router:
  expose:
    type: LoadBalancer
```

To find your MySQL Router endpoint, run:

```
kubectl get service ps-cluster1-router
```

Sample output

```
NAME                TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)
AGE
my-cluster-router   LoadBalancer   10.20.22.90   35.223.42.238
6446:30852/TCP,6447:31694/TCP,6448:31515/TCP,6449:31686/TCP   18h
```

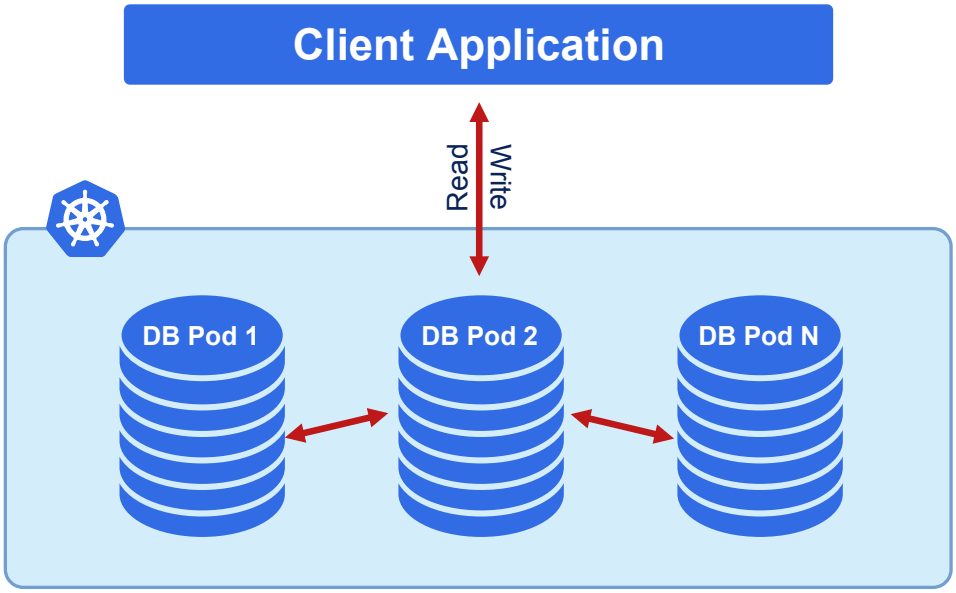
The MySQL Router service provides these ports:

- `3306` - read/write, default MySQL clients connection,
- `33062` - read/write, port for MySQL administrative connections,
- `6446` - read/write, routing traffic to a Primary node,
- `6447` - read-only, load balancing the traffic across Replicas.

Additional ports `6448` and `6449` are available to connect via MySQL X Protocol. This is useful for operations such as asynchronous calls.

Use Primary Service

You can expose your cluster without the proxy by exposing the primary Pod directly. Specify the `spec.mysql.exposePrimary.enabled` option to `true` in your Custom Resource. This creates the `<CLUSTER_NAME>-mysql-primary` service for connecting to the cluster.



You can change the type of the Service object by setting `mysql.exposePrimary.type` variable in the Custom Resource. For example, to use a LoadBalancer for the primary service, specify the following configuration in your `deploy/cr.yaml` manifest:

Asynchronous replication

```
mysql:
  clusterType: async
  ...
  exposePrimary:
    enabled: true
    type: LoadBalancer
```

Group replication

```
mysql:
  clusterType: group-replication
  ...
  exposePrimary:
    enabled: true
    type: LoadBalancer
```

To find your primary service endpoint, run:

```
kubectl get service ps-cluster1-mysql-primary
```

Sample output

```
NAME                       TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)
AGE
ps-cluster1-mysql-primary   LoadBalancer   10.40.37.98    35.192.172.85
3306:32146/TCP,33062:31062/TCP,33060:32026/TCP,6033:30521/TCP   3m31s
```

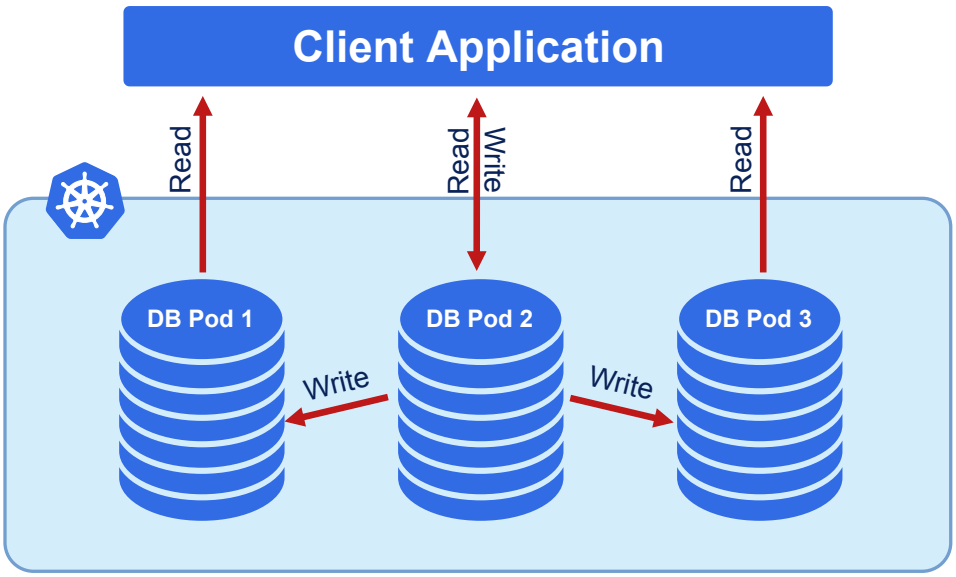
The `ps-cluster1-mysql-primary` Service listens on the following ports:

- `3306` - read/write, default MySQL clients connection,
- `33062` - read/write, port for MySQL administrative connections,
- `33060` - read/write, connection to MySQL via the MySQL X protocol
- `6450` - read/write, connection to MySQL via the MySQL Router
- `33061` - MySQL Group Replication internal communications port

In addition, the primary Pod is marked with the label `mysql.percona.com/primary=true` to distinguish it from the rest of the Pods.

Expose Individual Pods

Sometimes you need to expose each MySQL instance with its own IP address. This is useful when implementing load balancing at the application level.



To expose individual pods, configure the following in your `deploy/cr.yaml`:

```
mysql:
  expose:
    enabled: true
    type: LoadBalancer
```

To find all exposed services, run:

```
kubectl get services
```

Sample output

```
NAME                    TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)
AGE
...
ps-cluster1-mysql-0         LoadBalancer   10.40.44.110   104.198.16.21
3306:31009/TCP,33062:31319/TCP,33060:30737/TCP,6033:30660/TCP   75s
ps-cluster1-mysql-1         LoadBalancer   10.40.42.5     34.70.170.187
3306:30601/TCP,33062:30273/TCP,33060:30910/TCP,6033:30847/TCP   75s
ps-cluster1-mysql-2         LoadBalancer   10.40.42.158   35.193.50.44
3306:32042/TCP,33062:31576/TCP,33060:31656/TCP,6033:31448/TCP   75s
```

As you could notice, this command also shows mapped ports the application can use to communicate with MySQL instances (e.g. `3306` for the classic MySQL protocol, or `33060` for MySQL X Protocol useful for operations such as asynchronous calls).

Configuring Load Balancing with HAProxy

Percona Operator for MySQL provides load balancing and proxy service with HAProxy (enabled by default). HAProxy is the only solution proxy when asynchronous replication between MySQL instances is enabled, while group replication can be used either with HAProxy or MySQL Router. You can control whether to use HAProxy or not by enabling or disabling it via the `haproxy.enabled` option in the `deploy/cr.yaml` configuration file.

> **Note**
>
> When enabling HAProxy, make sure MySQL Router is disabled (`proxy.router.enabled` option should be set to `false`).

For example, you can use the following command to enable HAProxy for existing cluster:

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
  "spec": {
     "proxy": {
       "haproxy": {
          "enabled": true,
          "size": 3,
          "image": "percona/haproxy:2.8.18-1" }
       "router": {
          "enabled": false }
       }
  }}'
```

The resulting HAPproxy setup will contain the `ps-cluster1-haproxy` service listening on ports 3306 (MySQL cluster zero member) and 3307 (other members).

This service is pointing to the MySQL cluster member number zero (`ps-cluster1-mysql-0`) on the default 3306 port when this member is available. If a zero member is not available, members are selected in descending order of their numbers (e.g. `ps-cluster1-mysql-2`, then `ps-cluster1-mysql-1`, etc.). It can be used for both read and write load, or it can also be used just for write load (single writer mode) in setups with split write and read loads. On 3307 port this service selects MySQL cluster members to serve queries following the Round Robin load balancing algorithm.

When the cluster with HAProxy is upgraded, the following steps take place. First, reader members are upgraded one by one: the Operator waits until the upgraded Percona Distribution for MySQL cluster member becomes synced, and then proceeds to upgrade the next member. When the upgrade is finished for all the readers, then the writer MySQL cluster member is finally upgraded.

Passing custom configuration options to HAProxy

You can pass custom configuration to HAProxy, adding options from the haproxy.cfg configuration file to the `haproxy.configuration` Custom Resource option in the `deploy/cr.yaml` file. Here is an example:

```
...
haproxy:
  enabled: true
  size: 3
  image: perconalab/percona-xtradb-cluster-operator:1.2.0-haproxy
  configuration: |
    global
      maxconn 2048
      external-check
      insecure-fork-wanted
      stats socket /var/run/haproxy.sock mode 600 expose-fd listeners level
admin
    defaults
      default-server init-addr last,libc,none
      log global
      mode tcp
      retries 10
      timeout client 28800s
      timeout connect 100500
      timeout server 28800s
    frontend mysql-primary-in
      bind *:3309 accept-proxy
      bind *:3306
      mode tcp
      option clitcpka
      default_backend mysql-primary
    frontend mysql-replicas-in
      bind *:3307
      mode tcp
      option clitcpka
      default_backend mysql-replicas
    frontend stats
      bind *:8404
      mode http
      http-request use-service prometheus-exporter if { path /metrics }
```

the actual default configuration file can be found here.

MySQL Router Configuration

MySQL Router is lightweight middleware that provides transparent routing between your application and back-end MySQL servers. MySQL Router is part of the Operator and is deployed during the installation. MySQL Router can be used as an alternative to HAProxy based load balancing when group replication between MySQL instances is turned on.

To use the Router, enable it and make sure that HAProxy is disabled.

Enable MySQL Router

1. Edit the `deploy/cr.yaml` file

```
...
mysql:
  clusterType: group-replication
  ...
proxy:
  haproxy:
    enabled: false
    ...
  router:
    enabled: true
    ...
```

2. Update the cluster to apply the new configuration:

```
kubectl apply -f deploy/cr.yaml
```

When the process is over your cluster will obtain the `ready` status. You can check it with the following command:

```
kubectl get ps
```

Expected output

```
NAME       REPLICATION         ENDPOINT                  STATE   MYSQL
ORCHESTRATOR   HAPROXY   ROUTER   AGE
ps-cluster1   group-replication   ps-cluster1-router.default   ready   3
3        53m
```

Configure MySQL Router

When enabled, the MySQL Router operates with the reasonable default settings and can be used in a variety of use cases such as high-availability and scalability.

If you need to fine-tune the Router for the needs of your application and/or usage scenario, you can do this using the following methods:

- Edit the `deploy/cr.yaml` file
- Use the ConfigMap

To illustrate this, let's override the verbosity level and set it to `INFO`.

Before you start, check that you have enabled the MySQL Router for the Operator.

deploy/cr.yaml

The `router.configuration` subsection of the `deploy.cr.yaml` file contains the MySQL Router configuration.

1. To change the verbosity level, edit the configuration file as follows:

```
configuration: |
 [default]
 logging_folder=/tmp/router/log
 [logger]
 level=INFO
```

2. Update the cluster to apply the new configuration

```
kubectl apply -f deploy.cr.yaml
```

ConfigMap

A ConfigMap is a Kubernetes mechanism to pass or update configuration data inside a containerized application.

You can create a ConfigMap from a file using the `kubectl create configmap` command. For more information about ConfigMap usage, see Configure a Pod to use a ConfigMap.

To pass the new verbosity level of MySQL Router to the Operator using the ConfigMap, do the following:

1. Create the `mysqlrouter.conf` configuration file and specify the new verbosity level within.

   ```
   [logger]
   level = INFO
   ```

2. Get the name of your cluster to pass the configuration

   ```
   kubectl get ps
   ```

3. Create the ConfigMap. You should use the combination of the cluster name with the `-router` suffix as the naming convention for the ConfigMap. For example, to create the ConfigMap for the cluster `ps-cluster1`, the command is the following:

   ```
   kubectl create configmap ps-cluster1-router --from-file=mysqlrouter.conf
   ```

   Replace the `ps-cluster1` with the corresponding name of your cluster.

4. View the created ConfigMap using the following command:

   ```
   kubectl describe configmaps ps-cluster1-mysql
   ```

Control Pod scheduling on specific Kubernetes nodes with affinity, anti-affinity and tolerations

The Operator automatically assigns Pods to nodes with sufficient resources for balanced distribution across the cluster. You can configure Pods to be scheduled on specific nodes. For example, for improved performance on the SSD equipped machine or for cost optimization by choosing the nodes in the same availability zone.

Using the deploy/cr.yaml Custom Resource manifest, you can configure the following:

- Affinity and anti-affinity rules to bind Pods to specific Kubernetes nodes
- Taints and tolerations to ensure that Pods are not scheduled onto inappropriate nodes

Affinity and Anti-affinity

Affinity controls Pod placement based on nodes which already have Pods with specific labels. Use affinity to:

- Reduce costs by placing Pods in the same availability zone
- Improve high availability by distributing Pods across different nodes or zones

The Operator provides two approaches:

- Simple - set anti-affinity using built-in options
- Advanced - using Kubernetes constraints

Simple Anti-affinity

This approach does not require the knowledge of how Kubernetes assigns Pods to specific nodes.

Use the `antiAffinityTopologyKey` option with these values:

- `kubernetes.io/hostname` - Pods avoid the same host
- `topology.kubernetes.io/zone` - Pods avoid the same zone
- `topology.kubernetes.io/region` - Pods avoid the same region
- `none` - No constraints applied

Example

This configuration forces Percona Server for MySQL Pods to avoid reside on the same node:

```
affinity:
  antiAffinityTopologyKey: "kubernetes.io/hostname"
```

Advanced anti-affinity via Kubernetes constraints

For complex scheduling requirements, use the `advanced` option. This disables the `antiAffinityTopologyKey` effect and allows the use of standard Kubernetes affinity constraints:

```
affinity:
   advanced:
     podAffinity:
       requiredDuringSchedulingIgnoredDuringExecution:
       - labelSelector:
           matchExpressions:
           - key: security
             operator: In
             values:
             - S1
         topologyKey: topology.kubernetes.io/zone
     podAntiAffinity:
       preferredDuringSchedulingIgnoredDuringExecution:
       - weight: 100
         podAffinityTerm:
           labelSelector:
             matchExpressions:
             - key: security
               operator: In
               values:
               - S2
           topologyKey: kubernetes.io/hostname
     nodeAffinity:
       requiredDuringSchedulingIgnoredDuringExecution:
         nodeSelectorTerms:
         - matchExpressions:
           - key: kubernetes.io/e2e-az-name
             operator: In
             values:
             - e2e-az1
             - e2e-az2
       preferredDuringSchedulingIgnoredDuringExecution:
       - weight: 1
         preference:
           matchExpressions:
           - key: another-node-label-key
             operator: In
             values:
             - another-node-label-value
```

See Kubernetes affinity documentation for detailed explanations of these options.

Tolerations

Tolerations allow Pods to run on nodes with matching taints. A taint is a key-value pair associated with a node that marks the node to repel certain Pods.

Taints and tolerations work together to ensure Pods are not scheduled onto inappropriate nodes.

A toleration includes these fields:

- `key` - The taint key to match
- `operator` - Either `exists` (matches any value) or `equal` (requires exact value match)
- `value` - Required when `operator` is `equal`
- `effect` - The taint effect to tolerate:
- `NoSchedule` - Pods cannot be scheduled on the node
- `PreferNoSchedule` - Pods are discouraged from scheduling on the node
- `NoExecute` - Pods are evicted from the node (with optional `tolerationSeconds`)

This is the example configuration of a toleration:

```
tolerations:
- key: "node.alpha.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 6000
```

Common use cases

- **Dedicated nodes**: Reserve nodes for specific workloads by tainting them and adding corresponding tolerations to authorized Pods.
- **Special hardware**: Keep Pods that don't need specialized hardware (like GPUs) off dedicated nodes by tainting those nodes.
- **Node problems**: Handle node failures gracefully with automatic taints and tolerations.

See Kubernetes Taints and Tolerations for detailed examples and use cases.

Priority classes

Pods may belong to some *priority classes*. Priority classes help the scheduler distinguish important Pods when eviction is needed. To use priority classes:

1. Create PriorityClasses in your Kubernetes cluster
2. Specify `PriorityClassName` in the deploy/cr.yaml file:

```
priorityClassName: high-priority
```

See Kubernetes Pod Priority documentation for more information on how to define and use priority classes in your cluster.

Pod Disruption Budgets

A Pod Disruption Budget (PDB) in Kubernetes helps keep your applications available during voluntary disruptions, such as deleting a deployment or draining a node for maintenance by a cluster administrator. A Pod Disruption Budget sets a limit on how many Pods can be unavailable at the same time due to these voluntary actions.

You can configure Pod disruption budget for Percona Server for MySQL, HAProxy, MySQL Router and Orchestrator Pods using the `podDisruptionBudget` option in the Custom Resource.

This is the example configuration for Percona Server for MySQL Pods:

```
mysql:
  podDisruptionBudget:
      maxUnavailable: 1
      minAvailable: 0
```

Refer to the official Kubernetes documentation for more information about Pod disruption budgets and considerations how to protect your application.

Changing MySQL Options

You may require a configuration change for your application. MySQL allows the option to configure the database with a configuration file. You can pass options from the my.cnf configuration file to be included in the MySQL configuration in one of the following ways:

- edit the `deploy/cr.yaml` file,
- use a ConfigMap,
- use a Secret object.

Edit the `deploy/cr.yaml` file

You can add options from the my.cnf configuration file by editing the configuration section of the `deploy/cr.yaml`. Here is an example:

```
spec:
  secretsName: ps-cluster1-secrets
  mysql:
    ...
      configuration: |
        max_connections=250
```

See the Custom Resource options, MySQL section for more details.

Use a ConfigMap

You can use a configmap and the cluster restart to reset configuration options. A configmap allows Kubernetes to pass or update configuration data inside a containerized application.

Use the `kubectl` command to create the configmap from external resources, for more information see Configure a Pod to use a ConfigMap.

For example, let's suppose that your application requires more connections. To increase your `max_connections` setting in MySQL, you define a `my.cnf` configuration file with the following setting:

```
max_connections=250
```

You can create a configmap from the `my.cnf` file with the `kubectl create configmap` command.

You should use the combination of the cluster name with the `-mysql` suffix as the naming convention for the configmap. To find the cluster name, you can use the following command:

```
kubectl get ps
```

The syntax for `kubectl create configmap` command is:

```
kubectl create configmap <cluster-name>-mysql <resource-type=resource-name>
```

The following example defines `ps-cluster1-mysql` as the configmap name and the `my.cnf` file as the data source:

```
kubectl create configmap ps-cluster1-mysql --from-file=my.cnf
```

To view the created configmap, use the following command:

```
kubectl describe configmaps ps-cluster1-mysql
```

Use a Secret Object

The Operator can also store configuration options in Kubernetes Secrets. This can be useful if you need additional protection for some sensitive data.

You should create a Secret object with a specific name, composed of your cluster name and the `mysql` suffix.

Note

To find the cluster name, you can use the following command:

```
kubectl get ps
```

Configuration options should be put inside a specific key inside of the `data` section. The name of this key is `my.cnf` for Percona Server for MySQL pods.

Actual options should be encoded with Base64.

For example, let's define a `my.cnf` configuration file and put there a pair of MySQL options we used in the previous example:

```
max_connections=250
```

You can get a Base64 encoded string from your options via the command line as follows:

in Linux

```
cat my.cnf | base64 --wrap=0
```

in macOS

```
cat my.cnf | base64
```

> **Note**
>
> Similarly, you can read the list of options from a Base64 encoded string:
>
> ```
> echo "bWF4X2Nvbm5lY3Rpb25zPTI1MAo" | base64 --decode
> ```

Finally, use a yaml file to create the Secret object. For example, you can create a `deploy/mysql-secret.yaml` file with the following contents:

```
apiVersion: v1
kind: Secret
metadata:
  name: ps-cluster1-mysql
data:
  my.cnf: "bWF4X2Nvbm5lY3Rpb25zPTI1MAo"
```

When ready, apply it with the following command:

```
kubectl create -f deploy/mysql-secret.yaml
```

Note

Do not forget to restart Percona Server for MySQL pods to ensure the cluster has updated the configuration. You can do it with the following command:

```
kubectl rollout restart statefulset ps-cluster1-mysql
```

Auto-tuning MySQL options

Few configuration options for MySQL can be calculated and set by the Operator automatically based on the available Pod memory resource limits **if constant values for these options are not specified by the user** (either in cr.yaml or in ConfigMap).

Options which can be set automatically are the following ones:

- `innodb_buffer_pool_size`
- `max_connections`

If Percona Server for MySQL container resource limits are defined, then limits values are used to calculate these options. If Percona Server for MySQL container resource limits are not defined, auto-tuning is not done.

Also, starting from the Operator 0.4.0, there is another way of auto-tuning. You can use `"{{ containerMemoryLimit }}"` as a value in `spec.mysql.configuration` as follows:

```
mysql:
    configuration: |
    [mysqld]
    innodb_buffer_pool_size={{containerMemoryLimit * 3 / 4}}
    ...
```

Configure DNS suffix for service discovery

In Kubernetes, services are assigned a DNS name to be accessible within the cluster. The domain name follows the pattern `<service-name>.<namespace>.svc.<cluster-domain>`. The default cluster domain is `cluster.local`, so a typical FQDN looks like `<service-name>.<namespace>.svc.cluster.local`.

When you refer to a service using only its short name, Kubernetes automatically expands it with this domain so the name resolves inside the cluster. This enables workloads to communicate without the need to specify fully-qualified domain names.

Clusters with custom DNS configurations may use a domain suffix different from the default `cluster.local`. This includes scenarios such as running inside a vcluster, which may respond to DNS queries with records that do not match the actual cluster domain.

The Operator tries to auto-detect the cluster's domain suffix by performing a CNAME lookup on the `kubernetes` Service in the `default` namespace (`kubernetes.default.svc`). If the lookup returns a value, the Operator uses that as the domain suffix. If auto-detection fails, the Operator falls back to the default `cluster.local` domain. This may lead to services not resolving properly within the cluster.

To address this, you can set your custom DNS suffix for the Operator to use when it constructs service names. As a result, the Operator produces hostnames that match your cluster's DNS configuration, ensuring correct service resolution and discovery.

How to configure

Add `clusterServiceDNSSuffix` under `spec` in your `PerconaServerMySQL` Custom Resource. Set it to your cluster's DNS suffix. For example, `cluster.local` for a standard cluster, or the host cluster's suffix when the Operator runs in a vcluster:

```
spec:
  ...
  clusterServiceDNSSuffix: cluster.local
  # ... rest of your spec
```

Apply the change:

```
kubectl apply -f deploy/cr.yaml -n <namespace>
```

Percona Operator for MySQL single-namespace and multi-namespace deployment

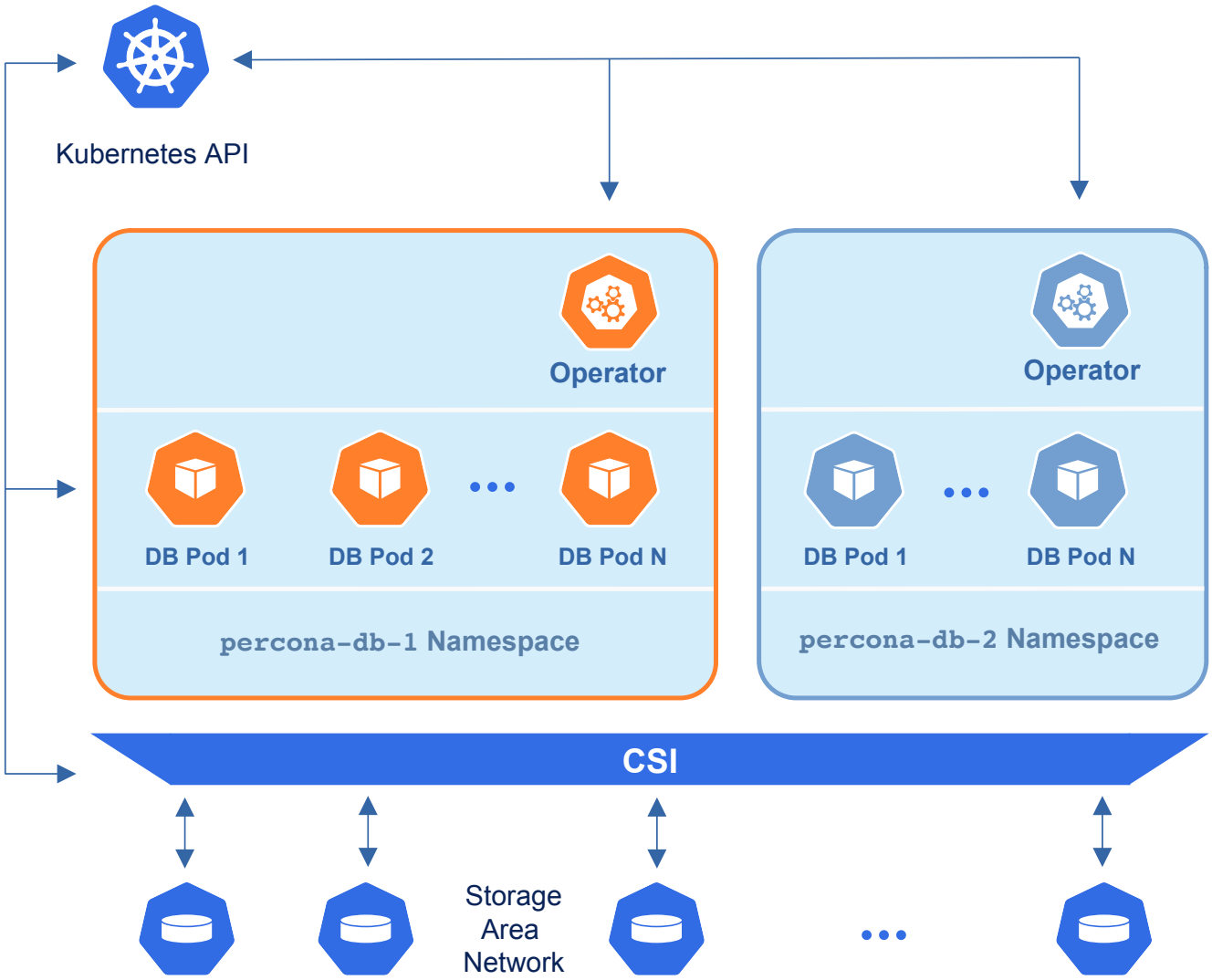
There are two design patterns that you can choose from when deploying Percona Operator for MySQL and Percona Server for MySQL clusters in Kubernetes:

- Namespace-scope - one Operator per Kubernetes namespace,
- Cluster-wide - one Operator can manage clusters in multiple namespaces.

This how-to explains how to configure Percona Operator for MySQL based on Percona Server for MySQL for each scenario.

Namespace-scope

By default, Percona Operator for MySQL functions in a specific Kubernetes namespace. You can create one during the installation (like it is shown in the installation instructions) or just use the `default` namespace. This approach allows several Operators to co-exist in one Kubernetes-based environment, being separated in different namespaces:



Normally this is a recommended approach, as isolation minimizes impact in case of various failure scenarios. This is the default configuration of our Operator.

Let's say you will use a Kubernetes Namespace called `percona-db-1`.

1. Clone `percona-server-mysql-operator` repository:

```
$ git clone -b v1.2.0 https://github.com/percona/percona-server-mysql-operator
$ cd percona-server-mysql-operator
```

2. Create your `percona-db-1` Namespace (if it doesn't yet exist) as follows:

```
$ kubectl create namespace percona-db-1
```

3. Deploy the Operator using the following command:

```
$ kubectl apply --server-side -f deploy/bundle.yaml -n percona-db-1
```

4. Once Operator is up and running, deploy the database cluster itself:

```
$ kubectl apply -f deploy/cr.yaml -n percona-db-1
```

You can deploy multiple clusters in this namespace.

Add more namespaces

What if there is a need to deploy clusters in another namespace? The solution for namespace-scope deployment is to have more than one Operator. We will use the `percona-db-2` namespace as an example.

1. Create your `percona-db-2` namespace (if it doesn't yet exist) as follows:

```
$ kubectl create namespace percona-db-2
```

2. Deploy the Operator:

```
$ kubectl apply --server-side -f deploy/bundle.yaml -n percona-db-2
```

3. Once Operator is up and running deploy the database cluster itself:

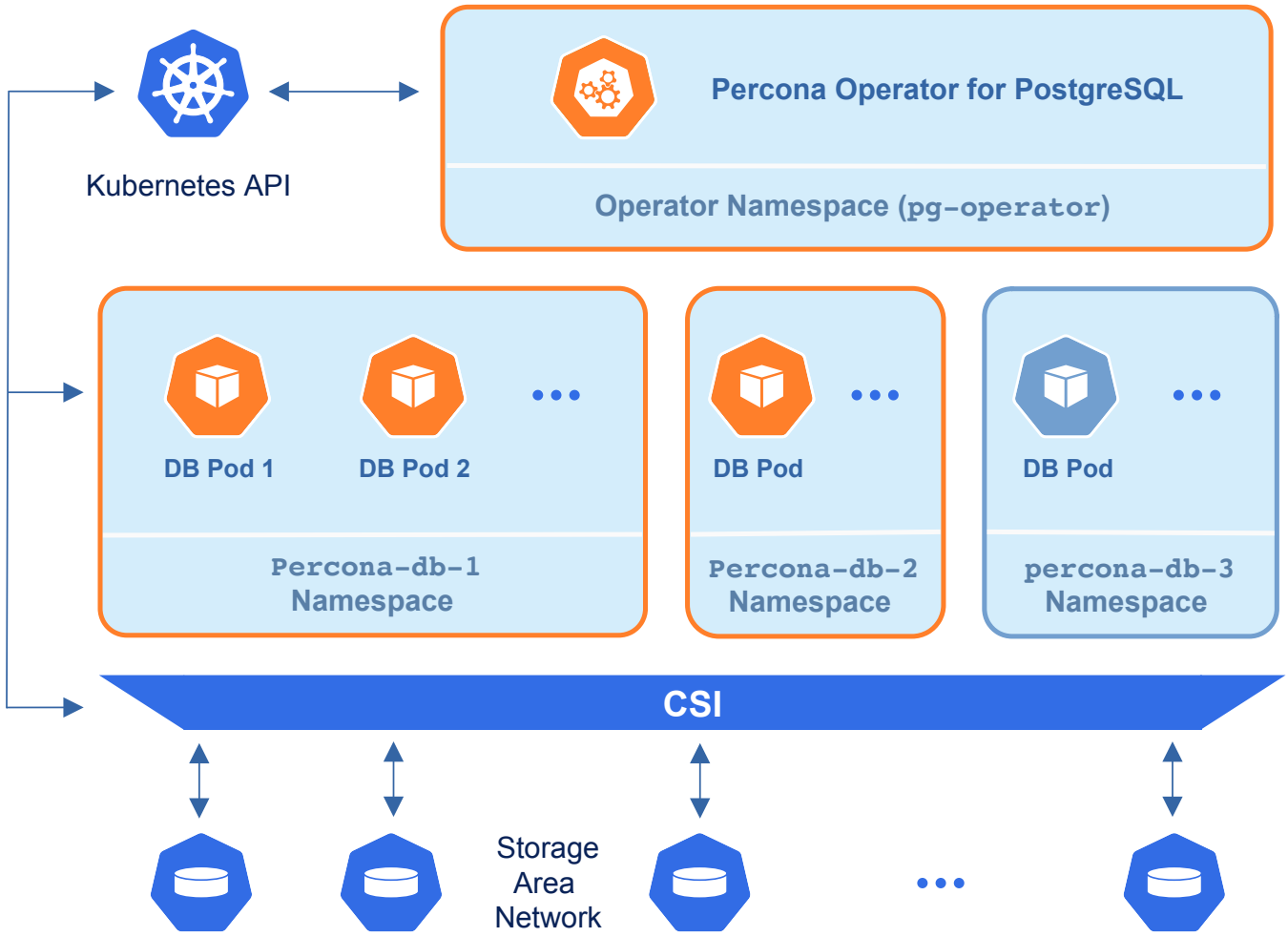
```
$ kubectl apply -f deploy/cr.yaml -n percona-db-2
```

> **Note**
>
> Cluster names may be the same in different namespaces.

Install the Operator cluster-wide

Sometimes it is more convenient to have one Operator watching for Percona Server for MySQL custom resources in several namespaces.

We recommend running Percona Operator for MySQL in a traditional way, limited to a specific namespace, to limit the blast radius. But it is possible to run it in so-called *cluster-wide* mode, one Operator watching several namespaces, if needed:



To use the Operator in such cluster-wide mode, you should install it with a different set of configuration YAML files, which are available in the deploy folder and have filenames with a special `cw-` prefix: e.g. `deploy/cw-bundle.yaml`.

While using this cluster-wide versions of configuration files, you should set the following information there:

- `subjects.namespace` option should contain the namespace which will host the Operator,
- `WATCH_NAMESPACE` key-value pair in the `env` section should have `value` equal to a comma-separated list of the namespaces to be watched by the Operator (or just a blank string to make the Operator deal with *all namespaces* in a Kubernetes cluster).

The following simple example shows how to install Operator cluster-wide on Kubernetes.

1. Clone `percona-server-mysql-operator` repository:

```
$ git clone -b v1.2.0 https://github.com/percona/percona-server-mysql-operator
$ cd percona-server-mysql-operator
```

2. Let's say you will use `ps-operator` namespace for the Operator, and `percona-db-1` namespace for the cluster. Create these namespaces, if needed:

```
$ kubectl create namespace ps-operator
$ kubectl create namespace percona-db-1
```

3. Edit the `deploy/cw-bundle.yaml` configuration file to make sure it contains proper namespace name for the Operator:

```
...
subjects:
- kind: ServiceAccount
  name: percona-server-mysql-operator
  namespace: ps-operator
...
spec:
  containers:
  - command:
    ...
    env:
    - name: WATCH_NAMESPACE
      value: "percona-db-1"
    ...
```

4. Apply the `deploy/cw-bundle.yaml` file with the following command:

```
$ kubectl apply --server-side -f deploy/cw-bundle.yaml -n ps-operator
```

   Right now the operator deployed in cluster-wide mode will monitor all namespaces in the cluster, either already existing or newly created ones.

5. Deploy the cluster in the namespace of your choice:

```
$ kubectl apply -f deploy/cr.yaml -n percona-db-1
```

Verifying the cluster operation

When creation process is over, you can try to connect to the cluster.

To connect to Percona Server for MySQL, use the connection Secret that the Operator creates for the `root` user. It is named `<cluster_name>-psuser-root` (by default, `ps-cluster1-psuser-root`) and contains hostnames, ports, credentials, and ready-to-use connection URIs. Read Connection secrets for the full reference.

Here's how to connect:

1. Export the namespace, cluster name and the Secret name as environment variables:

   ```
   export NAMESPACE=my-namespace
   export CLUSTER_NAME=ps-cluster1
   export SECRET_NAME=${CLUSTER_NAME}-psuser-root
   ```

   Replace `ps-cluster1` with the name of your cluster if you changed it during installation.

2. List the Secrets objects.

   ```
   $ kubectl get secrets -n $NAMESPACE
   ```

   Look for the Secret named `<cluster_name>-psuser-root`. By default, it is `ps-cluster1-psuser-root`.

3. Retrieve the user credentials from the Secret:

   `bash`
   `kubectl get secret "$SECRET_NAME" -n "$NAMESPACE" \`
   `  -o jsonpath='{.data.user}' | base64 --decode && echo`
   `kubectl get secret "$SECRET_NAME" -n "$NAMESPACE" \`
   `  -o jsonpath='{.data.password}' | base64 --decode && echo`

4. Run a container with the `mysql` tool and connect its console output to your terminal:

   ```
   kubectl run -i --rm --tty percona-client --image=percona/percona-server:8.4 --restart=Never -- bash -il
   ```

   It may require some time to execute the command and deploy the corresponding Pod.

5. Now run `mysql` tool in the `percona-client` command shell using the password obtained from the Secret instead of the `<root password>` placeholder. The command will look different depending on whether the cluster uses load balancing with HAProxy (the default behavior) or uses MySQL Router (can be used with Group Replication clusters):

   If using HAProxy (default)

   ```
   mysql -h ps-cluster1-haproxy -uroot -p<root password>
   ```

   If using MySQL Router

   ```
   mysql -h ps-cluster1-router -uroot -p<root password>
   ```

   Expected output

   ```
   mysql: [Warning] Using a password on the command line interface can be insecure.
   Welcome to the MySQL monitor.  Commands end with ; or \g.
   Your MySQL connection id is 1665
   Server version: 8.4.10-10.1 Percona Server (GPL), Release 6, Revision dbba4396

   Copyright (c) 2009-2026 Percona LLC and/or its affiliates
   Copyright (c) 2000, 2026, Oracle and/or its affiliates.

   Oracle is a registered trademark of Oracle Corporation and/or its
   affiliates. Other names may be trademarks of their respective
   owners.

   Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

   mysql>
   ```

   The following example uses the MySQL prompt to check the `max_connections` variable:

   ```
   SHOW VARIABLES LIKE "max_connections";
   ```

Expected output

```
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_connections | 158   |
+-----------------+-------+
1 row in set (0.02 sec)

mysql>
```

Note

Some Kubernetes-based environments are specifically configured to have communication across Namespaces is not allowed by default network policies. In this case, you should specifically allow the Operator communication across the needed Namespaces. Following the above example, you would need to allow ingress traffic for the `ps-operator` Namespace from the `percona-db-1` Namespace, and also from the `default` Namespace. You can do it with the NetworkPolicy resource, specified in the YAML file as follows:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: percona
  namespace: ps-operator
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: percona-db-1
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: default
  podSelector: {}
  policyTypes:
  - Ingress
```

Don't forget to apply the resulting file with the usual `kubectl apply` command.

You can find more details about Network Policies in the official Kubernetes documentation.

Upgrading the Operator in cluster-wide mode

Cluster-wide Operator is upgraded similarly to a single-namespace one. Both deployment variants provide you with the same three upgradable components:

- the Operator;
- Custom Resource Definition (CRD),
- Database Management System (Percona Server for MySQL).

To upgrade the cluster-wide Operator you follow the standard upgrade scenario concerning the Operator's namespace and a different YAML configuration file: the one with a special `cw-` prefix, `deploy/cw-rbac.yaml`. The resulting steps will look as follows.

1. Update the Custom Resource Definition for the Operator, and do the same for the Role-based access control:

   ```
   $ kubectl apply -f deploy/crd.yaml
   $ kubectl apply -f deploy/cw-rbac.yaml
   ```

2. Now you should apply a patch to your deployment, supplying the necessary image name with a newer version tag. You can find the proper image name for the current Operator release in the list of certified images. For example, updating to the `1.2.0` version in the `ps-operator` namespace should look as follows.

   ```
   $ kubectl patch deployment percona-server-mysql-operator \
     -p'{"spec":{"template":{"spec":{"containers":[{"name":"percona-server-
   mysql-operator","image":"percona/percona-server-mysql-
   operator:1.2.0"}]}}}}' -n ps-operator
   ```

3. The deployment rollout will be automatically triggered by the applied patch. You can track the rollout process in real time with the `kubectl rollout status` command with the name of your cluster:

   ```
   $ kubectl rollout status deployments percona-server-mysql-operator -n ps-
   operator
   ```

Cross-site replication

Cross-site replication

Enterprises running MySQL at scale often need to span multiple regions, availability zones, or clouds, with a live copy of data in each location. Each site must be a live replica of the primary site, protected from accidental writes and ready to take over traffic during an outage or planned failover. Group Replication keeps nodes in sync within a single cluster, but it does not by itself connect separate clusters across sites.

Cross-site replication closes that gap. It links multiple Group Replication clusters into a single MySQL InnoDB ClusterSet managed through a dedicated `PerconaServerMySQLClusterSet` Custom Resource (`ps-clusterset`). Each site keeps its own `PerconaServerMySQL` cluster of the Group Replication type. The ClusterSet Custom Resource coordinates replication and topology between them.

Why to use cross-site replication

With cross-site replication, you can achieve the following scenarios:

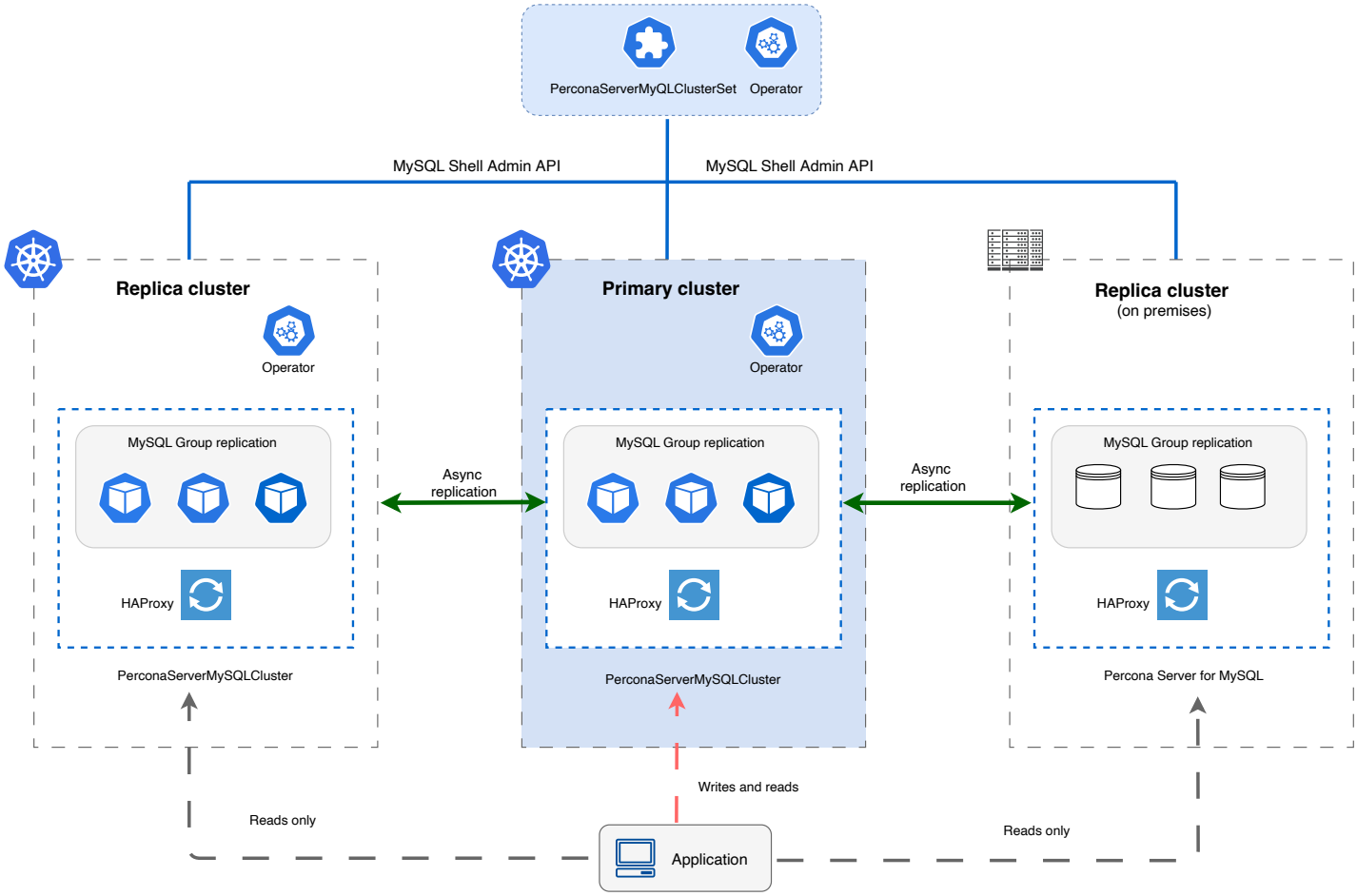
- **Disaster recovery** — Keep a live replica cluster in a second region that you can promote during an outage.
- **Geo-redundancy** — Protect against regional outages by maintaining independent but synchronized clusters in separate geographic locations, ensuring high data availability and business continuity.
- **Read distribution** — Serve regional read traffic from a local replica cluster while writes stay on the primary cluster.
- **Deployment flexibility** — Run your replica cluster in the same Kubernetes cluster, a different cluster, or outside Kubernetes entirely.

Architecture

Cross-site replication is implemented via a separate `PerconaServerMySQLClusterSet` Custom Resource.

You declare which clusters participate, which one is the primary, and how replicas receive the data when they join the ClusterSet in the `PerconaServerMySQLClusterSet` Custom Resource manifest. The Operator automates the ClusterSet provisioning and lifecycle management in a similar way it does for a standalone Percona Server for MySQL cluster. It bootstraps the ClusterSet, adds or removes replica sites, refreshes status, and runs switchover when you change the primary.

The Operator uses a dedicated `clusterset` system user with the grants required for ClusterSet AdminAPI operations. Each cluster must share this user's credentials.



- The **primary cluster** accepts writes and replicates changes asynchronously to replica clusters over a dedicated replication channel.
- Each **replica cluster** is a full Group Replication cluster (read-only at the ClusterSet level) that receives async replication from the primary cluster's PRIMARY member.
- The **ClusterSet controller** (in the Operator Pod) reconciles the `PerconaServerMySQLClusterSet` Custom Resource, manages the `mysqlshell-runner` Pod, and runs long operations (such as `createReplicaCluster`) as Kubernetes Jobs.

The ClusterSet controller is a **pure orchestrator**: it does not manage Pods in remote clusters. It only needs network reachability to MySQL endpoints and the ClusterSet Custom Resource in one Kubernetes namespace.

The communication in the ClusterSet is done through the MySQL protocol, which enables linking replica clusters deployed in the same or a different Kubernetes environment, or outside Kubernetes on-premises. All replicas must be reachable over the network.

The Operator starts a separate `mysqlshell-runner` Pod with **MySQL Shell (`mysqlsh`)** and uses it to manage the ClusterSet. The `mysqlsh` version must be compatible with the Percona Server for MySQL version you run in your clusters.

Data recovery modes

When a replica cluster joins the ClusterSet, it must receive the data from the primary. This can be done in two ways:

- Using the `clone` recovery method (default) — A replica cluster to join the ClusterSet must be without any data. MySQL makes a physical snapshot of the full dataset using the `CLONE` plugin on the primary cluster and transfers it to the replica.
- Using the `incremental` method — Restore the data from the primary cluster on the replica before adding it to the ClusterSet. When it joins the ClusterSet, it catches up the binlog changes that have occurred on the primary.

> **Important**
>
> Binlogs must exist on the primary cluster for the incremental recovery mode to work.

When to use each recovery mode

Recovery mode	Usage
CLONE	For small datasets and low-latency networks
Incremental	WAN links and multi-TB datasets where a full online clone can take a long time and is expensive to retry. Requires restoring the backup from the primary on the replica before adding it to the ClusterSet

TLS for replication channels

Cross-site replication data flows over asynchronous replication channels between the primary and replica clusters. To secure those channels, configure TLS as follows:

1. **Enable TLS on each participating cluster** — Set up transport encryption on every `PerconaServerMySQL` cluster as described in Transport Layer Security (TLS). Each site uses its own certificates and Secrets.
2. **Set `spec.sslMode` on the ClusterSet CR** — This tells MySQL Shell how to configure TLS on the ClusterSet replication channels.

Supported SSL modes

- `Auto` (default) – use TLS when the cluster supports it
- `DISABLED` - disable TLS encryption for the ClusterSet replication channels.
- `REQUIRED` - enable TLS encryption for the ClusterSet replication channels.
- `VERIFY_CA` - enable TLS encryption for the ClusterSet `REQUIRED`, and additionally verify the peer server TLS certificate against the configured Certificate Authority (CA) certificates. Server certificates do not need to be identical, but they must chain to a **mutually trusted CA** (in practice, often the same CA on both sides)
- `VERIFY_IDENTITY` - same CA trust level as for `VERIFY_CA` mode, plus the **primary** server's certificate must list the endpoint hostname replicas connect to in its SAN.

For cross-region links, prefer `REQUIRED` or a stricter mode over the default `AUTO`.

Availability

Requirement	Supported versions
Operator	Percona Operator for MySQL **1.2.0** and later
Percona Server for MySQL	**8.0.27** and later, **8.4** (tested with 8.0.46-37.1 and 8.4.10-10.1)
MySQL Shell	`mysqlsh` version must match MySQL endpoints it talks to
Cluster type	`group-replication` only
Kubernetes	Same platforms as the Operator — see Versions compatibility

Requirements

Before you create a ClusterSet, ensure the following:

- **Network connectivity** — ClusterSet members are linked by the network address, not Kubernetes references. Clusters can live in different Kubernetes clusters, on-premises or in another cloud but they must be reachable and managed by the Operator.
- **Matching system user credentials** — All system user credentials must be identical across every cluster in the ClusterSet. The primary's user table is cloned to replicas during `createReplicaCluster`, so credentials cannot differ per site.
- **Unique endpoint hosts** — Each `host` value must be unique across all clusters in the ClusterSet (CEL-validated at admission). You can use Services or load balancers as endpoints.
- **Group replication-compatible configuration** — Each cluster must be of group replication type and have the CLONE plugin loaded, `gtid_mode=ON`, `enforce_gtid_consistency=ON`, and other InnoDB ClusterSet instance requirements. The group replication configuration of Percona Server for MySQL already includes these settings by default.
- **Clean replicas** — Replica clusters must either be clean or have the dataset restored from the primary before joining the ClusterSet.

Implementation specifics

- **MySQL versioning** — Primary and replicas can have different but compatible MySQL versions. Replication is supported from one release series to the next higher series. For example, from the primary running Percona Server for MySQL 8.0.27+ to the replica running Percona Server for MySQL 8.4.x.
- **Manual bootstrap for replicas** — When you deploy a replica site, specify `spec.mysql.bootstrap.mode: manual` to prevent the replica from starting as a Group Replication cluster. Instead, Pod-0 comes up and waits for the ClusterSet controller to create the group. Pod-1+ do not start until Pod-0 is Ready.
- **Long operations as Jobs** — `createReplicaCluster`, switchover, and cluster removal run as Kubernetes Jobs so the reconcile loop is not blocked for hours during CLONE.
- **Finalizer on delete** — The `percona.com/clusterset-dissolve` finalizer ensures that the ClusterSet dissolves when the `PerconaServerMySQLClusterSet` Custom Resource is removed. Data in underlying clusters is preserved.
- **No automatic switchover** — All switchovers are user-initiated. You must edit the ClusterSet Custom Resource for a planned or a forced failover.

Known limitations

- **No automatic rejoin to ClusterSet after replication stops** — If replication on a ClusterSet replica is interrupted (for example, if a cluster is paused or stopped), the Operator does not automatically rejoin it to the ClusterSet when it starts. You must restore replication manually. Determine the safe window, connect to the replica cluster and run the `dba.rebootClusterFromCompleteOutage()` command to re-initialize the cluster metadata. Then connect to the primary cluster and run the `dba.getCluster().getClusterSet().rejoinCluster("<innodb_cluster_name>")` command to rejoin the replica.
- **No adopting existing ClusterSets** — The controller only bootstraps a new ClusterSet or refuses if the metadata is inconsistent. You cannot adopt into management any ClusterSet that was created manually with `mysqlsh`.
- **Restoring backups onto replica clusters in a ClusterSet is not supported** — While a replica is part of a ClusterSet, restoring a backup onto it is not supported. If you attempt to do so, the replica will fail to rejoin the ClusterSet afterward. In this case, you must manually reboot the cluster (using `dba.rebootClusterFromCompleteOutage()`) and rejoin it to the ClusterSet from the primary.
- **Backups taken on a ClusterSet cannot be restored on a fresh cluster** — Backups include the ClusterSet metadata that is tied to the specific topology and identities of the clusters within that ClusterSet. Restoring such a backup onto a new, unrelated cluster will lead to inconsistencies or errors, since the metadata will not match the new environment or cluster configuration.
- **Removal is one-way** — After a cluster is removed from a ClusterSet, it cannot be added back to the same ClusterSet.
- **Deletion of ClusterSet object is stuck if the cluster is deleted beforehand** - Ensure your ClusterSet members are running before you delete the ClusterSet object. If the deletion is stuck, edit the ClusterSet object and remove the `percona.com/clusterset-dissolve` finalizer to proceed.
- **Credential rotation on replicas** — The `clusterset` user password is always replicated from the primary; you cannot rotate it independently on replica clusters while they are ClusterSet members.
- **Minimum replica size** — A replica Group Replication cluster still needs enough members for local high availability (typically 3). Single-node replicas are supported for testing but not recommended for production.

Deployment

For step-by-step instructions, see Cross-site replication: setup and use.

Cross-site replication: setup and use

This guide walks you through creating a MySQL InnoDB ClusterSet, verifying replication, performing switchover and failover, and removing replica clusters or the entire ClusterSet.

In this setup both source and replica clusters run Percona Server for MySQL version 8.4 in Kubernetes. Refer to Implementation specifics section for more information about running different Percona Server for MySQL versions and their compatibility for replication.

Before you start, read Cross-site replication for architecture, requirements, and limitations.

Prerequisites

- Percona Operator for MySQL **1.2.0** or later installed
- Network connectivity between all MySQL endpoints
- `mysqlsh` version for each cluster must match the MySQL major version it runs
- `yq` utility for parsing YAML or JSON outputs from Kubernetes commands

Prepare your environment

1. Clone the repository with all manifests and source code by executing the following command:

```
git clone -b v1.2.0 https://github.com/percona/percona-server-mysql-operator
```

2. Export the namespaces of the source and replica clusters:

```
export SOURCE_NS=source
export REPLICA_NS=replica
```

3. Create namespaces:

```
kubectl create namespace source
kubectl create namespace replica
```

Step 1. Deploy the primary cluster

1. Follow the quickstart guide to install the Operator deployment, if you haven't done it, and deploy Percona Server for MySQL cluster with replication type `group replication`. Let's rename the cluster to `source-cluster` to differentiate them.

   Here's the example configuration:

   ```
   apiVersion: ps.percona.com/v1
   kind: PerconaServerMySQL
   metadata:
     name: source-cluster
   spec:
     crVersion: 1.2.0
     secretsName: source-cluster-secrets
     mysql:
       clusterType: group-replication
       size: 3
       image: percona/percona-server:8.4.10-10.1
     proxy:
       haproxy:
         enabled: true
         size: 3
     # The rest of the configuration
   ```

2. Apply the configuration and wait until the cluster is ready:

   ```
   kubectl get ps source-cluster -n $SOURCE_NS
   ```

3. Note the InnoDB cluster name from status — you need it for the ClusterSet CR:

   ```
   kubectl get ps source-cluster -n $SOURCE_NS -o
   jsonpath='{.status.innodbClusterName}{"\n"}'
   ```

   Sample output

   ```
   sourcecluster
   ```

4. Note a reachable endpoint for the primary cluster (Service DNS name, load balancer hostname, or external address). We will use the Primary service.

```
kubectl get services -n $SOURCE_NS
```

Sample output

```
NAME                          TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
source-cluster-haproxy         ClusterIP   34.118.234.124   <none>
3306/TCP,3307/TCP,3309/TCP,33060/TCP,33062/TCP    26m
source-cluster-mysql           ClusterIP   None             <none>
3306/TCP,33062/TCP,33060/TCP,6450/TCP,33061/TCP   26m
source-cluster-mysql-primary   ClusterIP   34.118.232.78    <none>
3306/TCP,33062/TCP,33060/TCP,6450/TCP,33061/TCP   26m
source-cluster-mysql-proxy     ClusterIP   None             <none>
3306/TCP,33062/TCP,33060/TCP,6450/TCP,33061/TCP   26m
source-cluster-mysql-unready   ClusterIP   None             <none>
3306/TCP,33062/TCP,33060/TCP,6450/TCP,33061/TCP   26m
```

5. All replica sites must have the same system user credentials as the primary site. To achieve this, export the Secrets object with the user credentials.

   - List the Secrets:

     ```
     kubectl get secrets -n $SOURCE_NS
     ```

     Sample output

     ```
     internal-source-cluster      Opaque              8      25m
     source-cluster-psuser-root   Opaque              11     25m
     source-cluster-secrets       Opaque              8      25m
     source-cluster-ssl           kubernetes.io/tls   3      25m
     ```

     The Secret with user credentials is `<cluster-name>-secrets`

   - Export the Secret:

     ```
     kubectl get secret source-cluster-secrets -n $SOURCE_NS -o yaml >
     source-secret.yaml
     ```

   - Edit the file. Remove the `annotations`, `creationTimestamp`, `resourceVersion`, `selfLink`, and `uid` metadata fields from the resulting file to make it ready for the replica site. Also change the `namespace` to the namespace of your replica site.

     Use the following scripts:

```
yq eval 'del(.metadata.ownerReferences, .metadata.annotations,
.metadata.creationTimestamp, .metadata.resourceVersion,
.metadata.selfLink, .metadata.uid)' source-secret.yaml > replica-
secret.yaml
yq eval '.metadata.namespace = "replica"' -i replica-secret.yaml
sed -i '' 's/source-cluster/replica-cluster/g' replica-secret.yaml
```

Step 2. Deploy the replica cluster

1. Follow the quickstart guide to install the Operator deployment in the `replica` namespace.
2. Create the Secret from the secret file you prepared from the primary cluster:

   ```
   kubectl apply -f replica-secret.yaml -n $REPLICA_NS
   ```

3. Prepare the Replica cluster configuration:

   - Set `spec.mysql.bootstrap.mode` to `manual` so Pod-0 does not form a Group Replication group until the ClusterSet adopts it
   - Reference the Secret you created in the `spec.secretsName`

   Here's the example configuration:

   cr-replica.yaml

   ```
   apiVersion: ps.percona.com/v1
   kind: PerconaServerMySQL
   metadata:
     name: replica-cluster
   spec:
     crVersion: 1.2.0
     secretsName: replica-cluster-secrets
     mysql:
       clusterType: group-replication
       size: 3
       image: percona/percona-server:8.4.10-10.1
       bootstrap:
         mode: manual
   ```

4. Apply the configuration to create the replica cluster:

   ```
   kubectl apply -f deploy/cr-replica.yaml -n $REPLICA_NS
   ```

After you apply the Custom Resource manifest, Pod-0 starts but stays **NotReady** — this is expected. The cluster reports the `Initializing` state and the `AwaitingExternalBootstrap` condition appears in its status. Pod-1 and Pod-2 do not start until Pod-0 joins a Group Replication group.

5. Verify the cluster status:

```
kubectl get ps replica-cluster -n $REPLICA_NS
kubectl get pods -l app.kubernetes.io/instance=replica-cluster -n
$REPLICA_NS
```

6. Note the InnoDB cluster name from status — you need it for the ClusterSet CR:

```
kubectl get ps replica-cluster -n $REPLICA_NS -o
jsonpath='{.status.innodbClusterName}{"\n"}'
```

Sample output

```
replicacluster
```

7. Note a reachable endpoint for the replica cluster. We will use the Primary service:

```
kubectl get service -n $REPLICA_NS
```

Sample output

```
NAME                          TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
replica-cluster-mysql         ClusterIP   None             <none>
3306/TCP,33062/TCP,33060/TCP,6450/TCP,33061/TCP   5m31s
```

Step 3. Restore the data on the replica cluster from source (optional)

By default, the replica cluster receives the data from source via the `clone` recovery method - the Operator uses `mysqlshell` to create a physical snapshot of the dataset from the source and transfer it to the replica.

For large datasets, a full clone can last long, add load to the `Donor` node and be expensive to retry if something goes wrong.

Instead of using the `clone` method, make a backup on the primary cluster and restore it on the replica. Then when the replica joins the ClusterSet, it already has the data and the GTID history from the source. So it only receives the binlog changes.

Refer to the make a backup and restore from a backup on a new cluster tutorials for step-by-step instructions.

Step 4. Create the ClusterSet

Now it's time to link clusters. To do this, configure a `PerconaServerMySQLClusterSet` Custom Resource.

1. Modify the deploy/clusterset.yaml template and specify the following:
   - `name` - the name of the ClusterSet
   - `primaryCluster` - the InnoDB name of the primary cluster
   - `credentialsSecret.name` - the name of the Secret with the `clusterset` user credentials
   - `createReplicaClusterOptions.recoveryMethod` — leave `clone` for clean replicas. Change to `incremental` if you restored the data on the replica from the source.
   - `mysqlshellRunner` — defines the helper Pod image used by the Operator to run MySQL Shell operations. The version must be compatible with the MySQL version you run in your clusters.
   - `clusters` - the list of the ClusterSet members. For each cluster specify its InnoDBCluster name and the endpoint that the Operator will use to reach it.

   Here's the example configuration:

   Clone recovery

```
apiVersion: ps.percona.com/v1
kind: PerconaServerMySQLClusterSet
metadata:
  name: my-cluster-set
  finalizers:
    - percona.com/clusterset-dissolve
spec:
  primaryCluster: sourcecluster
  credentialsSecret:
    name: source-cluster-secrets
```

```
    key: clusterset
  sslMode: AUTO
  createReplicaClusterOptions:
    recoveryMethod: clone
  mysqlshellRunner:
    image: percona/percona-server:8.4.10-10.1
  clusters:
    - innodbClusterName: sourcecluster
      endpoints:
        - host: source-cluster-mysql-primary.source.svc.cluster.local
          port: 3306
    - innodbClusterName: replicacluster
      endpoints:
        - host: replica-cluster-mysql-0.replica-cluster-
mysql.replica.svc.cluster.local
          port: 3306
```

Incremental recovery

```
apiVersion: ps.percona.com/v1
kind: PerconaServerMySQLClusterSet
metadata:
  name: my-cluster-set
  finalizers:
    - percona.com/clusterset-dissolve
spec:
  primaryCluster: sourcecluster
  credentialsSecret:
    name: source-cluster-secrets
    key: clusterset
  sslMode: AUTO
  createReplicaClusterOptions:
    recoveryMethod: incremental
  mysqlshellRunner:
    image: percona/percona-server:8.4.10-10.1
  clusters:
    - innodbClusterName: sourcecluster
      endpoints:
        - host: source-cluster-mysql-primary.source.svc.cluster.local
          port: 3306
    - innodbClusterName: replicacluster
      endpoints:
        - host: replica-cluster-mysql-0.replica-cluster-
mysql.replica.svc.cluster.local
          port: 3306
```

2. Apply the manifest:

```
kubectl apply -f clusterset.yaml -n $SOURCE_NS
```

After you apply the manifest, the controller:

a. Creates the `mysqlshell-runner` Pod

b. Bootstraps the ClusterSet on the primary with `dba.getCluster().createClusterSet()`

c. Starts a Job to run `createReplicaCluster()` for the replica with the specified recovery mode

d. Updates status as replication becomes healthy

3. Monitor progress:

```
kubectl get ps-clusterset my-cluster-set -n $SOURCE_NS
kubectl get ps-clusterset my-cluster-set -n $SOURCE_NS -o yaml
kubectl get jobs -n $SOURCE_NS | grep my-cluster-set
kubectl logs -l job-name=<job-name> -n $SOURCE_NS
```

When the Job completes, Pod-0 on the replica cluster becomes Ready and Pod-1, Pod-2 join the local Group Replication group automatically.

Step 5. Verify replication

Check ClusterSet status

1. Check ClusterSet status:

```
kubectl get ps-clusterset my-cluster-set -n $SOURCE_NS
```

Sample output

```
NAME             PRIMARY     ENDPOINT                              READY   AGE
my-cluster-set   sourcecluster  source-cluster-mysql-0:3306              True
15m
```

2. Inspect full status:

```
kubectl describe ps-clusterset my-cluster-set -n $SOURCE_NS
```

Confirm that:

- `status.conditions` includes `Ready: True` with reason `ClusterSetHealthy`
- `status.clusters` shows each member with `clusterRole` (`PRIMARY` or `REPLICA`) and `globalStatus` (`OK` for healthy members)
- `status.primaryCluster` matches your desired primary

Check per-cluster status

Run on the replica cluster:

```
kubectl get ps replica-cluster -n $REPLICA_NS -o jsonpath='{range
.status.conditions[*]}{.type}{": "}{.status}{"\n"}{end}'
```

Expected output

```
Initializing: False
Ready: True
ClusterSetReplicationRunning: True
InnoDBClusterBootstrapped: True
```

Verify data replication

1. Connect to the primary cluster and insert some data:

   a. Identify the primary Pod:

   ```
   export PRIMARY=$(kubectl get pods -n $SOURCE_NS -l
   mysql.percona.com/primary=true -o jsonpath='{.items[0].metadata.name}')
   ```

   b. Connect to the primary Pod:

   ```
   kubectl exec -it $PRIMARY -n $SOURCE_NS -c mysql -- \
     mysql -uroot -p$(kubectl get secret source-cluster-secrets -n
   $SOURCE_NS -o jsonpath='{.data.root}' | base64 -d)
   ```

   c. Write some data:

```
CREATE DATABASE IF NOT EXISTS test;
CREATE TABLE IF NOT EXISTS test.t1 (id INT PRIMARY KEY); INSERT INTO
test.t1 VALUES (1);
```

2. Connect to the replica cluster:

```
kubectl exec -it replica-cluster-mysql-0 -n $REPLICA_NS -c mysql -- \
  mysql -uroot -p$(kubectl get secret replica-cluster-secrets -n
$REPLICA_NS -o jsonpath='{.data.root}' | base64 -d)
```

3. Verify that the data is replicated by querying it:

```
SELECT * FROM test.t1;
```

   You should see the same data that you inserted.

Usage scenarios

Planned switchover

To promote a replica cluster to primary, edit `spec.primaryCluster`:

```
kubectl patch ps-clusterset my-cluster-set -n $SOURCE_NS \
  --type=merge -p '{"spec":{"primaryCluster":"replicacluster"}}'
```

The controller:

1. Verifies both clusters are reachable
2. Runs `setPrimaryCluster('replicacluster')` via a Job
3. Sets `SwitchoverInProgress` condition while the Job runs
4. Updates `status.primaryCluster` when complete

To monitor the progress, run:

```
kubectl get ps-clusterset my-cluster-set -n $SOURCE_NS -w
kubectl get jobs -n $SOURCE_NS | grep switchover
```

After switchover, writes go to the new primary cluster. The former primary becomes a replica.

Forced failover

Use forced failover only when the current primary cluster is **unreachable** and you accept the risk of split-brain or lost transactions if the old primary is still alive.

1. Confirm the primary is unreachable from the controller's perspective (the `ErrorReconcile: True` condition with the reason `PrimaryUnreachable` may be set).
2. Enable forced failover and change the primary:

```
kubectl patch ps-clusterset my-cluster-set -n $SOURCE_NS --type=merge -p '{
  "spec": {
    "primaryCluster": "replicacluster",
    "unsafeFlags": {
      "forcedFailover": true
    }
  }
}'
```

The controller runs `forcePrimaryCluster()`. The old primary cluster is marked **INVALIDATED** in ClusterSet metadata if it comes back online.

> **Warning**
>
> Set `unsafeFlags.forcedFailover` to `true` only when you are certain the old primary cannot recover or accept writes.

To recover an invalidated cluster later, either use MySQL Shell `rejoinCluster()` manually, or remove and recreate the cluster in the ClusterSet.

Remove a replica cluster

Remove a replica by deleting its entry from `spec.clusters[]`.

> **Important**
>
> Removal is **one-way**. You cannot add the same cluster back to the same ClusterSet after removal.

Run the following command to remove a cluster:

Replica is running (reachable)

Remove the replica by patching the ClusterSet:

```
kubectl patch ps-clusterset my-cluster-set -n $SOURCE_NS --type=json \
  -p='[{"op": "remove", "path": "/spec/clusters/1"}]'
```

Adjust the index in `"path": "/spec/clusters/1"` to match the cluster you want to remove.

The controller:

1. Runs `removeCluster()` for the removed member
2. Dissolves Group Replication on that cluster
3. Lets the per-site Operator re-bootstrap it as a standalone InnoDB Cluster

Replica is offline or unreachable

If the replica cluster is **offline or unreachable**, simply deleting its entry from `spec.clusters[]` is not enough, as this will not remove it. You must enable forced removal by also setting the `spec.unsafeFlags.forcedClusterRemoval` option to `true`:

```
kubectl patch ps-clusterset my-cluster-set -n $SOURCE_NS --type=json -p='[
  {"op": "remove", "path": "/spec/clusters/1"},
  {"op": "add", "path": "/spec/unsafeFlags/forcedClusterRemoval", "value":
true}
]'
```

Warning

Forced cluster removal abandons any unreplicated transactions on the removed cluster. The cluster may require manual cleanup or a full rebuild.

If you deleted an unreachable replica without the `forcedClusterRemoval`

If you try to remove an offline replica **without** the `forcedClusterRemoval` flag, the removal Job fails. Simply updating the ClusterSet resource to specify `unsafeFlags.forcedClusterRemoval: true` will **not** automatically resume or restart the failed job.

In this case, you must first **delete the failed Job** before retrying:

1. Find the removal job:

```
kubectl get jobs -n $SOURCE_NS | grep clusterset-remove
```

2. Delete the failed removal job:

```
kubectl delete job <remove-job-name> -n $SOURCE_NS
```

3. Retry the patch command to remove the cluster:

```
kubectl patch ps-clusterset my-cluster-set -n $SOURCE_NS --type=json \
  -p='[{"op": "remove", "path": "/spec/clusters/1"}]'
```

Only then the Operator recognizes the unsafe flag and retries forced removal.

Delete the ClusterSet

Delete the ClusterSet Custom Resource to stop cross-site replication and dissolve ClusterSet metadata:

```
kubectl delete ps-clusterset my-cluster-set -n $SOURCE_NS
```

The `percona.com/clusterset-dissolve` finalizer:

1. Waits for any in-flight `createReplicaCluster` Job to finish
2. Runs `.dissolve()` on the InnoDB ClusterSet
3. Removes the Custom Resource
4. Updates the `spec.bootstrap.mode` on the replica cluster to `auto`.

Underlying `PerconaServerMySQL` clusters **continue running** as standalone InnoDB Clusters. The per-site Operator keeps managing them.

If dissolve fails (for example, because the primary is permanently unreachable), the Custom Resource stays with `deletionTimestamp` set. Fix connectivity or resolve the MySQL-side issue, then the finalizer retries automatically.

To delete replica and primary clusters themselves, delete their `PerconaServerMySQL` Custom Resources separately after the ClusterSet is gone.

Troubleshooting

Symptom	What to check
`ClusterSetBootstrapped: False`	Primary endpoint unreachable; verify DNS, firewall, and TLS
`ReplicaManagementFailure: True`	Inspect the Job logs: `kubectl logs job/<job-name>`
Replica Pod-0 stays NotReady	ClusterSet Job still running, or `createReplicaCluster` failed
`Ready: False`, reason `ReplicaNotStandalone`	Target cluster is already in another InnoDB Cluster or ClusterSet
Switchover stuck	Check `SwitchoverInProgress` condition and switchover Job status
`ErrorReconcile: True`, reason `AccessDenied`	Incorrect password configured on the replica site
`ErrorReconcile: True`, reason `PrimaryUnreachable`	Primary cluster is not reachable
`ReplicaManagementFailure`	One or more replicas could not be added or removed. See the condition message for exact details. Make sure that your replicas are reachable before removing them.

For status field reference, see Custom resource statuses.

Define environment variables

Configure environment variables

You can configure environment variables in Percona Operator for MySQL based on Percona Server for MySQL for the following purposes:

1. **Configure Operator environment variables** — Control the Operator's behavior, such as logging, telemetry, and which namespaces the Operator watches. You set these on the Operator Deployment (`deploy/operator.yaml` or equivalent).
2. **Define environment variables for cluster components** — Set variables for database and proxy Pods through the Custom Resource (`deploy/cr.yaml`): use **supported tuning names** (for example bootstrap or HAProxy timeouts) or **arbitrary custom** names, via `env` / `envFrom` and optional ConfigMaps or Secrets.

When to use environment variables

Type	Use cases
Operator environment variables	- Control logging for debugging and log aggregation. - Manage telemetry. - Configure the namespaces for the Operator to watch.
Cluster component environment variables	- Use documented names to tune bootstrap, HAProxy, Router, Orchestrator, and backups. - Pass any extra variables your containers need (for example `TZ`). - All of this is configured in the **Custom Resource**, not on the Operator Deployment.

For field-level details, see Custom Resource options.

Configure Operator environment variables

You can configure the Percona Operator for MySQL by setting environment variables on the Operator Deployment. This lets you tune Operator behavior without rebuilding images.

You can set environment variables in the following ways:

- For installations with `kubectl`, edit the Operator Deployment manifest (`deploy/bundle.yaml` / `deploy/operator.yaml` or `deploy/bundle-cw.yaml` / `deploy/operator-cw.yaml` in the percona-server-mysql-operator repository) before you apply it. You can also change the existing Deployment with `kubectl patch` or `kubectl edit`.
- For Helm installations, you can set environment variables through Helm values. Use the ps-operator chart values.
- For OpenShift installations that use OLM, configure environment variables in the Subscription.

Available environment variables

`LOG_STRUCTURED`

Controls whether Operator logs are structured (JSON) or plain text.

Value type	Default	Example
string	`"false"`	`"true"`

Example configuration:

```
env:
  - name: LOG_STRUCTURED
    value: "true"
```

Structured logs work well with tools such as jq. See also Check the logs.

`LOG_LEVEL`

Sets the verbosity of Operator logs.

Value type	Default	Example

string	`INFO`	`DEBUG`

Valid values are:

- `VERBOSE` or `DEBUG` — Most verbose; `VERBOSE` also enables additional diagnostic output in some code paths.
- `ERROR` — Error messages only.
- `INFO` — Standard informational messages (default).

Any other value falls back to `INFO` with a message in the log.

Example configuration:

```
env:
  - name: LOG_LEVEL
    value: DEBUG
```

`WATCH_NAMESPACE`

Specifies which namespaces the Operator watches for `PerconaServerMySQL` and related custom resources.

By default, the value is set to the Operator's own namespace from the metadata.namespace option via a downward API `fieldRef`:

```
- name: WATCH_NAMESPACE
  valueFrom:
    fieldRef:
      apiVersion: v1
      fieldPath: metadata.namespace
```

Value type	Default	Example
string	See below	`ns-one,ns-two` or `""`

Accepted values:

- If set to a comma-separated list, the Operator watches those specific namespaces. The namespace list must include the namespace where the Operator itself is deployed. Use this

approach for the multi-namespace deployment.

- If set to an empty string (`""`), the Operator watches all namespaces.

When you deploy the Operator in cluster-wide mode, it should be associated with the appropriate ClusterRole.

Example configuration:

```
env:
 - name: WATCH_NAMESPACE
   value: "mysql,mysql-dev,mysql-prod"
```

`MAX_CONCURRENT_RECONCILES`

Sets the number of concurrent workers that reconcile `PerconaServerMySQL` and related custom resources in parallel. Useful when you manage multiple clusters with a single Operator. See Configure concurrency for a cluster reconciliation.

Value type	Default	Example
string	`"1"`	`"2"`

The value must be a positive integer. The Operator fails to start if the value is not a valid integer or is less than or equal to 0.

Example configuration:

```
env:
  - name: MAX_CONCURRENT_RECONCILES
    value: "2"
```

`DISABLE_TELEMETRY`

Disables anonymous telemetry data collection by the Operator. For what is collected when telemetry is enabled, see Telemetry.

Value type	Default	Example
string	`"false"`	`"true"`

Example configuration:

```
env:
  - name: DISABLE_TELEMETRY
    value: "true"
```

Update environment variables

Using `kubectl patch`

You can update environment variables in an existing Operator Deployment by applying a patch. To keep existing environment variables, you must specify the full list of them.

Here's how to do it:

1. Get the current environment variables:

   ```
   kubectl get deployment percona-server-mysql-operator -n <operator-
   namespace> -o jsonpath='{.spec.template.spec.containers[0].env}' | jq
   ```

2. Update the deployment. This example command keeps downward API for `WATCH_NAMESPACE` and sets `LOG_LEVEL` to `DEBUG`:

   ```
   kubectl patch deployment percona-server-mysql-operator -n <operator-
   namespace> \
     --type='json' \
     -p='[{"op": "replace", "path": "/spec/template/spec/containers/0/env",
   "value": [
       {"name": "LOG_STRUCTURED", "value": "false"},
       {"name": "LOG_LEVEL", "value": "DEBUG"},
       {"name": "WATCH_NAMESPACE", "valueFrom": {"fieldRef": {"apiVersion":
   "v1",      "fieldPath": "metadata.namespace"}}},
       {"name": "MAX_CONCURRENT_RECONCILES", "value": "1"},
       {"name": "DISABLE_TELEMETRY", "value": "false"}
     ]}]'
   ```

Adjust the list to match your current Deployment (for example cluster-wide `WATCH_NAMESPACE` with a string `value` instead of `valueFrom`).

Using `kubectl edit`

You can also edit the Deployment directly:

```
kubectl edit deployment percona-server-mysql-operator -n <operator-namespace>
```

Then modify the env section in the container specification, save, and exit. Kubernetes rolls out a new ReplicaSet for the Operator Pod.

After you change environment variables, the Operator Pod restarts with the new settings.

Define environment variables for cluster components

Tuning through environment variables helps when you need to:

- **Change timeouts and limits** for bootstrap, replication, or health checks without building a new image.
- **Match your platform** using standard names such as time zone or locale.
- **Feed flags or paths** into scripts and binaries that already read `getenv`-style configuration.
- **Keep secrets out of the CR** by loading keys from a Kubernetes Secret instead of plain YAML.

You declare this configuration in the Custom Resource. The Operator reconciles the cluster and updates Pod specs so containers receive the variables you specify.

How to tune cluster components

You can tune cluster components in the following ways:

- Using existing environment variables
- Passing custom environment variables

You use the same Custom Resource fields for either way: `env` for explicit name/value pairs, and `envFrom` to load many variables from a ConfigMap or Secret. All values must be strings. Note that default values or behaviors may change between Operator versions. Check the sample `deploy/cr.yaml` for reference.

You can combine both approaches for a single component: use existing environment variables where available, and add your own custom names as needed.

Existing environment variables for cluster components

Percona Server for MySQL environment variable

The following environment variables are available to tune Percona Server for MySQL

Variable	Description

`BOOTSTRAP_READ_TIMEOUT`	Non-negative integer (seconds). Read/write timeout for bootstrap and related MySQL client operations.
`BOOTSTRAP_CLONE_TIMEOUT`	Non-negative integer (seconds). Upper bound for clone operations during asynchronous bootstrap.
`ASYNC_SOURCE_RETRY_COUNT`	Non-negative integer. Used when configuring asynchronous replication.
`ASYNC_SOURCE_CONNECT_RETRY`	Non-negative integer. Used when configuring asynchronous replication.

HAProxy environment variables

The following environment variables are available for HAProxy

Variable	Description
`HA_CONNECTION_TIMEOUT`	Sets the timeout (in milliseconds) for HAProxy health checks on MySQL nodes. The default is 10000 milliseconds (10 seconds), but you can increase this value for unstable Kubernetes networking or if you experience soft lockups on nodes.
`HA_RLIMIT_NOFILE`	Sets the soft file descriptor limit in the entrypoint before HAProxy container starts. The default value is 1048576. If the set value is invalid, the Operator falls back to the default one.
`HA_SERVER_OPTIONS`	Extra server options passed into generated HAProxy server lines (`haproxy_add_mysql_nodes.sh`).

Custom environment variables

Apart from existing environment variables, you can also pass your custom configurations in these ways:

- Set variables directly in the Custom Resource — Under `spec.<component>.env` as a list of `name` and `value` pairs.
- Load variables from a ConfigMap — Under `envFrom` with `configMapRef`; each key becomes an environment variable name.
- Load variables from a Secret — Under `envFrom` with `secretRef`; use for sensitive values.

Supported components

Custom env and envFrom are supported for these components:

Component	Custom Resource options
MySQL	`spec.mysql.env`, `spec.mysql.envFrom`
HAProxy	`spec.proxy.haproxy.env`, `spec.proxy.haproxy.envFrom`
MySQL Router	`spec.proxy.router.env`, `spec.proxy.router.envFrom`
Orchestrator	`spec.orchestrator.env`, `spec.orchestrator.envFrom`
Backup storage (per entry)	`spec.backup.storages.<name>.containerOptions.env`

Full field definitions are in Custom Resource options.

Set variables directly in the Custom Resource

Use this method when you have a small number of non-sensitive values and you want everything in a single file.

For example, you want to override the soft file descriptor limit for HAProxy.

1. Edit the `deploy/cr.yaml` Custom Resource manifest:

```
spec:
  proxy:
    haproxy:
      env:
        - name: HA_CONNECTION_TIMEOUT
          value: "30"
```

2. Apply the changes:

```
kubectl apply -f deploy/cr.yaml -n <namespace>
```

Load variables from a ConfigMap

Use this when you want to share the same variables across multiple clusters or update them without editing the Custom Resource.

1. Export the namespace where your cluster is running as an environment variable. Replace `<my-namespace>` with your value:

   ```
   export NAMESPACE = my-namespace
   ```

2. Create a ConfigMap file. For example, `haproxy-configmap.yaml`. Specify the variables within:

   haproxy-configmap.yaml

   ```
   apiVersion: v1
   kind: ConfigMap
   metadata:
     name: ps-haproxy-env
   data:
     HA_CONNECTION_TIMEOUT: "30"
   ```

3. Create a ConfigMap object:

   ```bash kubectl apply -f haproxy-configmap.yaml -n $NAMESPACE

4. Reference the ConfigMap in the Custom Resource:

   ```
   spec:
     mysql:
       envFrom:
         - configMapRef:
             name: ps-haproxy-env
   ```

5. Apply the changes:

   ```
   kubectl apply -f deploy/cr.yaml -n $NAMESPACE
   ```

Load variables from a Secret

Use Secrets when you need to supply sensitive values (tokens, passwords, keys).

For example, you need to provide a token used by a custom sidecar container.
   ```

1. Encode your sensitive values before adding them to a Secret, as Kubernetes stores Secret data in base64-encoded form. This helps prevent accidental exposure of sensitive information in plaintext, even though it is not a secure encryption method.

   To encode an API token, run:

   ```
   echo -n "your-token" | base64
   ```

   Copy the encoded string for use in your Secret manifest

2. Export the namespace where your cluster is running as an environment variable. Replace my-namespace with your value:

   ```
   export NAMESPACE=my-namespace
   ```

3. Create a Secret configuration file, for example, `integration-token.yaml`. Specify your encoded value within:

   integration-token.yaml

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: ps-app-env
   type: Opaque
   data:
     MY_INTEGRATION_TOKEN: "your-base64-encoded-token"
   ```

4. Create the Secret object:

   ```
   kubectl apply -f integration-token.yaml
   ```

5. Reference the Secret in your Custom Resource:

   ```
   spec:
     mysql:
       envFrom:
         - secretRef:
             name: ps-app-env
   ```

6. Apply the changes:

```
kubectl apply -f deploy/cr.yaml -n $NAMESPACE
```

Combining `env` and `envFrom`

You can use both on one component. If a name appears in multiple sources, Kubernetes resolution rules apply: later explicit `env` entries override earlier values (see the Kubernetes documentation).

Cluster lifecycle management

Upgrade

Upgrade Database and Operator

You can upgrade Percona Operator for MySQL based on Percona Server for MySQL to newer versions.

The upgrade process consists of these steps:

- update the Custom Resource Definition version
- upgrade the Operator
- upgrade the database (Percona Server for MySQL)

Update scenarios

You can either upgrade both the Operator and the database, or you can upgrade only the database. To decide which scenario to choose, read on.

Full upgrade (CRD, Operator, and the database)

When to use this scenario:

- The new Operator version has changes that are required for new features of the database to work
- The Operator has new features or fixes that enhance automation and management.
- Compatibility improvements between the Operator and the database require synchronized updates.

When going on with this scenario, make sure to test it in a staging or testing environment first. Upgrading the Operator may cause performance degradation.

Upgrade only the database

When to use this scenario:

- The new version of the database has new features or fixes that are not related to the Operator or other components of your infrastructure
- You have updated the Operator earlier and now want to proceed with the database update.

When choosing this scenario, consider the following:

- Check that the current Operator version supports the new database version.
- Some features may require an Operator upgrade later for full functionality.

Update strategies

The Operator supports the *Smart Update* strategy. This is the automated way to update the database cluster. The Operator controls how objects are updated. It restarts Pods in a specific order, with the primary instance updated last to avoid connection issues until the whole cluster is updated to the new settings.

This update method applies during database upgrades and when making changes like updating a ConfigMap, rotating passwords, or changing resource values.

Upgrade the Operator

Upgrade the Operator and CRD

To update the Operator, you need to update the Custom Resource Definition (CRD) and the Operator deployment. Also we recommend to update the Kubernetes database cluster configuration by updating the Custom Resource and the database components to the latest version. This step ensures that all new features that come with the Operator release work in your environment.

Considerations for Kubernetes Cluster versions and upgrades

1. Before upgrading the Kubernetes cluster, have a disaster recovery plan in place. Ensure that a backup is taken prior to the upgrade.
2. Plan your Kubernetes cluster or Operator upgrades with version compatibility in mind.

   The Operator is supported and tested on specific Kubernetes versions. Always refer to the Operator's release notes to verify the supported Kubernetes platforms.

   Note that while the Operator might run on unsupported or untested Kubernetes versions, this is not recommended. Doing so can cause various issues, and in some cases, the Operator may fail if deprecated API versions have been removed.
3. During a Kubernetes cluster upgrade, you must also upgrade the `kubelet`. It is advisable to drain the nodes hosting the database Pods during the upgrade process.
4. During the `kubelet` upgrade, nodes transition between `Ready` and `NotReady` states. Also in some scenarios, older nodes may be replaced entirely with new nodes. Ensure that nodes hosting database or proxy pods are functioning correctly and remain in a stable state after the upgrade.
5. Regardless of the upgrade approach, pods will be rescheduled or recycled. Plan your Kubernetes cluster upgrade accordingly to minimize downtime and service disruption.

Considerations for Operator upgrades

1. The Operator version has three digits separated by a dot (`.`) in the format `major.minor.patch`. Here's how you can understand the version `1.0.1`:
   - `1` is the major version
   - `0` is the minor version

- `1` is the patch version.

  You can only upgrade the Operator to the nearest `major.minor` version (for example, from `1.0.0` to `1.1.0`).

  If the current Operator version and the version you want to upgrade to differ by more than one minor version, you need to upgrade step by step. For example, if your current version is `1.0.0` and you want to move to `1.2.0`, first upgrade to `1.1.0`, then to `1.2.0`.

  Check the Release notes index for the list of the Operator versions.

2. CRD supports the **last 3 minor versions of the Operator**. This means it is compatible with the newest Operator version and the two older minor versions.
3. The API version in CRDs is changed from `v1alpha` to `v1`. To update to version 0.12.0, you must manually delete the CRDs, apply new ones and recreate the cluster. To keep the data, do the following:
   - check that the `percona.com/delete-mysql-pvc` finalizer is not enabled in `deploy/cr.yaml`
   - don't delete PVCs manually
   - Recreate the cluster with the same name. The Operator then automatically reuses the same PVCs.

Upgrade guides

Choose the upgrade instructions below based on how you originally deployed the Operator:

Manual upgrade | Upgrade via Helm | Upgrade on OpenShift

Upgrade the Operator and CRD manually

Before you start, export your namespace as an environment variable to simplify the configuration:

```
export NAMESPACE=<my-namespace>
```

The upgrade includes the following steps.

1. Update the Custom Resource Definition for the Operator and the Role-based access control. Take the latest versions from the official repository on GitHub with the following commands:

   ```
   kubectl apply --server-side -f
   https://raw.githubusercontent.com/percona/percona-server-mysql-
   operator/v1.2.0/deploy/crd.yaml
   kubectl apply --server-side -f
   https://raw.githubusercontent.com/percona/percona-server-mysql-
   operator/v1.2.0/deploy/rbac.yaml
   ```

2. Next, update the Percona Server for MySQL Operator Deployment in Kubernetes by changing the container image of the Operator Pod to the latest version. Find the image name for the current Operator release in the list of certified images. Use the following command to update the Operator to the `1.2.0` version:

For single-namespace deployment

Use the following command if you deploy both the Operator and the database cluster in the same namespace:

```
kubectl apply --server-side -f
https://raw.githubusercontent.com/percona/percona-server-mysql-
operator/v1.2.0/deploy/operator.yaml -n $NAMESPACE
```

For cluster-wide deployment

If you deployed the Operator to manage several clusters in different namespaces (the so-called cluster-wide mode), use the following command:

```
kubectl apply --server-side -f
https://raw.githubusercontent.com/percona/percona-server-mysql-
operator/v1.2.0/deploy/cw-operator.yaml -n $NAMESPACE
```

For previous releases, please refer to the old releases documentation archive

3. The deployment rollout will be automatically triggered. You can track the rollout process in real time with the `kubectl rollout status` command with the name of your cluster:

```
kubectl rollout status deployments percona-server-mysql-operator -n
$NAMESPACE
```

> **Note**
>
> Labels set on the Operator Pod will not be updated during upgrade.

4. Update the Custom Resource, the database and components. This step ensures all new features and improvements of the latest release work well within your environment.

Update the Custom Resource, the database and components

Update the Custom Resource, the database, backup, proxy and PMM Client image names with a newer version tag. This step ensures all new features and improvements of the latest release work well within your environment.

Find the image names in the list of certified images.

We recommend to update the PMM Server **before** the upgrade of PMM Client. If you haven't updated your PMM Server yet, exclude PMM Client from the list of images to update.

Since this is a working cluster, the way to update the Custom Resource is to apply a patch with the `kubectl patch ps` command.

Select the command that matches your setup from the sections below.

MySQL 8.4

Asynchronous replication (tech preview)

Asynchronous replication uses HAProxy and Orchestrator. It does not use MySQL Router.

- **With PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "orchestrator":{ "image": "percona/percona-orchestrator:3.2.6-22" },
       "backup":{ "image": "percona/percona-xtrabackup:8.4.0-6.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" },
       "pmm": { "image": "percona/pmm-client:3.8.1" }
   }}'
```

- **Without PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "orchestrator":{ "image": "percona/percona-orchestrator:3.2.6-22" },
       "backup":{ "image": "percona/percona-xtrabackup:8.4.0-6.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

- **With Binlog Server**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "orchestrator":{ "image": "percona/percona-orchestrator:3.2.6-22" },
       "backup":{
          "image": "percona/percona-xtrabackup:8.4.0-6.1",
          "pitr": { "binlogServer": { "image": "perconalab/percona-binlog-
server:0.3.1" } }
       },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

After the upgrade, enable point-in-time binlog collection.

Group replication with HAProxy

Group replication with HAProxy uses HAProxy only. It does not use MySQL Router or Orchestrator.

- **With PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.4.0-6.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" },
       "pmm": { "image": "percona/pmm-client:3.8.1" }
   }}'
```

- **Without PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.4.0-6.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

- **With Binlog Server**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "backup":{
          "image": "percona/percona-xtrabackup:8.4.0-6.1",
          "pitr": { "binlogServer": { "image": "perconalab/percona-binlog-
server:0.3.1" } }
       },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

After the upgrade, enable point-in-time binlog collection.

Group replication with MySQL Router

Group replication with MySQL Router uses MySQL Router only. It does not use HAProxy or Orchestrator.

- **With PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "router":{ "image": "percona/percona-mysql-router:8.4.10-10.1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.4.0-6.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" },
       "pmm": { "image": "percona/pmm-client:3.8.1" }
   }}'
```

- **Without PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "router":{ "image": "percona/percona-mysql-router:8.4.10-10.1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.4.0-6.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

- **With Binlog Server**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "router":{ "image": "percona/percona-mysql-router:8.4.10-10.1" }
       },
       "backup":{
          "image": "percona/percona-xtrabackup:8.4.0-6.1",
          "pitr": { "binlogServer": { "image": "perconalab/percona-binlog-
server:0.3.1" } }
       },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

Asynchronous replication (tech preview)

Asynchronous replication uses HAProxy and Orchestrator. It does not use MySQL Router.

- **With PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "orchestrator":{ "image": "percona/percona-orchestrator:3.2.6-22" },
       "backup":{ "image": "percona/percona-xtrabackup:8.0.35-36.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" },
       "pmm": { "image": "percona/pmm-client:3.8.1" }
   }}'
```

- **Without PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "orchestrator":{ "image": "percona/percona-orchestrator:3.2.6-22" },
       "backup":{ "image": "percona/percona-xtrabackup:8.0.35-36.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

- **With Binlog Server**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "orchestrator":{ "image": "percona/percona-orchestrator:3.2.6-22" },
       "backup":{
          "image": "percona/percona-xtrabackup:8.0.35-36.1",
          "pitr": { "binlogServer": { "image": "perconalab/percona-binlog-
server:0.3.1" } }
       },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

After the upgrade, enable point-in-time binlog collection.

Group replication with HAProxy

Group replication with HAProxy uses HAProxy only. It does not use MySQL Router or Orchestrator.

- **With PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.0.35-36.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" },
       "pmm": { "image": "percona/pmm-client:3.8.1" }
   }}'
```

- **Without PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.0.35-36.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

- **With Binlog Server**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "backup":{
          "image": "percona/percona-xtrabackup:8.0.35-36.1",
          "pitr": { "binlogServer": { "image": "perconalab/percona-binlog-
server:0.3.1" } }
       },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

After the upgrade, enable point-in-time binlog collection.

Group replication with MySQL Router

Group replication with MySQL Router uses MySQL Router only. It does not use HAProxy or Orchestrator.

- **With PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "router":{ "image": "percona/percona-mysql-router:8.0.46-37.1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.0.35-36.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" },
       "pmm": { "image": "percona/pmm-client:3.8.1" }
   }}'
```

- **Without PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "router":{ "image": "percona/percona-mysql-router:8.0.46-37.1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.0.35-36.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

- **With Binlog Server**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "router":{ "image": "percona/percona-mysql-router:8.0.46-37.1" }
       },
       "backup":{
          "image": "percona/percona-xtrabackup:8.0.35-36.1",
          "pitr": { "binlogServer": { "image": "perconalab/percona-binlog-
server:0.3.1" } }
       },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

After the upgrade, enable point-in-time binlog collection.

Upgrade the Operator and CRD via Helm

If you have installed the Operator using Helm, you can upgrade the Operator with the `helm upgrade` command.

The helm upgrade command updates only the Operator deployment. The update flow for the database management system is the same for all installation methods, whether it was installed via Helm or `kubectl`.

1. Update the Custom Resource Definition for the Operator, taking it from the official repository on GitHub, and do the same for the Role-based access control:

   ```
   kubectl apply --server-side -f
   https://raw.githubusercontent.com/percona/percona-server-mysql-
   operator/v1.2.0/deploy/crd.yaml
   kubectl apply --server-side -f
   https://raw.githubusercontent.com/percona/percona-server-mysql-
   operator/v1.2.0/deploy/rbac.yaml
   ```

2. Next, update the Operator deployment.

   #### With default parameters

   If you installed the Operator with default parameters, the upgrade can be done as follows:

   ```
   helm upgrade my-op percona/ps-operator --version 1.2.0
   ```

   #### With customized parameters

   If you installed the Operator with some customized parameters, you should list these options in the upgrade command.

   You can get the list of the used options in YAML format with the `helm get values my-op -a > my-values.yaml` command. Then pass this file directly to the upgrade command as follows:

   ```
   helm upgrade my-op percona/ps-operator --version 1.2.0 -f my-values.yaml
   ```

3. Update the Custom Resource, the database and components. This step ensures all new features and improvements of the latest release work well within your environment.

Update the Custom Resource, the database and components ↓

Update the Custom Resource, the database and components

Update the Custom Resource, the database, backup, proxy and PMM Client image names with a newer version tag. This step ensures all new features and improvements of the latest release work well within your environment.

Find the image names in the list of certified images.

We recommend to update the PMM Server **before** the upgrade of PMM Client. If you haven't updated your PMM Server yet, exclude PMM Client from the list of images to update.

Since this is a working cluster, the way to update the Custom Resource is to apply a patch with the `kubectl patch ps` command.

Select the command that matches your setup from the sections below.

MySQL 8.4

Asynchronous replication (tech preview)

Asynchronous replication uses HAProxy and Orchestrator. It does not use MySQL Router.

- **With PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "orchestrator":{ "image": "percona/percona-orchestrator:3.2.6-22" },
       "backup":{ "image": "percona/percona-xtrabackup:8.4.0-6.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" },
       "pmm": { "image": "percona/pmm-client:3.8.1" }
   }}'
```

- **Without PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "orchestrator":{ "image": "percona/percona-orchestrator:3.2.6-22" },
       "backup":{ "image": "percona/percona-xtrabackup:8.4.0-6.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

- **With Binlog Server**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "orchestrator":{ "image": "percona/percona-orchestrator:3.2.6-22" },
       "backup":{
          "image": "percona/percona-xtrabackup:8.4.0-6.1",
          "pitr": { "binlogServer": { "image": "perconalab/percona-binlog-
server:0.3.1" } }
       },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

After the upgrade, enable point-in-time binlog collection.

Group replication with HAProxy

Group replication with HAProxy uses HAProxy only. It does not use MySQL Router or Orchestrator.

- **With PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.4.0-6.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" },
       "pmm": { "image": "percona/pmm-client:3.8.1" }
   }}'
```

- **Without PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.4.0-6.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

- **With Binlog Server**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "backup":{
          "image": "percona/percona-xtrabackup:8.4.0-6.1",
          "pitr": { "binlogServer": { "image": "perconalab/percona-binlog-
server:0.3.1" } }
       },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

After the upgrade, enable point-in-time binlog collection.

Group replication with MySQL Router

Group replication with MySQL Router uses MySQL Router only. It does not use HAProxy or Orchestrator.

- **With PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "router":{ "image": "percona/percona-mysql-router:8.4.10-10.1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.4.0-6.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" },
       "pmm": { "image": "percona/pmm-client:3.8.1" }
   }}'
```

- **Without PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "router":{ "image": "percona/percona-mysql-router:8.4.10-10.1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.4.0-6.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

- **With Binlog Server**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "router":{ "image": "percona/percona-mysql-router:8.4.10-10.1" }
       },
       "backup":{
          "image": "percona/percona-xtrabackup:8.4.0-6.1",
          "pitr": { "binlogServer": { "image": "perconalab/percona-binlog-
server:0.3.1" } }
       },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

After the upgrade, enable point-in-time binlog collection.

MySQL 8.0

Asynchronous replication (tech preview)

Asynchronous replication uses HAProxy and Orchestrator. It does not use MySQL Router.

- **With PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "orchestrator":{ "image": "percona/percona-orchestrator:3.2.6-22" },
       "backup":{ "image": "percona/percona-xtrabackup:8.0.35-36.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" },
       "pmm": { "image": "percona/pmm-client:3.8.1" }
   }}'
```

- **Without PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "orchestrator":{ "image": "percona/percona-orchestrator:3.2.6-22" },
       "backup":{ "image": "percona/percona-xtrabackup:8.0.35-36.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

- **With Binlog Server**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "orchestrator":{ "image": "percona/percona-orchestrator:3.2.6-22" },
       "backup":{
          "image": "percona/percona-xtrabackup:8.0.35-36.1",
          "pitr": { "binlogServer": { "image": "perconalab/percona-binlog-
server:0.3.1" } }
       },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

After the upgrade, enable point-in-time binlog collection.

Group replication with HAProxy

Group replication with HAProxy uses HAProxy only. It does not use MySQL Router or Orchestrator.

- **With PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.0.35-36.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" },
       "pmm": { "image": "percona/pmm-client:3.8.1" }
   }}'
```

- **Without PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.0.35-36.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

- **With Binlog Server**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "backup":{
          "image": "percona/percona-xtrabackup:8.0.35-36.1",
          "pitr": { "binlogServer": { "image": "perconalab/percona-binlog-
server:0.3.1" } }
       },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

After the upgrade, enable point-in-time binlog collection.

Group replication with MySQL Router

Group replication with MySQL Router uses MySQL Router only. It does not use HAProxy or

Orchestrator.

- **With PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "router":{ "image": "percona/percona-mysql-router:8.0.46-37.1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.0.35-36.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" },
       "pmm": { "image": "percona/pmm-client:3.8.1" }
   }}'
```

- **Without PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "router":{ "image": "percona/percona-mysql-router:8.0.46-37.1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.0.35-36.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

- **With Binlog Server**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "router":{ "image": "percona/percona-mysql-router:8.0.46-37.1" }
       },
       "backup":{
          "image": "percona/percona-xtrabackup:8.0.35-36.1",
          "pitr": { "binlogServer": { "image": "perconalab/percona-binlog-
server:0.3.1" } }
       },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

After the upgrade, enable point-in-time binlog collection.

Upgrade the Operator and CRD via Operator Lifecycle Manager (OLM)

Considerations for using OpenShift 4.22

Starting with OpenShift 4.22, the way images with not fully qualified names are pulled has changed for repositories that share the same repository name on DockerHub and Red Hat Marketplace. By default the tags are pulled from Red Hat Marketplace. Specifying not fully qualified image names may result in the `ImagePullBackOff` error.

- **OLM installation:** Images are provided with the fully qualified names and are pulled from the Red Hat Marketplace/DockerHub registry.
- **Manual install/update with default manifests:** Images must use the `docker.io` registry prefix to guarantee successful download from the DockerHub `percona-server-mysql-operator` repository. See the Update via the command-line interface section for the exact steps.

The upgrade on OpenShift consists of two steps:

- Upgrade the Operator Deployment
- Upgrade the database cluster

Upgrade the Operator via Operator Lifecycle Manager (OLM)

You can upgrade the Operator Deployment for MySQL that was installed on the OpenShift platform using OLM directly through the Operator Lifecycle Manager.

If you know the OLM upgrade workflow, jump to the update Deployment steps.

Understand how OLM applies Operator upgrades

OLM manages the Operator using a resource called a `ClusterServiceVersion` (CSV). Each CSV represents a specific version of the Operator and contains:

- the Operator Deployment specification
- required RBAC permissions

- CRD definitions
- metadata and examples

When a new Operator version is available and the upgrade is approved, OLM installs the new CSV and reconciles the Operator Deployment to match it. The following items are replaced with the values defined in the new CSV:

- container image
- command and arguments
- labels and annotations
- probes
- most Deployment fields

If you previously customized the Operator Deployment manually, these changes are overwritten during the upgrade.

The CRD may be updated too, if the new Operator version introduces schema changes. However, OLM doesn't modify the `PerconaServerMySQL` Custom Resource. It remains unchanged and continues running with its current configuration. For how to update it, refer to Update Percona Server for MySQL.

Persisting custom Operator configuration

If you need to customize the Operator Deployment (for example, to adjust resource limits or set environment variables), you can do it through the Subscription.

A Subscription is the OLM resource that defines which operator you want to install and how you want it to be upgraded. A Subscription connects your cluster to an Operator package in a CatalogSource and ensures that OLM continuously manages that Operator according to your chosen update strategy.

Here's how you can customize the Operator Deployment. This example command sets an environment variable for the Operator:

```
kubectl patch subscription percona-server-mysql-operator -n <namespace> \
  --type merge \
  -p '{"spec":{"config":{"env":[{"name":"LOG_LEVEL","value":"DEBUG"}]}}}'
```

OLM supports overriding only the following fields through the Subscription:

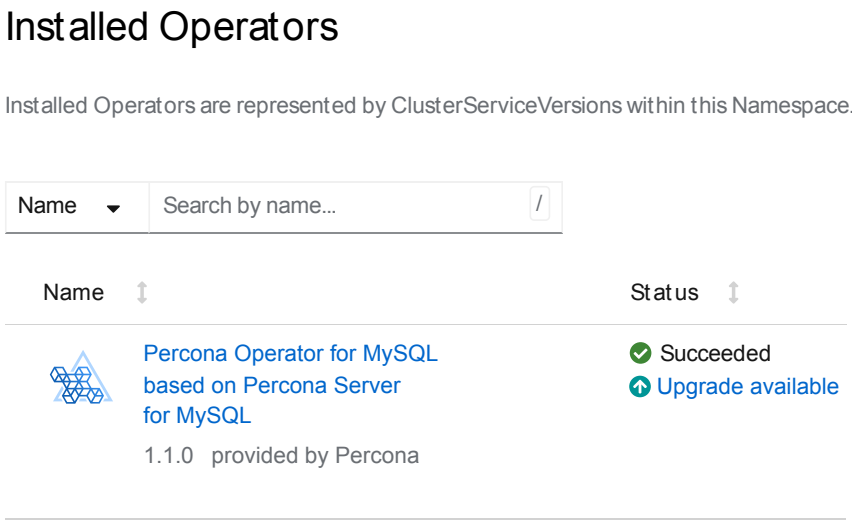
- env

- envFrom
- volumes
- volumeMounts
- resources
- nodeSelector
- tolerations
- affinity

These overrides are applied on top of the CSV and persist across upgrades. All other fields are overridden by the values from the new CSV during the Operator Deployment upgrade.

Upgrade the Operator

1. Log in to the OpenShift web console and check the list of installed Operators in your namespace to see if upgrades are available.



2. Click the “Upgrade available” link to review details, click “Preview InstallPlan,” and then click “Approve” to upgrade the Operator.

Update via the command-line interface

The following steps apply if you plan to use OpenShift 4.22. See the Considerations for using OpenShift 4.22.

1. Check all clusters managed by the Operator to see if `initContainer.image` is set.

- If defined: skip the next step.
- If undefined: proceed to step 2.

2. Apply a patch to the clusters with undefined `initContainer.image` to define this image with the `docker.io` registry in the image path:

   ```
   kubectl patch ps ps-cluster1 --type=merge --patch '{
     "spec": {
       "initcontainer": {
             "image": "docker.io/percona-server-mysql-operator:1.1.0"
       }
     }
   }'
   ```

   Important! This command triggers the restart of your clusters. Wait till they restart and report the `Ready` status.

3. Update the Operator deployment and specify the `docker.io` registry name in the image path:

   ```
   kubectl patch deployment percona-server-mysql-operator \
   -p'{"spec":{"template":{"spec":{"containers":[{"name":"percona-server-
   mysql-operator","image":"docker.io/percona/percona-server-mysql-
   operator:1.2.0"}]}}}}'
   ```

4. Update the Custom Resource version and the database cluster. Specify the `initContainer` image with the `docker.io` registry name in the path.

   The following example shows how to update Percona Server for MySQL cluster 8.4 with Group Replication and HAProxy:

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
  "spec": {
    "crVersion": "1.2.0",
    "initContainer": "docker.io/percona/percona-server-mysql-
operator:1.2.0",
    "mysql":{ "image": "docker.io/percona/percona/percona-server:8.4.10-
10.1" },
    "proxy":{
        "haproxy":{ "image": "docker.io/percona/haproxy:2.8.18-1" }
    },
    "backup":{ "image": "docker.io/percona/percona-xtrabackup:8.4.0-6.1"
},
    "toolkit":{ "image": "docker.io/percona/percona-toolkit:3.7.1-4" },
    "pmm":{ "image": "docker.io/percona/pmm-client:3.8.1" }
  }
}'
```

Upgrading Percona Server for MySQL

You can decide how to run the database upgrades:

- Automatically - the Operator periodically checks for new versions of the database images and for valid image paths and automatically updates your deployment with the latest, recommended or a specific version of the database and other components included. To do so, the Operator queries a special *Version Service* server at scheduled times. If the current version should be upgraded, the Operator updates the Custom Resource to reflect the new image paths and sequentially deletes Pods, allowing StatefulSet to redeploy the cluster Pods with the new image.
- Manually - you manually update the Custom Resource and specify the desired version of the database. Then, depending on the configured update strategy, either the Operator automatically updates the deployment to this version. Or you manually trigger the upgrade by deleting Pods.

The way to instruct the Operator how it should run the database upgrades is to set the `upgradeOptions.apply` Custom Resource option to one of the following:

- `Never` - the Operator never makes automatic upgrades. You must upgrade the Custom Resource and images manually.
- `Disabled` - the Operator doesn't not carry on upgrades automatically. You must upgrade the Custom Resource and images manually.
- `Recommended` - the Operator automatically updates the database and components to the version flagged as Recommended.
- `Latest` - the Operator automatically updates the database and components to the most recent available version
- `version` - specify the specific database version that you want to update to in the format `8.4.10-10.1`, `8.0.46-37.1`, etc. The Operator updates the database to it automatically. Find available versions in the list of certified images.

For previous versions, refer to the old releases documentation archive.

Minor upgrade to a specific version

Assumptions

For the procedures in this tutorial, we assume that you have set up the `Smart Update` strategy to update the objects in your database cluster.

Before you start

Before updating the database, complete the following preparation steps:

1. Check the version of the Operator you have in your Kubernetes environment. If you need to update it, refer to the Operator upgrade guide.
2. Check the Custom Resource manifest configuration to ensure the following:
   - `spec.updateStrategy` option is set to `SmartUpdate`
   - `spec.upgradeOptions.apply` option is set to `Never` or `Disabled` (this means that the Operator will not carry on upgrades automatically)

   ```
   ...
   spec:
     updateStrategy: SmartUpdate
     upgradeOptions:
       apply: Disabled
       ...
   ```
3. Before selecting the update command, identify your current setup:
   - **MySQL version**: Check your current `mysql.image` value in your Custom Resource
   - **Replication type**: Check the `mysql.clusterType` option in your Custom Resource (`async` or `group`)
   - **Proxy type**: For async replication, you use HAProxy. For group replication, check if `proxy.haproxy.enabled` or `proxy.router.enabled` is set to `true` in your Custom Resource
4. We recommend to update PMM Server before upgrading PMM Client.

If you use Binlog Server

The new Binlog Server image 0.3.1 is incompatible with the previous version. To upgrade the Binlog Server you must start a fresh binlog collection. Here's how to do it:

1. Stop point-in-time recovery binlog collection
2. Make a new full backup
3. Change the binlog storage settings in your cluster configuration to point to a new location. It can be a new bucket or a new prefix for the existing bucket

Update commands

Update the Custom Resource, the database, backup, proxy and PMM Client image names with a newer version tag. This step ensures all new features and improvements of the latest release work well within your environment.

Find the image names in the list of certified images.

We recommend to update the PMM Server **before** the upgrade of PMM Client. If you haven't updated your PMM Server yet, exclude PMM Client from the list of images to update.

Since this is a working cluster, the way to update the Custom Resource is to apply a patch with the `kubectl patch ps` command.

Select the command that matches your setup from the sections below.

MySQL 8.4

Asynchronous replication (tech preview)

Asynchronous replication uses HAProxy and Orchestrator. It does not use MySQL Router.

- **With PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "orchestrator":{ "image": "percona/percona-orchestrator:3.2.6-22" },
       "backup":{ "image": "percona/percona-xtrabackup:8.4.0-6.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" },
       "pmm": { "image": "percona/pmm-client:3.8.1" }
   }}'
```

- **Without PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "orchestrator":{ "image": "percona/percona-orchestrator:3.2.6-22" },
       "backup":{ "image": "percona/percona-xtrabackup:8.4.0-6.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

- **With Binlog Server**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "orchestrator":{ "image": "percona/percona-orchestrator:3.2.6-22" },
       "backup":{
          "image": "percona/percona-xtrabackup:8.4.0-6.1",
          "pitr": { "binlogServer": { "image": "perconalab/percona-binlog-
server:0.3.1" } }
       },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

After the upgrade, enable point-in-time binlog collection.

Group replication with HAProxy

Group replication with HAProxy uses HAProxy only. It does not use MySQL Router or Orchestrator.

- **With PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.4.0-6.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" },
       "pmm": { "image": "percona/pmm-client:3.8.1" }
   }}'
```

- **Without PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.4.0-6.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

- **With Binlog Server**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "backup":{
          "image": "percona/percona-xtrabackup:8.4.0-6.1",
          "pitr": { "binlogServer": { "image": "perconalab/percona-binlog-
server:0.3.1" } }
       },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

After the upgrade, enable point-in-time binlog collection.

Group replication with MySQL Router

Group replication with MySQL Router uses MySQL Router only. It does not use HAProxy or Orchestrator.

- **With PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "router":{ "image": "percona/percona-mysql-router:8.4.10-10.1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.4.0-6.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" },
       "pmm": { "image": "percona/pmm-client:3.8.1" }
   }}'
```

- **Without PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "router":{ "image": "percona/percona-mysql-router:8.4.10-10.1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.4.0-6.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

- **With Binlog Server**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.4.10-10.1" },
       "proxy":{
          "router":{ "image": "percona/percona-mysql-router:8.4.10-10.1" }
       },
       "backup":{
          "image": "percona/percona-xtrabackup:8.4.0-6.1",
          "pitr": { "binlogServer": { "image": "perconalab/percona-binlog-
server:0.3.1" } }
       },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
    }}'
```

Asynchronous replication (tech preview)

Asynchronous replication uses HAProxy and Orchestrator. It does not use MySQL Router.

- **With PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "orchestrator":{ "image": "percona/percona-orchestrator:3.2.6-22" },
       "backup":{ "image": "percona/percona-xtrabackup:8.0.35-36.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" },
       "pmm": { "image": "percona/pmm-client:3.8.1" }
    }}'
```

- **Without PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "orchestrator":{ "image": "percona/percona-orchestrator:3.2.6-22" },
       "backup":{ "image": "percona/percona-xtrabackup:8.0.35-36.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
    }}'
```

- **With Binlog Server**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "orchestrator":{ "image": "percona/percona-orchestrator:3.2.6-22" },
       "backup":{
          "image": "percona/percona-xtrabackup:8.0.35-36.1",
          "pitr": { "binlogServer": { "image": "perconalab/percona-binlog-
server:0.3.1" } }
       },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

Group replication with HAProxy

After the upgrade, enable point-in-time binlog collection.

Group replication with HAProxy uses HAProxy only. It does not use MySQL Router or Orchestrator.

- **With PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.0.35-36.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" },
       "pmm": { "image": "percona/pmm-client:3.8.1" }
   }}'
```

- **Without PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.0.35-36.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

- **With Binlog Server**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "haproxy":{ "image": "percona/haproxy:2.8.18-1" }
       },
       "backup":{
          "image": "percona/percona-xtrabackup:8.0.35-36.1",
          "pitr": { "binlogServer": { "image": "perconalab/percona-binlog-
server:0.3.1" } }
       },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

After the upgrade, enable point-in-time binlog collection.

Group replication with MySQL Router

Group replication with MySQL Router uses MySQL Router only. It does not use HAProxy or Orchestrator.

- **With PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "router":{ "image": "percona/percona-mysql-router:8.0.46-37.1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.0.35-36.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" },
       "pmm": { "image": "percona/pmm-client:3.8.1" }
   }}'
```

- **Without PMM Client**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "router":{ "image": "percona/percona-mysql-router:8.0.46-37.1" }
       },
       "backup":{ "image": "percona/percona-xtrabackup:8.0.35-36.1" },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

- **With Binlog Server**

```
kubectl patch ps ps-cluster1 --type=merge --patch '{
   "spec": {
       "crVersion":"1.2.0",
       "mysql":{ "image": "percona/percona-server:8.0.46-37.1" },
       "proxy":{
          "router":{ "image": "percona/percona-mysql-router:8.0.46-37.1" }
       },
       "backup":{
          "image": "percona/percona-xtrabackup:8.0.35-36.1",
          "pitr": { "binlogServer": { "image": "perconalab/percona-binlog-
server:0.3.1" } }
       },
       "toolkit":{ "image": "percona/percona-toolkit:3.7.1-4" }
   }}'
```

After the upgrade, enable point-in-time binlog collection.

Track the upgrade progress

After applying the patch, the deployment rollout will be automatically triggered. You can track the rollout process in real time with the `kubectl rollout status` command with the name of your cluster:

```
kubectl rollout status sts ps-cluster1-ps
```

Automated upgrade

Assumptions

For the procedures in this tutorial, we assume that you have set up the `Smart Update` strategy to update the objects in your database cluster.

Before you start

We recommend to update PMM Server before upgrading PMM Client.

If you use Binlog Server for point-in-time recovery, complete the pre-upgrade steps.

Procedure

1. Check the version of the Operator you have in your Kubernetes environment. If you need to update it, refer to the Operator upgrade guide
2. Change `spec.crVersion` option to match the version of the Custom Resource Definition upgrade while upgrading the Operator:

   ```
   ...
   spec:
     crVersion: 1.2.0
     ...
   ```

   Note

   If you don't update `crVersion`, minor version upgrade is the only one to occur. For example, the image `percona-server:8.0.30-22` can be upgraded to `percona-server:8.0.32-24`.

3. Make sure that `spec.updateStrategy` option is set to `SmartUpdate`.
4. Change the `upgradeOptions.apply` option from `Disabled` to one of the following values:

- `Recommended` - the Operator will choose the most recent version of software flagged as “Recommended”
- `Latest` - automatic upgrades will choose the most recent version of the software available,
- *version number* - specify the desired version explicitly (version numbers are specified as `8.0.46-37.1`, etc.). Actual versions can be found in the list of certified images (for older releases, please refer to the old releases documentation archive).

5. Make sure to set the valid Version Server URL for the `upgradeOptions.versionServiceEndpoint` key. The Operator checks the new software versions in the Version Server. If the Operator can’t reach the Version Server, the upgrades won’t happen.

   #### Percona’s Version Service (default)

   You can use the URL of the official Percona’s Version Service (default). Set `upgradeOptions.versionServiceEndpoint` to `https://check.percona.com`.

   #### Version Service inside your cluster

   Alternatively, you can run Version Service inside your cluster. This can be done with the `kubectl` command as follows:

   ```
   kubectl run version-service --image=perconalab/version-service --
   env="SERVE_HTTP=true" --port 11000 --expose
   ```

6. Specify the schedule to check for the new versions in in CRON format for the `upgradeOptions.schedule` option.

   The following example sets the midnight update checks with the official Percona’s Version Service:

   ```
   spec:
     updateStrategy: SmartUpdate
     upgradeOptions:
       apply: Recommended
       versionServiceEndpoint: https://check.percona.com
       schedule: "0 0 * * *"
   ...
   ```

> **Note**
>
> You can force an immediate upgrade by changing the schedule to `* * * * *` (continuously check and upgrade) and changing it back to another more conservative schedule when the upgrade is complete.

7. Apply your changes to the Custom Resource in the usual way:

```
kubectl apply -f deploy/cr.yaml
```

Pause/resume the cluster

There may be external situations when it is needed to shutdown the Percona Server for MySQL cluster for a while and then start it back up (some works related to the maintenance of the enterprise infrastructure, etc.).

The `deploy/cr.yaml` file contains a special `spec.pause` key for this. Setting it to `true` gracefully stops the cluster:

```
spec:
  .......
  pause: true
```

Pausing the cluster may take some time, and when the process is over, you will see only the Operator Pod running:

```
kubectl get pods
NAME                                               READY   STATUS    RESTARTS
AGE
percona-server-mysql-operator-7ff9cff46f-6dtgs   1/1     Running   0
9m19s
```

To start the cluster after it was shut down just revert the `spec.pause` key to `false`.

Starting the cluster will take time. The process is over when all Pods have reached their Running status:

```
NAME                                               READY   STATUS    RESTARTS
AGE
ps-cluster1-mysql-0                                   2/2     Running   0
3m43s
ps-cluster1-mysql-1                                   2/2     Running   0
3m3s
ps-cluster1-mysql-2                                   2/2     Running   0
2m27s
ps-cluster1-router-c89b8487-bbtqm                     1/1     Running   0
89s
ps-cluster1-router-c89b8487-cll6l                     1/1     Running   0
89s
ps-cluster1-router-c89b8487-q6mnl                     1/1     Running   0
89s
percona-server-mysql-operator-7ff9cff46f-6dtgs   1/1     Running   0
13m
```

Note

Clusters with Group Replication undergo crash recovery on each unpause. This is caused by the specifics of Group Replication, which supposes continuous availability of the database service by design. See upstream documentation for more details.

Cleanup

Delete Percona Operator for MySQL based on Percona Server for MySQL

You may have different reasons to clean up your Kubernetes environment: moving from trial deployment to a production one, testing experimental configurations and the like. In either case, you need to remove some (or all) of these objects:

- Percona Server for MySQL managed by the Operator
- Percona Operator for MySQL itself
- Custom Resource Definition deployed with the Operator
- Resources like PVCs and Secrets

Delete the database cluster

To delete the database cluster means to delete the Custom Resource associated with it.

Note

There are 2 finalizers defined in the Custom Resource, which define whether to delete or preserve TLS-related objects and data volumes when the cluster is deleted.

- `finalizers.percona.com/delete-mysql-pvc`: if present, Persistent Volume Claims for the database cluster Pods and all user Secrets are deleted along with the cluster deletion.
- `finalizers.percona.com/delete-ssl`: if present, objects, created for SSL (Secret, certificate, and issuer) are deleted along with the cluster deletion.

These finalizers are off by default in the `deploy/cr.yaml` configuration file, and it allows you to recreate the cluster without losing data, credentials for the system users, etc. You can always delete TLS-related objects and PVCs manually, if needed.

The steps are the following:

1. List the Custom Resources. Replace the `<namespace>` placeholder with your value

```
kubectl get ps -n <namespace>
```

2. Delete the Custom Resource with the name of your cluster

```
kubectl delete ps <cluster_name> -n <namespace>
```

Sample output

```
perconaservermysql.ps.percona.com "ps-cluster1" deleted
```

It may take a while to stop and delete the cluster.

3. Check that the cluster is deleted by listing the Custom Resources again:

```
kubectl get ps -n <namespace>
```

Sample output

```
No resources found in <namespace> namespace.
```

Delete the Operator

Choose the instructions relevant to the way you installed the Operator.

kubectl

To uninstall the Operator, delete the Deployments related to it.

1. List the deployments. Replace the `<namespace>` placeholder with your namespace.

```
kubectl get deploy -n <namespace>
```

Sample output

```
NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
percona-server-mysql-operator   1/1     1            1           42m
```

2. Delete the `percona-*` deployment

```
kubectl delete deploy percona-server-mysql-operator -n <namespace>
```

Sample output

```
deployment.apps "percona-server-mysql-operator" deleted
```

3. Check that the Operator is deleted by listing the Pods. As a result you should have no Pods related to it.

```
kubectl get pods -n <namespace>
```

Sample output

```
No resources found in <namespace> namespace.
```

4. If you are not just deleting the Operator and Percona Server for MySQL from a specific namespace, but want to clean up your entire Kubernetes environment, you can also delete the CustomResourceDefinitions (CRDs).

Warning: CRDs in Kubernetes are non-namespaced but are available to the whole environment. This means that you shouldn't delete CRDs if you still have the Operator and database cluster in some namespace.

Get the list of CRDs.

```
kubectl get crd
```

Sample output

```
NAME                                        CREATED AT
perconaservermysqlbackups.ps.percona.com    2025-02-07T20:10:42Z
perconaservermysqlrestores.ps.percona.com   2025-02-07T20:10:42Z
perconaservermysqls.ps.percona.com          2025-02-07T20:10:42Z
```

5. Delete the `percona*.ps.percona.com` CRDs

```
kubectl delete crd perconaservermysqlbackups.ps.percona.com
perconaservermysqlrestores.ps.percona.com
perconaservermysqls.ps.percona.com
```

Sample output

```
customresourcedefinition.apiextensions.k8s.io
"perconaservermysqlbackups.ps.percona.com" deleted
customresourcedefinition.apiextensions.k8s.io
"perconaservermysqlrestores.ps.percona.com" deleted
customresourcedefinition.apiextensions.k8s.io "perconaservermysqls.ps.percona.com"
deleted
```

Helm

To delete the Operator, do the following:

1. List the Helm charts:

```
helm list -n <namespace>
```

Sample output

```
ps-cluster1    <namespace>         1           2023-10-31 10:18:10.763049 +0100 CET
deployed    ps-db-1.2.0        1.2.0
my-op       <namespace>         1           2023-10-31 10:15:18.41444 +0100 CET
deployed    ps-operator-1.2.0   1.2.0
```

2. Delete the release object for Percona XtraDB Cluster

```
helm uninstall ps-cluster1 --namespace <namespace>
```

3. Delete the release object for the Operator

```
helm uninstall my-op --namespace <namespace>
```

Clean up resources

By default, TLS-related objects, user Secrets and data volumes remain in Kubernetes environment after you delete the cluster to allow you to recreate it without losing the data.

You can automate resource cleanup by turning on `percona.com/delete-mysql-pvc` and/or `percona.com/delete-ssl` finalizers). You can also delete TLS-related objects and PVCs manually.

To manually clean up resources, do the following:

1. Delete Persistent Volume Claims.

    1. List PVCs. Replace the `<namespace>` placeholder with your namespace:

        ```
        kubectl get pvc -n <namespace>
        ```

        Sample output

        ```
        NAME                       STATUS   VOLUME
        CAPACITY   ACCESS MODES   STORAGECLASS   AGE
        datadir-ps-cluster1-mysql-0   Bound    pvc-8683d0ab-7ed4-48cb-93a9-
        bc6ceb6ec285   2G         RWO            standard       <unset>
        47m
        datadir-ps-cluster1-mysql-1   Bound    pvc-fbc5a8a4-94ff-4259-9d15-
        f798c97e0788   2G         RWO            standard       <unset>
        45m
        datadir-ps-cluster1-mysql-2   Bound    pvc-4c164ff3-a4f5-431c-9aa8-
        e5c7eb71a31b   2G         RWO            standard       <unset>
        44m
        ```

    2. Delete PVCs related to your cluster. The following command deletes PVCs for the `ps-cluster1` cluster:

        ```
        kubectl delete pvc datadir-ps-cluster1-mysql-0 datadir-ps-cluster1-
        mysql-1 datadir-ps-cluster1-mysql-2 -n <namespace>
        ```

        Sample output

        ```
        persistentvolumeclaim "datadir-ps-cluster1-mysql-0" deleted
        persistentvolumeclaim "datadir-ps-cluster1-mysql-1" deleted
        persistentvolumeclaim "datadir-ps-cluster1-mysql-2" deleted
        ```

2. Delete the Secrets

    1. List Secrets:

    ```
    kubectl get secrets -n <namespace>
    ```

    2. Delete the Secret:

    ```
    kubectl delete secret <secret_name> -n <namespace>
    ```

Advanced operations

Change replication type

By default, Percona Operator for MySQL is deployed with group-replication replication type and HAProxy enabled.

You can change the proxy from HAProxy to MySQL Router or vice versa. Note that you can use MySQL router only with the group-replication replication type.

To change a proxy, edit the `deploy/cr.yaml` file and set the `proxy.haproxy.enabled` option to `false` and the `proxy.router.enabled` option to `true`. Then apply the new configuration with the `kubectl apply -f deploy/cr.yaml` command.

You can also change the replication type for your cluster from group replication to asynchronous replication. To do it, do the following:

1. Pause the cluster. Since the cluster is running, run the `kubectl patch` command to update the cluster configuration. Replace the `<namespace>` placeholder with your namespace. For example, for the cluster with the name `ps-cluster1`, the command is:

   ```
   kubectl patch ps ps-cluster1 -n <namespace> --type json -
   p='[{"op":"add","path":"/spec/pause","value":true}]'
   ```

2. Edit the `deploy/cr.yaml` file and set the `mysql.clusterType` option to `async`. Make sure you have HAProxy enabled as a proxy in your configuration.

   ```
   mysql:
     clusterType: async
     ...
   proxy:
     haproxy:
       enabled: true
       ...
   ```

3. Apply the new configuration:

   ```
   kubectl apply -f deploy/cr.yaml -n <namespace>
   ```

4. Unpause the cluster.

   ```
   kubectl patch ps ps-cluster1 -n <namespace> --type json -
   p='[{"op":"add","path":"/spec/pause","value":false}]'
   ```

5. Wait for the cluster to be resumed. Check the status with the `kubectl get ps` command.

Changing replication type on a running cluster is not supported.

Using sidecar containers

The Operator allows you to deploy additional (so-called *sidecar*) containers to the Pod. You can use this feature to run debugging tools, some specific monitoring solutions, etc.

> **Note**
>
> Custom sidecar containers can easily access other components of your cluster. Therefore they should be used carefully and by experienced users only.

Adding a sidecar container

You can add sidecar containers to Percona Server for MySQL Pods. Just use `sidecars` subsection in the `mysql` section of the `deploy/cr.yaml` configuration file. In this subsection, you should specify the name and image of your container and possibly a command to run:

```
spec:
  mysql:
    ....
    sidecars:
    - image: busybox
      command: ["sleep", "30d"]
      name: my-sidecar-1
    ....
```

Apply your modifications as usual:

```
kubectl apply -f deploy/cr.yaml
```

Running `kubectl describe` command for the appropriate Pod can bring you the information about the newly created container:

```
kubectl describe pod ps-cluster1-mysql-0
....
Containers:
....
my-sidecar-1:
  Container ID:
docker://e8fbaae09c3b20c49da259c490d65cd68182b227c33e0fec560271a569b01394
  Image:         busybox
  Image ID:      docker-
pullable://busybox@sha256:5acba83a746c7608ed544dc1533b87c737a0b0fb730301639a0
179f9344b1678
  Port:          <none>
  Host Port:     <none>
  Command:
    sleep
    30d
  State:          Running
    Started:      Thu, 06 Jan 2022 10:38:15 +0300
  Ready:          True
  Restart Count:  0
  Environment:    <none>
  Mounts:
    /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-lkk2n
(ro)
....
```

Getting shell access to a sidecar container

You can login to your sidecar container as follows:

```
kubectl exec -it ps-cluster1-mysql-0 -c my-sidecar-1 -- sh
/ #
```

Mount volumes into sidecar containers

It is possible to mount volumes into sidecar containers.

Following subsections describe different volume types, which were tested with sidecar containers and are known to work.

Persistent Volume

You can use Persistent volumes when you need dynamically provisioned storage which doesn't depend on the Pod lifecycle.

To use such volume, you should *claim* durable storage with persistentVolumeClaim without specifying any non-important details.

> **Important**
>
> You can use PVCs with sidecar containers only when you deploy a new cluster. Updates to running cluster are not supported.

The following example requests 1G storage with `sidecar-volume-claim` PersistentVolumeClaim, and mounts the correspondent Persistent Volume to the `my-sidecar-1` container's filesystem under the `/volume1` directory:

```
...
  sidecars:
  - image: busybox
    command: ["sleep", "30d"]
    name: my-sidecar-1
    volumeMounts:
    - mountPath: /volume1
      name: sidecar-volume-claim
  sidecarPVCs:
  - name: sidecar-volume-claim
    spec:
      resources:
        requests:
          storage: 1Gi
      volumeMode: Filesystem
      accessModes:
        - ReadWriteOnce
```

Secret

You can use a secret volume to pass the information which needs additional protection (e.g. passwords), to the container. Secrets are stored with the Kubernetes API and mounted to the container as RAM-stored files.

You can mount a secret volume as follows:

```
...
  sidecars:
  - image: busybox
    command: ["sleep", "30d"]
    name: my-sidecar-1
    volumeMounts:
    - mountPath: /secret
      name: sidecar-secret
  sidecarVolumes:
  - name: sidecar-secret
    secret:
      secretName: mysecret
```

The above example creates a `sidecar-secret` volume (based on already existing `mysecret` Secret object) and mounts it to the `my-sidecar-1` container's filesystem under the `/secret` directory.

> **Note**
>
> Don't forget you need to create a Secret Object before you can use it.

configMap

You can use a configMap volume to pass some configuration data to the container.

You can mount a configMap volume as follows:

```
...
  sidecars:
  - image: busybox
    command: ["sleep", "30d"]
    name: my-sidecar-1
    volumeMounts:
    - mountPath: /config
      name: sidecar-config
  sidecarVolumes:
  - name: sidecar-config
    configMap:
      name: myconfigmap
```

The above example creates a `sidecar-config` volume (based on already existing `myconfigmap` configMap object) and mounts it to the `my-sidecar-1` container's filesystem under the `/config` directory.

Note

Don’t forget you need to create a configMap Object before you can use it.

Labels and annotations

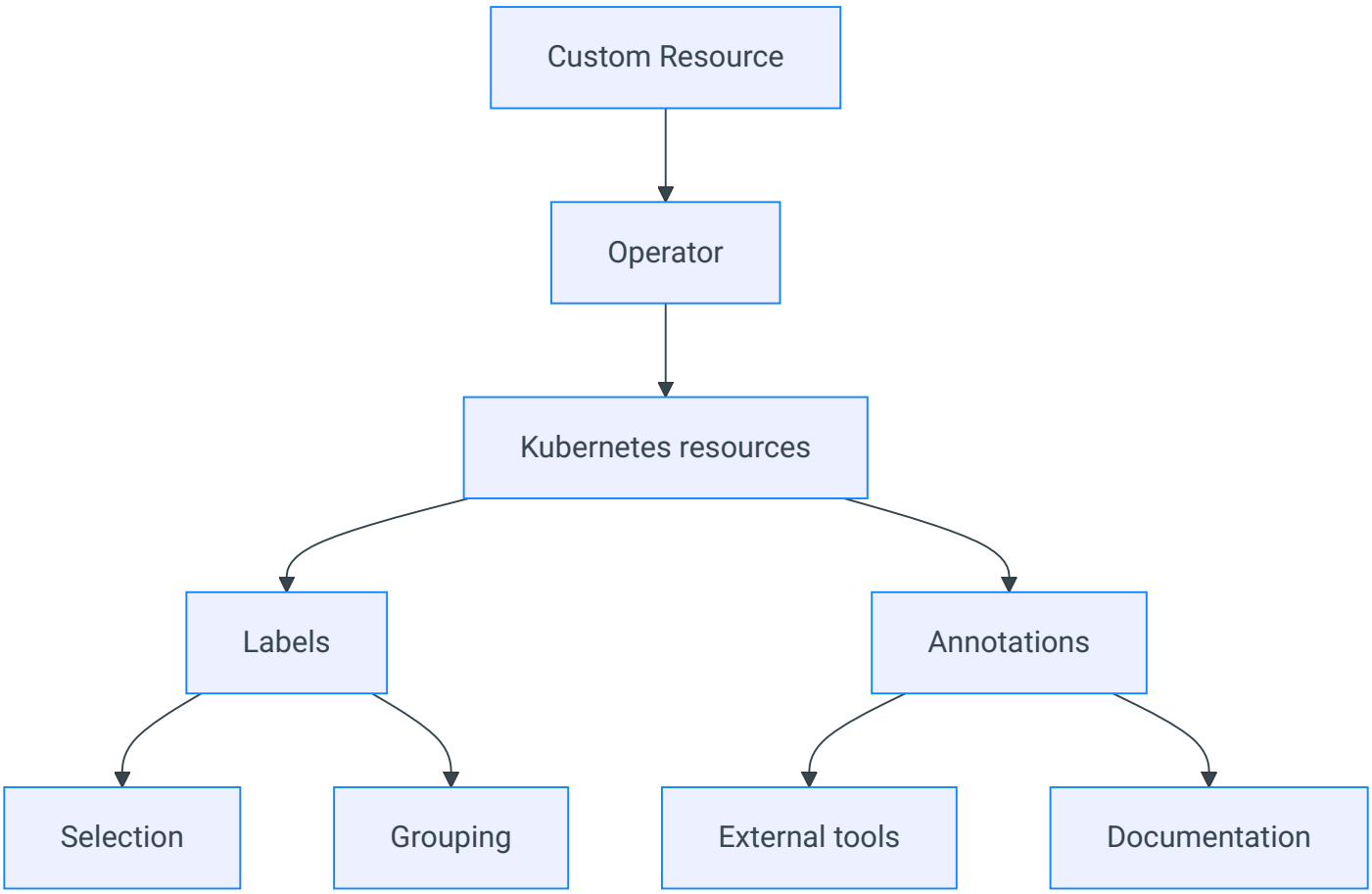
Labels and annotations are used to attach additional metadata information to Kubernetes resources.

Labels and annotations are rather similar but differ in purpose.

Labels are used by Kubernetes to identify and select objects. They enable filtering and grouping, allowing users to apply selectors for operations like deployments or scaling.

Annotations are assigning additional *non-identifying* information that doesn't affect how Kubernetes processes resources. They store descriptive information like deployment history, monitoring configurations or external integrations.

The following diagram illustrates this difference:



Both Labels and Annotations are assigned to the following objects managed by Percona Operator for MySQL:

- Custom Resource Definitions
- Custom Resources

- Deployments
- Services
- StatefulSets
- PVCs
- Pods
- ConfigMaps and Secrets

When to use labels and annotations

Use **Labels** when:

- The information is used for object selection
- The data is used for grouping or filtering
- The information is used by Kubernetes controllers
- The data is used for operational purposes

Use **Annotations** when:

- The information is for external tools
- The information is used for debugging
- The data is used for monitoring configuration

Labels and annotations used by Percona Operator for MySQL

Labels

Name	Objects	Description	Example values
`app.kubernetes.io/name`	Services, StatefulSets, Deployments, etc.	Specifies the name of the application	percona-server
`app.kubernetes.io/instance`	Services, StatefulSets, Deployments	Identifies a specific instance of the application	ps-cluster1

`app.kubernetes.io/managed-by`	Services, StatefulSets	Indicates the controller managing the object	percona-server-mysql-operator
`app.kubernetes.io/component`	Services, StatefulSets	Specifies the component within the application	mysql, haproxy, router
`app.kubernetes.io/part-of`	Services, StatefulSets	Indicates the higher-level application the object belongs to	percona-server
`app.kubernetes.io/version`	CustomResourceDefinition	Specifies the version of the Percona MySQL Operator.	1.2.0
`percona.com/exposed`	Services	Indicates if the service is exposed externally	true, false
`percona.com/cluster`	Custom Resource	Identifies the MySQL cluster instance	ps-cluster1
`percona.com/backup-type`	Custom Resource	Specifies the type of backup being performed (e.g. cron for scheduled backups)	cron, manual
`percona.com/backup-ancestor`	Custom Resource	Specifies the name of the backup that was used as a base for the current backup	ps-cluster1-backup-2026-05-23
`mysql.percona.com/primary`	Pods	Marks the primary node in the MySQL cluster	true

Annotations

Name	Objects	Description	Example Values
`service.beta.kubernetes.io/aws-load-balancer-backend-protocol`	Services	Specifies the protocol for AWS load balancers	http, http-test

`service.beta.kubernetes.io/aws-load-balancer-backend`	Services	Specifies the backend type for AWS load balancers	test-type
`controller-gen.kubebuilder.io/version`	CustomResourceDefinition	Indicates the version of the Kubebuilder controller-gen tool used.	
`percona.com/last-applied-tls`	Services	Stores the hash of the last applied TLS configuration for the service	
`percona.com/last-applied-secret`	Secrets	Stores the hash of the last applied user Secret configuration	
`percona.com/configuration-hash`	Services	Used to track and validate configuration changes in the MySQL cluster components	
`percona.com/last-config-hash`	Services	Stores the hash of the most recent configuration	
`percona.com/passwords-updated`	Secrets	Indicates when passwords were last updated in the Secret	

Setting labels and annotations in the Custom Resource

You can define both Labels and Annotations as `key-value` pairs in the metadata section of a YAML manifest for a specific resource.

Set labels and annotations for Pods

You can set labels and annotations for Percona XtraBackup Pods, specific to the backup storage you use. To do this, use the `.spec.backup.storages.<STORAGE_NAME>.annotations` / `.spec.backup.storages.<STORAGE_NAME>.labels` keys in the Custom Resource manifest.

```
spec:
  backup:
    storages:
      s3-us-west:
        annotations:
          testName: scheduled-backup
        labels:
          backupWorker: 'True'
```

Set labels and annotations for Services

You can set labels and annotations for Services. For example, to control placement based on physical infrastructure or to improve your CI automation.

Use the following options in the Custom Resource manifest:

- `.spec.mysql.exposePrimary.annotations` / `.spec.mysql.exposePrimary.labels` - for MySQL primary service
- `.spec.mysql.expose.annotations` / `.spec.mysql.expose.labels` - for MySQL service for every Pod
- `.spec.proxy.haproxy.expose.annotations` / `.spec.proxy.haproxy.expose.labels` - for HAProxy Service,
- `.spec.proxy.router.expose.annotations` / `.spec.proxy.router.expose.labels` - for MySQL Router Service
- `.spec.orchestrator.expose.annotations` / `.spec.orchestrator.expose.labels` - for the Orchestrator Service

The following example shows how to set labels and annotations for a `<CLUSTER-NAME>-mysql-primary` service:

```
spec:
  mysql:
    exposePrimary:
      enabled: true
      type: ClusterIP
      annotations:
        my-annotation: annotation-value
     ...
      labels:
        my-label: label-value
     ...
```

Set global labels and annotations

You can also use the top-level spec `metadata.annotations` and `metadata.labels` options to set annotations and labels at a global level, for all resources created by the Operator:

```
apiVersion: ps.percona.com/v1alpha1
kind: PerconaServerMySQL
metadata:
  name: ps-cluster1
  annotations:
    percona.com/issue-vault-token: "true"
  labels:
    ...
```

Querying labels and annotations

To check which **labels** are attached to a specific object, use the additional `--show-labels` option of the `kubectl get` command.

For example, to see the Operator version associated with a Custom Resource Definition, use the following command:

```
kubectl get crd perconaservermysqls.ps.percona.com --show-labels
```

Sample output

```
NAME                                  CREATED AT            LABELS
perconaservermysqls.ps.percona.com    2025-05-23T10:40:54Z
mysql.percona.com/version=v1.2.0
```

To check **annotations** associated with an object, use the following command:

```
kubectl get <resource> <resource-name> -o jsonpath='{.metadata.annotations}'
```

For example, this command lists annotations assigned to a `ps-cluster1-mysql-0` Pod:

```
kubectl get pod ps-cluster1-mysql-0 -o jsonpath='{.metadata.annotations}'
```

Sample output

```
{"percona.com/last-applied-tls":"c8dfc846cb62b75ba8eab61b7e86a46c"}
```

Specifying labels and annotations ignored by the Operator

Sometimes various Kubernetes flavors can add their own annotations to the objects managed by the Operator.

The Operator keeps track of all changes to its objects and can remove annotations that it didn't create.

Here's how the Operator manages labels and annotations:

- If there are no annotations or labels in the `expose*.*` subsections of the Custom Resource, the Operator does nothing if a new label or an annotation is added to the Service object.
- The Operator doesn't remove any global labels or annotations that you defined in the `spec.metadata` section of the Custom Resource.
- The Operator keeps custom annotations and labels a Service if the `expose.labels` and `expose.annotations` fields in the Custom Resource are empty for this Service. If they are not empty, the Operator overrides custom labels and annotations with the `expose.annotations` and `expose.labels` values.
- If you exposed individual Pods, the Operator removes unknown annotations and labels from Services that the Operator created for Pods.

You can still specify which annotations and labels the Operator should keep. It is useful if a cloud provider adds own labels and annotations to Services. Or you may have custom automation tools that add own labels or annotations and you need to keep them.

List these labels and annotations in the `spec.ignoreAnnotations` or `spec.ignoreLabels` fields of the `deploy/cr.yaml`, as follows:

```
spec:
  ignoreAnnotations:
    - some.custom.cloud.annotation/smth
  ignoreLabels:
    - some.custom.cloud.label/smth
...
```

The label and annotation values must exactly match the ones defined for the Service to be kept.

Delete labels and annotations

The Operator can only add custom labels and annotations to objects and it cannot delete them. This means you mst manually delete custom annotations and labels when they are no longer needed.

To delete a label or an annotation, run the following commands:

- For labels:

```
kubectl label <resource> <name> <label-key>-
```

where `<label-key>` is the label you want to delete.

- For annotations:

```
kubectl annotate <resource> <name> <annotation-key>-
```

where `<annotation-key>` is the annotation you want to delete.

Telemetry

The Telemetry function enables the Operator gathering and sending basic anonymous data to Percona, which helps us to determine where to focus the development and what is the uptake for each release of Operator.

The following information is gathered:

- ID of the Custom Resource (the `metadata.uid` field)
- Kubernetes version
- Platform (is it Kubernetes or Openshift)
- PMM Version
- Operator version
- Percona Server for MySQL version
- HAProxy version
- Percona XtraBackup version

We do not gather anything that identify a system, but the following thing should be mentioned: Custom Resource ID is a unique ID generated by Kubernetes for each Custom Resource.

Telemetry is enabled by default and is sent to the Version Service server when the Operator connects to it at scheduled times to obtain fresh information about version numbers and valid image paths needed for the upgrade.

The landing page for this service, check.percona.com, explains what this service is.

You can disable telemetry with a special option when installing the Operator: edit the `operator.yaml` before applying it with the `kubectl apply -f deploy/operator.yaml` command. Open the `operator.yaml` file with your text editor, find the value of the `DISABLE_TELEMETRY` environment variable and set it to `true`:

```
env:
  ...
  - name: DISABLE_TELEMETRY
    value: "true"
  ...
```

For all supported Operator environment variables and how to change them after installation, see Configure Operator environment variables.

Configure DNS suffix for service discovery

In Kubernetes, services are assigned a DNS name to be accessible within the cluster. The domain name follows the pattern `<service-name>.<namespace>.svc.<cluster-domain>`. The default cluster domain is `cluster.local`, so a typical FQDN looks like `<service-name>.<namespace>.svc.cluster.local`.

When you refer to a service using only its short name, Kubernetes automatically expands it with this domain so the name resolves inside the cluster. This enables workloads to communicate without the need to specify fully-qualified domain names.

Clusters with custom DNS configurations may use a domain suffix different from the default `cluster.local`. This includes scenarios such as running inside a vcluster, which may respond to DNS queries with records that do not match the actual cluster domain.

The Operator tries to auto-detect the cluster's domain suffix by performing a CNAME lookup on the `kubernetes` Service in the `default` namespace (`kubernetes.default.svc`). If the lookup returns a value, the Operator uses that as the domain suffix. If auto-detection fails, the Operator falls back to the default `cluster.local` domain. This may lead to services not resolving properly within the cluster.

To address this, you can set your custom DNS suffix for the Operator to use when it constructs service names. As a result, the Operator produces hostnames that match your cluster's DNS configuration, ensuring correct service resolution and discovery.

How to configure

Add `clusterServiceDNSSuffix` under `spec` in your `PerconaServerMySQL` Custom Resource. Set it to your cluster's DNS suffix. For example, `cluster.local` for a standard cluster, or the host cluster's suffix when the Operator runs in a vcluster:

```
spec:
  ...
  clusterServiceDNSSuffix: cluster.local
  # ... rest of your spec
```

Apply the change:

```
kubectl apply -f deploy/cr.yaml -n <namespace>
```

Configure concurrency for a cluster reconciliation

Reconciliation is the process by which the Operator continuously compares the desired state with the actual state of the cluster. The desired state is defined in a Kubernetes custom resource, like PerconaServerMySQL.

If the actual state does not match the desired state, the Operator takes actions to bring the system into alignment. This means creating, updating, or deleting Kubernetes resources (Pods, Services, ConfigMaps, etc.) or performing database-specific operations like scaling, backups, or failover.

Reconciliation is triggered by a variety of events, including:

- Changes to the cluster configuration
- Changes to the cluster state
- Changes to the cluster resources

By default, the Operator has one reconciliation worker. This means that if you deploy or update 2 clusters at the same time, the Operator will reconcile them sequentially.

The `MAX_CONCURRENT_RECONCILES` environment variable in the `percona-server-mysql-operator` deployment controls the number of concurrent workers that can reconcile resources in Percona Server for MySQL clusters in parallel.

Thus, to extend the previous example, if you set the number of reconciliation workers to `2`, the Operator will reconcile both clusters in parallel. This also helps you with benchmarking the Operator performance.

The general recommendation is to set the number of concurrent workers equal to the number of Percona Server for MySQL clusters. When the number of workers is greater, the excessive workers will remain idle.

Set the number of reconciliation workers

1. Check the current value of the `MAX_CONCURRENT_RECONCILES` environment variable:

```
kubectl get deployment percona-server-mysql-operator -o
jsonpath='{.spec.template.spec.containers[0].env[?
(@.name=="MAX_CONCURRENT_RECONCILES")].value}'
```

Sample output

```
1
```

2. Set a new value with `kubectl set env`. The following command sets the number of workers to `2`:

   ```
   kubectl set env deployment/percona-server-mysql-operator
   MAX_CONCURRENT_RECONCILES=2
   ```

   You can set the value to any positive integer.

3. Verify the change by re-running the command from step 1:

   ```
   kubectl get deployment percona-server-mysql-operator -o
   jsonpath='{.spec.template.spec.containers[0].env[?
   (@.name=="MAX_CONCURRENT_RECONCILES")].value}'
   ```

   Sample output

   ```
   2
   ```

After the environment variable is updated, Kubernetes rolls out a new Operator Pod with the new setting.

Troubleshooting

Initial troubleshooting

Percona Operator for MySQL uses Custom Resources to manage options for the various components of the cluster.

- `PerconaServerMySQL` Custom Resource with Percona Server for MySQL cluster options (it has handy `ps` shortname also),
- `PerconaServerMySQLBackup` and `PerconaServerMySQLRestore` Custom Resources contain options for Percona XtraBackup used to backup Percona Server for MySQL and to restore it from backups (`ps-backup` and `ps-restore` shortnames are available for them).

The first thing you can check for the Custom Resource is to query it with `kubectl get` command:

```
kubectl get ps
```

Expected output

```
NAME       REPLICATION         ENDPOINT                    STATE   MYSQL   ORCHESTRATOR
HAPROXY   ROUTER   AGE
ps-cluster1   group-replication   ps-cluster1-haproxy.default   ready   3
3        20m
```

The Custom Resource should have `ready` state.

Note

You can check which Percona's Custom Resources are present and get some information about them as follows:

```
kubectl api-resources | grep -i percona
```

Expected output

```
perconaservermysqlbackups         ps-backup,ps-backups   ps.percona.com/v1alpha1
true         PerconaServerMySQLBackup
perconaservermysqlrestores        ps-restore             ps.percona.com/v1alpha1
true         PerconaServerMySQLRestore
perconaservermysqls               ps                     ps.percona.com/v1alpha1
true         PerconaServerMySQL
```

Check the Pods

If Custom Resource is not getting `ready` state, it makes sense to check individual Pods. You can do it as follows:

```
kubectl get pods
```

Expected output

```
NAME                                               READY   STATUS    RESTARTS      AGE
cluster1-haproxy-0                                 2/2     Running   0             44m
cluster1-haproxy-1                                 2/2     Running   0             44m
cluster1-haproxy-2                                 2/2     Running   0             44m
cluster1-mysql-0                                   3/3     Running   0             46m
cluster1-mysql-1                                   3/3     Running   2 (44m ago)   45m
cluster1-mysql-2                                   3/3     Running   2 (42m ago)   43m
cluster1-orc-0                                     2/2     Running   0             46m
cluster1-orc-1                                     2/2     Running   0             45m
cluster1-orc-2                                     2/2     Running   0             44m
percona-server-mysql-operator-7c984f7c9-mgwh4      1/1     Running   0             47m
```

The above command provides the following insights:

- `READY` indicates how many containers in the Pod are ready to serve the traffic. In the above example, `ps-cluster1-haproxy-0` container has all two containers ready (2/2). For an application to work properly, all containers of the Pod should be ready.
- `STATUS` indicates the current status of the Pod. The Pod should be in a `Running` state to confirm that the application is working as expected. You can find out other possible states in the official Kubernetes documentation.
- `RESTARTS` indicates how many times containers of Pod were restarted. This is impacted by the Container Restart Policy. In an ideal world, the restart count would be zero, meaning no issues from the beginning. If the restart count exceeds zero, it may be reasonable to check why it happens.
- `AGE`: Indicates how long the Pod is running. Any abnormality in this value needs to be checked.

You can find more details about a specific Pod using the `kubectl describe pods <pod-name>` command.

```
kubectl describe pods ps-cluster1-mysql-0
```

Expected output

```
...
Name:         ps-cluster1-mysql-0
Namespace:    default
...
Controlled By:  StatefulSet/ps-cluster1-mysql
Init Containers:
 mysql-init:
...
Containers:
 mysql:
...
   Restart Count:  0
   Limits:
     memory:  2G
   Requests:
     memory:   2G
   Liveness:   exec [/opt/percona/healthcheck liveness] delay=15s timeout=30s period=10s
#success=1 #failure=3
   Readiness:  exec [/opt/percona/healthcheck readiness] delay=30s timeout=3s period=5s
#success=1 #failure=3
   Startup:    exec [/opt/percona/bootstrap] delay=15s timeout=300s period=10s
#success=1 #failure=1
   Environment:
...
   Mounts:
...
Volumes:
...
Events:                      <none>
```

This gives a lot of information about containers, resources, container status and also events. So, describe output should be checked to see any abnormalities.

Exec into the containers

If you want to examine the contents of a container “in place” using remote access to it, you can use the `kubectl exec` command. It allows you to run any command or just open an interactive shell session in the container. Of course, you can have shell access to the container only if container supports it and has a “Running” state.

In the following examples we will access the container `mysql` of the `ps-cluster1-pxc-0` Pod.

- Run `date` command:

  ```
  kubectl exec -ti ps-cluster1-mysql-0 -c mysql -- date
  ```

  Expected output

  ```
  Thu Nov 24 10:01:17 UTC 2023
  ```

  You will see an error if the command is not present in a container. For example, trying to run the `time` command, which is not present in the container, by executing `kubectl exec -ti ps-cluster1-mysql-0 -c mysql -- time` would show the following result:

  ```
  OCI runtime exec failed: exec failed: unable to start container process:
  exec: "time": executable file not found in $PATH: unknown command
  terminated with exit code 126
  ```

- Print `/var/log/mysqld.log` file to a terminal:

  ```
  kubectl exec -ti ps-cluster1-mysql-0 -c mysql -- cat /var/log/mysqld.log
  ```

- The following example demonstrates how to enter an interactive shell in a container, run a few commands, and then exit:

  a. Start an interactive shell session inside the container:

  ```
  kubectl exec -ti ps-cluster1-mysql-0 -c mysql -- bash
  ```

  b. Inside the shell, you can run commands such as:

```
hostname
```

```
ls /var/log/mysqld.log
```

c. To exit the container shell, use:

```
exit
```

Avoid the restart-on-fail loop for Percona Server for MySQL containers

The restart-on-fail loop takes place when the container entry point fails (e.g. `mysqld` crashes). In such a situation, Pod is continuously restarting. Continuous restarts prevent to get console access to the container, and so a special approach is needed to make fixes.

You can prevent such infinite boot loop by putting the Percona Server for MySQL containers into the infinity loop *without* starting `mysqld`. This behavior of the container entry point is triggered by the presence of the `/var/lib/mysql/sleep-forever` file.

For example, you can do it for the `mysql` container of an appropriate Percona Server for MySQL instance as follows:

```
kubectl exec -it ps-cluster1-mysql-0 -c mysql -- sh -c 'touch
/var/lib/mysql/sleep-forever'
```

The instance will restart automatically and run in its usual way as soon as you remove this file (you can do it with a command similar to the one you have used to create the file, just substitute `touch` to `rm` in it).

Check the Events

Kubernetes Events always provide a wealth of information and should always be checked while troubleshooting issues.

Events can be checked by the following command

```
kubectl get events
```

Expected output

```
LAST SEEN   TYPE      REASON                  OBJECT
MESSAGE
3s          Normal    Provisioning            persistentvolumeclaim/datadir-ps-
cluster1-mysql-2        External provisioner is provisioning volume for claim
"default/datadir-ps-cluster1-mysql-2"
3s          Normal    ProvisioningSucceeded   persistentvolumeclaim/datadir-ps-
cluster1-mysql-2        Successfully provisioned volume pvc-fbd347c2-adf7-413b-86bb-
b5e381313cc0
...
```

Events capture many information happening at Kubernetes level and provide valuable information. By default, the ordering of events cannot be guaranteed. Use the following command to sort the output in a reverse chronological fashion.

```
kubectl get events --sort-by=".lastTimestamp"
```

Expected output

```
LAST SEEN   TYPE      REASON                  OBJECT
MESSAGE
33m         Warning   FailedScheduling          pod/ps-cluster1-mysql-0
0/1 nodes are available: pod has unbound immediate PersistentVolumeClaims. preemption:
0/1 nodes are available: 1 Preemption is not helpful for scheduling.
33m         Normal    Provisioning              persistentvolumeclaim/datadir-ps-
cluster1-mysql-0        External provisioner is provisioning volume for claim
"default/datadir-ps-cluster1-mysql-0"
33m         Normal    ProvisioningSucceeded     persistentvolumeclaim/datadir-ps-
cluster1-mysql-0        Successfully provisioned volume pvc-aad3d7cf-2bd4-4823-8e6f-
38b9a8528aaa
...
```

When there are too many events and there is a need of filtering output, tools like yq, jq can be used to filter specific items or know the structure of the events.

Example:

```
kubectl get events -o yaml | yq '.items[11]'
```

Expected output

```
apiVersion: v1
count: 1
eventTime: null
firstTimestamp: "2024-07-16T08:57:32Z"
involvedObject:
  apiVersion: v1
  kind: Pod
  name: ps-cluster1-mysql-0
  namespace: default
  resourceVersion: "623"
  uid: 689338c7-d5f7-4bfb-9f7e-ca1d13e782a3
kind: Event
lastTimestamp: "2024-07-16T08:57:32Z"
message: '0/1 nodes are available: pod has unbound immediate PersistentVolumeClaims.
preemption: 0/1 nodes are available: 1 Preemption is not helpful for scheduling.'
metadata:
  creationTimestamp: "2024-07-16T08:57:32Z"
  name: ps-cluster1-mysql-0.17e2a5c04f66588f
  namespace: default
  resourceVersion: "625"
  uid: 6d4a0eeb-8eea-4e1d-91f5-91fb795bd743
reason: FailedScheduling
reportingComponent: default-scheduler
reportingInstance: ""
source:
  component: default-scheduler
type: Warning
```

Flag `--field-selector` can be used to filter out the output as well. For example, the following command provides events of Pod only:

```
kubectl get events --field-selector involvedObject.kind=Pod
```

See the full list of supported `involvedObject` fields here.

More fields can be added to the field-selector flag for filtering events further. For example, the following command provides events of the `ps-cluster1-mysql-0` Pod:

```
kubectl get events --field-selector
involvedObject.kind=Pod,involvedObject.name=ps-cluster1-mysql-0
```

Expected output

```
LAST SEEN   TYPE      REASON            OBJECT                  MESSAGE
53m         Normal    Created           pod/ps-cluster1-mysql-0   Created container
mysql
53m         Normal    Started           pod/ps-cluster1-mysql-0   Started container
mysql
53m         Normal    Pulling           pod/ps-cluster1-mysql-0   Pulling image
"percona/percona-server-mysql-operator:0.8.0-backup"
...
```

Same way you can query events for other Kubernetes object (StatefulSet, Custom Resource, etc.) to investigate any problems to them:

```
kubectl get events --field-selector
involvedObject.kind=PerconaServerMySQL,involvedObject.name=ps-cluster1
```

Expected output

```
LAST SEEN   TYPE      REASON                    OBJECT                          MESSAGE
10m         Warning   AsyncReplicationNotReady  perconaservermysql/ps-cluster1  ps-
cluster1-mysql-1: [not_replicating]
...
```

Alternatively, you can see events for a specific object in the output of `kubectl describe` command:

```
kubectl describe ps ps-cluster1
```

Expected output

```
Name:         ps-cluster1
...
Events:
  Type     Reason                    Age                From           Message
  ----     ------                    ----               ----           -------
  Warning  AsyncReplicationNotReady  10m (x23 over 13m)    ps-controller  ps-cluster1-
mysql-1: [not_replicating]
...
```

Check `kubectl get events --help` to know about more options.

Note

It is important to note that events are stored in the etcd for only 60 minutes. Ensure that events are checked within 60 minutes of the issue. Kubernetes cluster administrators might also use event exporters for storing the events.

Check the Logs

Logs provide valuable information. It makes sense to check the logs of the database Pods and the Operator Pod. Following flags are helpful for checking the logs with the `kubectl logs` command:

Flag	Description
`--container=<container-name>`	Print log of a specific container in case of multiple containers in a Pod
`--follow`	Follows the logs for a live output
`--since=<time>`	Print logs newer than the specified time, for example: `--since="10s"`
`--timestamps`	Print timestamp in the logs (timezone is taken from the container)
`--previous`	Print previous instantiation of a container. This is extremely useful in case of container restart, where there is a need to check the logs on why the container restarted. Logs of previous instantiation might not be available in all the cases.

In the following examples we will access containers of the `ps-cluster1-mysql-0` Pod.

- Check logs of the `mysql` container:

  ```
  kubectl logs ps-cluster1-mysql-0 -c mysql
  ```

- Check logs of the `xtrabackup` container:

  ```
  kubectl logs ps-cluster1-mysql-0 -c xtrabackup
  ```

  See more about troubleshooting backups in the Troubleshoot backups and restores chapter.

- Filter logs of the `mysql` container which are not older than 600 seconds:

```
kubectl logs ps-cluster1-mysql-0 -c mysql --since=600s
```

- Check logs of a previous instantiation of the `mysql` container, if any:

```
kubectl logs ps-cluster1-mysql-0 -c mysql --previous
```

- Check logs of the `mysql` container for `ERROR` messages:

```
kubectl logs ps-cluster1-mysql-0 -c mysql | grep 'ERROR'
```

- Check logs of the Operator:

```
kubectl logs -l "app.kubernetes.io/name=percona-server-mysql-operator" --
tail=-1
```

- Check logs of the operator, parsing the output with jq JSON processor (requires `LOG_STRUCTURED` to be `true`):

```
kubectl logs -l "app.kubernetes.io/name=percona-server-mysql-operator" --
tail=-1 -f | jq -R 'fromjson?'
```

- Check logs of the HAProxy pod for configuration issues:

```
kubectl logs ps-cluster1-haproxy-0 | grep 'The custom config /etc/haproxy-
custom/haproxy.cfg is not valid and will be ignored.'
```

Troubleshoot backups and restores

You can troubleshoot failed backups or restores by checking the status of backup and restore objects, examining the jobs that executed them, and reviewing logs from the jobs and /or the pods created by the Jobs.

The overall troubleshooting workflow looks like this:

1. Check the object status. Use `kubectl get ps-backup` or `kubectl get ps-restore` to see the current state.
2. Review error details. Use `kubectl describe` to get the error message from the `State Description` field.
3. Examine job logs. Check the logs from the backup or restore job for detailed execution information.
4. Check source pod logs. For backups, review logs from the `xtrabackup` container in the source pod. For restores, view logs of the Pod created by the restore Job.
5. Verify configuration. Ensure storage configurations, credentials, and cluster settings are correct.

Refer to the following sections for details.

Backups

Check backup status

Start by checking the status of your backup objects:

```
kubectl get ps-backup -n <namespace>
```

This shows you all backup objects with their current state. The `STATE` column indicates whether a backup succeeded, failed, or is in progress.

Example output

```
NAME                                              STORAGE      DESTINATION
STATE       COMPLETED   AGE
backup1                                           aws-s3       s3://operator-testing/ps-
cluster1-2025-11-07-20:59:28-full   Succeeded   11m          11m
backup2                                           azure
Error                   10m
backup3                                           azure-blob   operator-testing/ps-
cluster1-2025-11-07-21:00:14-full        Succeeded   10m          10m
backup4                                           gcp-cs       s3://operator-testing/ps-
cluster1-2025-11-07-21:01:19-full   Succeeded   3m37s        9m29s
backup5                                           gcp-cs       s3://operator-testing/ps-
cluster1-2025-11-07-21:07:20-full   Succeeded   3m17s        3m28s
cron-ps-cluster1-aws-s3-20251107211029-45srk   aws-s3       s3://operator-testing/ps-
cluster1-2025-11-07-21:10:29-full   Succeeded
```

Check backup error details

When a backup fails, use `kubectl describe` to get detailed error information:

```
kubectl describe ps-backup <backup-name> -n <namespace>
```

The `Status` section contains the `State` and `State Description` fields that explain why the backup failed.

Example error output

```
Name:         backup2
Namespace:    default
...
Status:
  State:              Error
  State Description:  azure not found in spec.backup.storages in PerconaServerMySQL
CustomResource
Events:               <none>
```

Common error scenarios include:

- **Storage not found**: The storage name specified in the backup doesn't exist in the cluster configuration
- **Authentication failures**: Invalid credentials for accessing cloud storage
- **Network issues**: Problems connecting to the storage service

- **Insufficient permissions**: The backup job doesn't have permission to write to the storage location

Check backup Jobs

When a backup runs, the Operator creates a Kubernetes Job to execute it. You can view these Jobs to see which backups are running or have completed:

```
kubectl get jobs -n <namespace>
```

Backup jobs follow this naming pattern:

- `xb-<backup-name>-<storage-name>` for on-demand backups
- `xb-cron-<cluster-name>-<storage-name>-<timestamp>-<hash>-<storage-name>` for scheduled backups

Example output

```
NAME                                                STATUS     COMPLETIONS
DURATION   AGE
xb-backup1-aws-s3                                   Complete   1/1           14s
11m
xb-backup3-azure-blob                               Complete   1/1           12s
10m
xb-backup4-gcp-cs                                   Complete   1/1           5m52s
9m45s
xb-backup5-gcp-cs                                   Complete   1/1           10s
3m43s
xb-cron-ps-cluster1-aws-s3-20251107211029-45srk-aws-s3   Complete   1/1           12s
35s
```

View backup job logs

To see detailed logs from a backup job, use the job name:

```
kubectl logs job/<job-name> -n <namespace>
```

Find the name using the steps from the Check backup Jobs section

For example, to view logs from the `xb-backup1-aws-s3` job:

```
kubectl logs job/xb-backup1-aws-s3 -n <namespace>
```

These logs show the backup process execution, including Percona XtraBackup operations and any errors that occurred.

Example output

```
Defaulted container "xtrabackup" out of: xtrabackup, xtrabackup-init (init)
Trying to run backup backup1 on ps-cluster1-mysql-1.ps-cluster1-mysql.default
2025-11-07T20:59:33.889841-00:00 0 [Note] [MY-011825] [Xtrabackup] recognized server
arguments: --datadir=/var/lib/mysql
2025-11-07T20:59:33.890011-00:00 0 [Note] [MY-011825] [Xtrabackup] recognized client
arguments: --backup=1 --stream=xbstream --safe-slave-backup=1 --slave-info=1 --target-
dir=/backup/ --user=xtrabackup --password=*
xtrabackup version 8.4.0-4 based on MySQL server 8.4.0 Linux (x86_64) (revision id:
c584cb20)
2025-11-07T20:59:33.890062-00:00 0 [Note] [MY-011825] [Xtrabackup] Connecting to MySQL
server host: localhost, user: xtrabackup, password: set, port: not set, socket: not set
2025-11-07T20:59:33.903741-00:00 0 [Note] [MY-011825] [Xtrabackup] Using server version
8.4.6-6
2025-11-07T20:59:33.934529-00:00 0 [Note] [MY-011825] [Xtrabackup] Not checking slave
open temp tables for --safe-slave-backup because host is not a slave
2025-11-07T20:59:33.934783-00:00 0 [Note] [MY-011825] [Xtrabackup] Executing LOCK TABLES
FOR BACKUP ...
2025-11-07T20:59:33.940450-00:00 0 [Note] [MY-011825] [Xtrabackup] uses posix_fadvise().
2025-11-07T20:59:33.940508-00:00 0 [Note] [MY-011825] [Xtrabackup] cd to /var/lib/mysql
```

You can also view logs from the Pod created by the Job:

```
kubectl logs <pod-name> -n <namespace>
```

To find the Pod name, list pods and look for pods with names matching your backup job:

```
kubectl get pods -n <namespace> | grep xb-
```

Find the backup source Pod

Each backup object contains information about which MySQL Pod was used as the backup source. You can retrieve this information using:

```
kubectl get ps-backup <backup-name> -o jsonpath='{.status.backupSource}' -n
<namespace>
```

For example:

```
kubectl get ps-backup backup1 -o jsonpath='{.status.backupSource}'
```

This returns the fully qualified domain name (FQDN) of the source pod, such as `ps-cluster1-mysql-1.ps-cluster1-mysql.default`.

View backup logs from the source pod

The `xtrabackup` container in the source pod also contains backup logs. To view them, run:

```
kubectl logs <source-pod-name> -c xtrabackup -n <namespace>
```

For example, if the backup source is `ps-cluster1-mysql-1`:

```
kubectl logs ps-cluster1-mysql-1 -c xtrabackup -n <namespace>
```

These logs show the backup request received by the sidecar container, including the backup destination, storage type, and Percona XtraBackup commands executed.

Example output

```
2025-11-07T20:44:11Z    INFO    startServer     starting http server
2025-11-07T20:59:33Z    INFO    sidecar.create backup   Checking if backup exists
{"namespace": "default", "name": "backup1"}
2025-11-07T20:59:33Z    INFO    sidecar.create backup   Backup starting {"namespace":
"default", "name": "backup1", "destination": "ps-cluster1-2025-11-07-20:59:28-full",
"storage": "s3", "xtrabackupCmd": "/usr/bin/xtrabackup --backup --stream=xbstream --
safe-slave-backup --slave-info --target-dir=/backup/ --user=xtrabackup ", "xbcloudCmd":
"/usr/bin/xbcloud put --parallel=10 --curl-retriable-errors=7 --md5 --storage=s3 --s3-
bucket=operator-testing --s3-region=us-east-1   ps-cluster1-2025-11-07-20:59:28-full"}
2025-11-07T20:59:33.889841-00:00 0 [Note] [MY-011825] [Xtrabackup] recognized server
arguments: --datadir=/var/lib/mysql
2025-11-07T20:59:33.890011-00:00 0 [Note] [MY-011825] [Xtrabackup] recognized client
arguments: --backup=1 --stream=xbstream --safe-slave-backup=1 --slave-info=1 --target-
dir=/backup/ --user=xtrabackup --password=*
xtrabackup version 8.4.0-4 based on MySQL server 8.4.0 Linux (x86_64) (revision id:
c584cb20)
2025-11-07T20:59:33.890062-00:00 0 [Note] [MY-011825] [Xtrabackup] Connecting to MySQL
server host: localhost, user: xtrabackup, password: set, port: not set, socket: not set
2025-11-07T20:59:33.903741-00:00 0 [Note] [MY-011825] [Xtrabackup] Using server version
8.4.6-6
2025-11-07T20:59:33.934529-00:00 0 [Note] [MY-011825] [Xtrabackup] Not checking slave
open temp tables for --safe-slave-backup because host is not a slave
2025-11-07T20:59:33.934783-00:00 0 [Note] [MY-011825] [Xtrabackup] Executing LOCK TABLES
FOR BACKUP ...
```

Restores

Check restore status

To check the status of restore operations, run:

```
kubectl get ps-restore -n <namespace>
```

This shows all restore objects with their current state.

Example output

```
NAME       STATE       AGE
restore1   Error       5m56s
restore2   Succeeded   5m37s
```

Check restore error details

When a restore fails, use the `kubectl describe` command to get detailed error information:

```
kubectl describe ps-restore <restore-name> -n <namespace>
```

The `Status` section contains the `State` and `State Description` fields that explain why the restore failed.

Example error output

```
Name:         restore1
Namespace:    default
...
Status:
  State:              Error
  State Description:  PerconaServerMySQLBackup backup11 in namespace default is not
found
Events:               <none>
```

Example success output

```
Name:         restore2
Namespace:    default
...
Status:
  State:  Succeeded
Events:   <none>
```

Common restore error scenarios include:

- **Backup not found**: The backup specified in the restore doesn't exist
- **Storage access issues**: Problems reading from the backup storage location
- **Cluster state conflicts**: The cluster is not in a state that allows restore
- **Insufficient resources**: Not enough disk space or memory to complete the restore

Check restore jobs

When a restore runs, the Operator creates a Kubernetes Job to execute it. View these Jobs:

```
kubectl get jobs -n <namespace>
```

Restore jobs follow the naming pattern `xb-restore-<restore-name>`.

Example output

```
NAME                                                STATUS     COMPLETIONS
DURATION   AGE
xb-restore-restore2                                 Complete   1/1           23s
5m42s
```

View restore job logs

To see detailed logs from a restore Job, run:

```
kubectl logs job/<restore-job-name> -n <namespace>
```

For example:

```
kubectl logs job/xb-restore-restore2 -n <namespace>
```

Sample output

```
Defaulted container "xtrabackup" out of: xtrabackup, xtrabackup-init (init)
++ awk '/^xtrabackup version/{print $3}'
++ awk -F. '{print $1"."$2}'
++ xtrabackup --version
* XTRABACKUP_VERSION=8.4
* DATADIR=/var/lib/mysql
++ grep -c processor /proc/cpuinfo
* PARALLEL=4
* XBCLOUD_ARGS='--curl-retriable-errors=7 --parallel=4 '
* '[' -n true ']'
* [[ true == \f\a\l\s\e ]]
```

View logs from the Pod created by the restore Job

You can also view logs from the Pod created by the restore Job. To find the Pod name, list pods and look for pods with names matching your restore job:

```
kubectl get pods -n <namespace> | grep xb-restore
```

Then check the logs:

```
kubectl logs <restore-pod-name> -n <namespace>
```

Example output

```
Defaulted container "xtrabackup" out of: xtrabackup, xtrabackup-init (init)
++ awk '/^xtrabackup version/{print $3}'
++ awk -F. '{print $1"."$2}'
++ xtrabackup --version
+ XTRABACKUP_VERSION=8.4
+ DATADIR=/var/lib/mysql
++ grep -c processor /proc/cpuinfo
+ PARALLEL=4
+ XBCLOUD_ARGS='--curl-retriable-errors=7 --parallel=4 '
+ '[' -n true ']'
+ [[ true == \f\a\l\s\e ]]
```

Reference

Custom Resource reference

Custom Resource options reference

Percona Operator for MySQL uses Custom Resources to manage options for the various components of the cluster.

- `PerconaServerMySQL` Custom Resource with options for the cluster,
- `PerconaServerMySQLClusterSet` Custom Resource with options for InnoDB ClusterSet — see ClusterSet Resource options and cross-site replication,
- `PerconaServerMySQLBackup` Custom Resource contains options for Percona XtraBackup used to backup Percona Server for MySQL
- `PerconaServerMySQLRestore` Custom Resource contains options for restoring Percona Server for MySQL from backups.

PerconaServerMySQL Custom Resource options

Percona Server for MySQL managed by the Operator is configured via the spec section of the deploy/cr.yaml file.

The metadata part of PerconaServerMySQL Custom Resource contains the following keys:

- `name` (`ps-cluster1` by default) sets the name of your Percona Server for MySQL cluster; it should include only URL-compatible characters, not exceed 22 characters, start with an alphabetic character, and end with an alphanumeric character;
- `finalizers` subsection:
  - `percona.com/delete-mysql-pods-in-order` if present, activates the Finalizer which controls the proper Pods deletion order in case of the cluster deletion event (on by default).
  - `percona.com/delete-mysql-pvc` if present, activates the Finalizer which deletes Persistent Volume Claims for Percona Server for MySQL Pods after the cluster deletion event (off by default). It also triggers deletion of user Secrets.
  - `percona.com/delete-ssl` if present, activates the Finalizer which deletes objects, created for SSL (Secret, certificate, and issuer) after the cluster deletion event (off by default).

The top-level spec elements of the deploy/cr.yaml are the following ones:

Toplevel `spec` elements

The spec part of the deploy/cr.yaml file contains the following:

`crVersion`

Version of the Operator the Custom Resource belongs to.

Value type	Example
S string	`1.2.0`

`metadata.annotations`

The Kubernetes annotations metadata that you can set at a global level for all resources created by the Operator.

Value type	Example
label	`example-annotation: value`

`metadata.labels`

The Kubernetes labels metadata that you can set at a global level for all resources created by the Operator.

Value type	Example
label	`example-label: value`

`pause`

Pause/resume: setting it to `true` gracefully stops the cluster, and setting it to `false` after shut down starts the cluster back.

Value type	Example
boolean	`false`

`enableVolumeExpansion`

Enables or disables storage resizing with volume expansion.

Value type	Example
boolean	`false`

This option is deprecated and will be removed in version 1.5.0. Use the `storageScaling.enableVolumeScaling` option instead.

Storage scaling section

The `storageScaling` section in the deploy/cr.yaml file controls volume expansion and automatic storage resizing for MySQL PVCs. See Scale storage for usage details.

`storageScaling.enableVolumeScaling`

Enables or disables storage scaling / volume expansion with Volume Expansion capability.

Value type	Example
boolean	`false`

`storageScaling.enableExternalAutoscaling`

Enables or disables the use of external volume autoscaler. When disabled, the Operator uses its own expansion logic with Volume Expansion capability.

Value type	Example
boolean	`false`

`storageScaling.autoscaling.enabled`

Enables automatic storage resizing based on user-defined thresholds. Read more about this feature in Automatic storage resizing.

Value type	Example
boolean	`false`

`storageScaling.autoscaling.triggerThresholdPercent`

The percentage of the storage usage that triggers automatic resizing. Valid values are from 50 to 95. Default value is 80.

Value type	Example
int	`80`

`storageScaling.autoscaling.growthStep`

The amount to increase the storage during automatic resizing. Default value is `2Gi`.

Value type	Example
string	`2Gi`

`storageScaling.autoscaling.maxSize`

The maximum size to which storage can be automatically resized. Autoscaling stops when storage reaches this limit. Must be at least `1Gi`.

Value type	Example
string	`10Gi`

`initContainer.Image`

An alternative init image for the Operator.

Value type	Example
S string	`perconalab/percona-server-mysql-operator:1.2.0`

`initContainer.containerSecurityContext`

A custom Kubernetes Security Context for a Container used for the initial Operator installation.

Value type	Example

≡ subdoc	`privileged: false`
	`runAsUser: 1001`
	`runAsGroup: 1001`

`initContainer.resources.requests.memory`

The Kubernetes memory requests for an image used for the initial Operator installation.

Value type	Example
S string	`100M`

`initContainer.resources.requests.cpu`

Kubernetes CPU requests for an image used for the initial Operator installation.

Value type	Example
S string	`100m`

`initContainer.resources.limits.memory`

Kubernetes memory limits for an image used for the initial Operator installation.

Value type	Example
S string	`200M`

`initContainer.resources.limits.cpu`

Kubernetes CPU limits for an image used for the initial Operator installation.

Value type	Example
S string	`200m`

`secretsName`

A name for users secrets. When undefined, the Operator creates the Secrets object named in the format `<cluster-name>-secrets`. Otherwise, it uses the provided name.

Value type	Example
S string	`ps-cluster1-secrets`

`clusterServiceDNSSuffix`

A custom cluster domain suffix the Operator uses when constructing internal service hostnames. Use this when the Operator runs in a vcluster or in a cluster with a custom DNS domain so it can correctly resolve services. See Configure DNS suffix for service discovery for details.

Value type	Example
S string	`cluster.local`

`sslSecretName`

A secret with TLS certificate generated for *external* communications, see Transport Layer Security (TLS) for details. When undefined, the Operator creates the Secrets object named in the format `<cluster-name>-secrets-ssl`. Otherwise, it uses the provided name.

Value type	Example
S string	`ps-cluster1-ssl`

`ignoreAnnotations`

The list of annotations to be ignored by the Operator.

Value type	Example
≡ subdoc	`service.beta.kubernetes.io/aws-load-balancer-backend-protocol`

`ignoreLabels`

The list of labels to be ignored by the Operator.

Value type	Example
≡ subdoc	`rack`

`updateStrategy`

A strategy the Operator uses for upgrades.

Value type	Example
S string	`SmartUpdate`

`pause`

Pause/resume: setting it to `true` gracefully stops the cluster, and setting it to `false` after shut down starts the cluster back.

Value type	Example

boolean	`false`

`allowUnsafeConfigurations`

Prevents users from configuring a cluster with unsafe parameters such as starting a group replication cluster with less than 3 or more than 9 Percona Server for MySQL instances. **This option is deprecated and will be removed in future releases**. Use `unsafeFlags` subsection instead. Setting `allowUnsafeConfigurations` won't have any effect with the Operator version 0.8.0 and newer, and upgrading existing clusters with `allowUnsafeConfigurations=true` will cause everything under the unsafeFlags subsection set to `true`.

Value type	Example
boolean	`false`

Unsafe flags section

The `unsafeFlags` section in the deploy/cr.yaml file contains various configuration options to prevent users from configuring a cluster with unsafe parameters. *

Warning

Once unsafe configurations are enabled, you cannot revert these settings simply by setting the same keys back to `false`. Any such changes will be ignored, so enable these options with caution.

`unsafeFlags.backupNonReadyCluster`

Allows the Operator to run a backup from a healthy MySQL Pod when the cluster is not yet ready.

Value type	Example
boolean	`false`

`unsafeFlags.mysqlSize`

Allows users to start the cluster with less than 3 MySQL instances or with more than 9 (the maximum safe size).

Value type	Example
boolean	`false`

`unsafeFlags.proxy`

Allows users to configure a cluster with disabled proxy (both HAProxy and Router).

Value type	Example
boolean	`false`

`unsafeFlags.proxySize`

Allows users to set proxy (HAProxy or Router) size to a value less than 2 Pods.

Value type	Example
boolean	`false`

`unsafeFlags.orchestrator`

Allows users to configure a cluster with disabled Orchestrator even if asynchronous replication is turned on.

Value type	Example
boolean	`false`

`unsafeFlags.orchestratorSize`

Allows users to set orchestrator.size option to a value less than the minimum safe size (3).

Value type	Example
boolean	`false`

Extended cert-manager configuration section

The `tls` section in the deploy/cr.yaml file contains various configuration options for additional customization of the TLS cert-manager.

`tls.SANs`

Additional domains (SAN) to be added to the TLS certificate within the extended cert-manager configuration.

Value type	Example
subdoc	

`tls.issuerConf.name`

A cert-manager issuer name.

Value type	Example
string	`special-selfsigned-issuer`

`tls.issuerConf.kind`

A cert-manager issuer type.

Value type	Example
string	`ClusterIssuer`

`tls.issuerConf.group`

A cert-manager issuer group. Should be `cert-manager.io` for built-in cert-manager certificate issuers |

Value type	Example
S string	`cert-manager.io`

Upgrade options section

The `upgradeOptions` section in the deploy/cr.yaml file contains various configuration options to control Percona Server for MySQL version choice at the deployment time and during upgrades.

`upgradeOptions.versionServiceEndpoint`

The Version Service URL used to check versions compatibility for upgrade.

Value type	Example
S string	`https://check.percona.com`

`upgradeOptions.apply`

Specifies how images are picked up from the version service on initial start by the Operator. `Never` or `Disabled` will completely disable querying version service for images, otherwise it can be set to `Latest` or `Recommended` or to a specific version string of Percona Server for MySQL (e.g. `8.0.32-24`) that is wished to be version-locked (so that the user can control the version running) |

Value type	Example
S string	`Disabled`

Percona Server for MySQL section

The `mysql` section in the deploy/cr.yaml file contains general configuration options for the Percona Server for MySQL.

`mysql.clusterType`

The cluster type: `async` for Asynchronous replication, `group-replication` for Group Replication.

Value type	Example
1 int	`group-replication`

`mysql.bootstrap.mode`

Controls whether the Operator forms a Group Replication cluster automatically on first start. Use `manual` for replica sites in cross-site replication.

Value type	Example
S string	`manual`

Supported values:

Value	Meaning
`auto` (default)	Pod-0 runs the `dba.createCluster()` function when no peers exist. The same behavior is when the value for `bootstrap` is not set.
`manual`	Pod-0 configures MySQL for Group Replication but does not create a cluster. Pod-0 stays in the `NotReady` state until the ClusterSet controller runs `createReplicaCluster()`. Pod-1 and above do not start until Pod-0 is `Ready`. The Percona Server for MySQL cluster reports the `Initializing` state and the Operator sets the `AwaitingExternalBootstrap` status condition.

`mysql.autoRecovery`

Enables or disables the Operator from attempting to fix the issue in the event of a full cluster crash .

Value type	Example
boolean	`true`

`mysql.vaultSecretName`

Specifies the secret for the HashiCorp Vault to carry on Data at Rest Encryption.

Value type	Example
string	`ps-cluster1-vault`

`mysql.size`

The number of the Percona Server for MySQL instances. This setting is required.

Value type	Example
int	`3`

`mysql.image`

The Docker image of the Percona Server for MySQL used (actual image names for Percona Server for MySQL 8.0 and Percona Server for MySQL 5.7 can be found in the list of certified images).

Value type	Example
string	`percona/percona-server:8.0.46-37.1`

`mysql.imagePullPolicy`

The policy used to update images.

Value type	Example
string	`Always`

`mysql.runtimeClassName`

Specifies the name of the RuntimeClass resource used to define and select the container runtime configuration.

Value type	Example
S string	`image-rc`

`mysql.schedulerName`

The name of a Kubernetes scheduler used to assign MySQL Pods to Kubernetes nodes. The `default-scheduler` means `kube-scheduler` is used. You can define your custom schedulers here.

Value type	Example
S string	`default-scheduler`

`mysql.priorityClassName`

The name of the Kubernetes PriorityClass, which is a way to assign priority levels to pods, helping the scheduler decide which pods to schedule first and which ones to evict last when resources are tight.

Value type	Example
S string	`high-priority`

`mysql.nodeSelector`

Kubernetes nodeSelector.

Value type	Example

label	`disktype: ssd`

`mysql.serviceAccountName`

The Kubernetes Service Account for the MySQL Pods.

Value type	Example
string	`percona-server-mysql-operator-orchestrator`

`mysql.tolerations`

Specifies the Kubernetes tolerations applied to MySQL Pods allowing them to be scheduled on nodes with matching taints. Tolerations enable the Pod to tolerate specific node conditions, such as temporary unreachability or resource constraints, without being evicted immediately.

Value type	Example
subdoc	`node.alpha.kubernetes.io/unreachable`

`mysql.imagePullSecrets.name`

Specifies the Kubernetes imagePullSecrets for the MySQL image.

Value type	Example
string	`my-secret-1`

`mysql.initContainer.image`

An alternative init image for Percona Server for MySQL.

Value type	Example
string	`perconalab/percona-server-mysql-operator:1.2.0`

`mysql.initContainer.containerSecurityContext`

A custom Kubernetes Security Context for a Container used for the MySQL installation.

Value type	Example

subdoc	`privileged: false`
	`runAsUser: 1001`
	`runAsGroup: 1001`

`mysql.initContainer.resources.requests.memory`

The Kubernetes memory requests for an image used for the MySQL installation.

Value type	Example
string	`100M`

`mysql.initContainer.resources.requests.cpu`

Kubernetes CPU requests for an image used for the MySQL installation.

Value type	Example
string	`100m`

`mysql.initContainer.resources.limits.memory`

Kubernetes memory limits for an image used for the MySQL installation.

Value type	Example
string	`200M`

`mysql.initContainer.resources.limits.cpu`

Kubernetes CPU limits for an image used for the MySQL installation.

Value type	Example
S string	`200m`

`mysql.podDisruptionBudget.maxUnavailable`

The number of unavailable Pods your cluster can tolerate during voluntary disruption. It can be either an absolute value or a percentage.

To learn more, see podDisruptionBudgets.

Value type	Example
1 int	1

`mysql.podDisruptionBudget.minAvailable`

The number of Pods that must remain available during voluntary disruption. It can be either an absolute value or a percentage.

To learn more, see podDisruptionBudgets.

Value type	Example
1 int	0

`mysql.resources.requests.memory`

The Kubernetes memory requests for a Percona Server for MySQL container.

Value type	Example
S string	`512M`

`mysql.resources.requests.cpu`

Kubernetes CPU requests for an image used for a Percona Server for MySQL container.

Value type	Example
S string	`100m`

`mysql.resources.limits.memory`

Kubernetes memory limits for a Percona Server for MySQL container.

Value type	Example
S string	`1G`

`mysql.resources.limits.cpu`

Kubernetes CPU limits for a Percona Server for MySQL container.

Value type	Example
S string	`200m`

`mysql.startupProbe.initialDelaySeconds`

The number of seconds to wait before performing the startup probe.

Value type	Example
1 int	`15`

`mysql.startupProbe.timeoutSeconds`

The number of seconds after which the startup probe times out.

Value type	Example
1 int	`43200`

`mysql.startupProbe.periodSeconds`

How often to perform the startup probe. Measured in seconds.

Value type	Example
1 int	`10`

`mysql.startupProbe.successThreshold`

The number of successful probes required to mark the container successful.

Value type	Example
1 int	`1`

`mysql.startupProbe.failureThreshold`

The number of failed probes required to mark the container unready.

Value type	Example
1 int	`1`

`mysql.readinessProbe.initialDelaySeconds`

The number of seconds to wait before performing the first readiness probe.

Value type	Example
1 int	`30`

`mysql.readinessProbe.timeoutSeconds`

The number of seconds after which the readiness probe times out.

Value type	Example
1 int	`10`

`mysql.readinessProbe.periodSeconds`

How often to perform the readiness probe. Measured in seconds.

Value type	Example
1 int	`10`

`mysql.readinessProbe.successThreshold`

The number of successful probes required to mark the container successful.

Value type	Example
1 int	`1`

`mysql.readinessProbe.failureThreshold`

The number of failed probes required to mark the container unready.

Value type	Example
1 int	`3`

`mysql.livenessProbe.initialDelaySeconds`

The number of seconds to wait before performing the first liveness probe.

Value type	Example
1 int	`15`

`mysql.livenessProbe.timeoutSeconds`

The number of seconds after which the liveness probe times out.

Value type	Example
1 int	`10`

`mysql.livenessProbe.periodSeconds`

How often to perform the liveness probe. Measured in seconds.

Value type	Example
1 int	`10`

`mysql.livenessProbe.successThreshold`

The number of successful probes required to mark the container successful.

Value type	Example
1 int	`1`

`mysql.livenessProbe.failureThreshold`

The number of failed probes required to mark the container unhealthy.

Value type	Example
1 int	`3`

`mysql.env.name`

Name of an environment variable for MySQL Pods. The `BOOTSTRAP_READ_TIMEOUT` variable controls the timeout for bootstrapping the cluster.

Read more about defining environment variables in Kubernetes documentation.

Value type	Example
S string	`BOOTSTRAP_READ_TIMEOUT`

`mysql.env.value`

The value you set for the environment variables for a MySQL container.

Value type	Example
S string	`"600"`

`mysql.envFrom.secretRef.name`

Name of a Secret or a ConfigMap, key/values of which are used as environment variables for MySQL Pods.

Value type	Example
S string	`my-env-secret`

`mysql.affinity.antiAffinityTopologyKey`

The Operator topology key node anti-affinity constraint.

Value type	Example
S string	`kubernetes.io/hostname`

`mysql.affinity.advanced`

In cases where the Pods require complex tuning the advanced option turns off the `topologyKey` effect. This setting allows the standard Kubernetes affinity constraints of any complexity to be used.

Value type	Example
subdoc	

`mysql.topologySpreadConstraints.labelSelector.matchLabels`

The Label selector for the Kubernetes Pod Topology Spread Constraints.

Value type	Example
label	`app.kubernetes.io/name: percona-server`

`mysql.topologySpreadConstraints.maxSkew`

The degree to which Pods may be unevenly distributed under the Kubernetes Pod Topology Spread Constraints.

Value type	Example
int	1

`mysql.topologySpreadConstraints.topologyKey`

The key of node labels for the Kubernetes Pod Topology Spread Constraints.

Value type	Example
string	`kubernetes.io/hostname`

`mysql.topologySpreadConstraints.whenUnsatisfiable`

What to do with a Pod if it doesn't satisfy the Kubernetes Pod Topology Spread Constraints.

Value type	Example
string	`DoNotSchedule`

`mysql.exposePrimary.enabled`

Enable or disable exposing Percona Server for MySQL primary node with dedicated IP addresses.

Value type	Example
boolean	`false`

`mysql.exposePrimary.type`

The Kubernetes Service Type used for exposure.

Value type	Example
string	`ClusterIP`

`mysql.exposePrimary.allocateLoadBalancerNodePorts`

Determines if Kubernetes should allocate NodePorts for the LoadBalancer Service. This option is relevant only when the Service type is set to `LoadBalancer`.

Value type	Example
boolean	`false`

`mysql.exposePrimary.annotations`

The Kubernetes annotations.

Value type	Example
S string	`service.beta.kubernetes.io/aws-load-balancer-backend-protocol: tcp`, `service.beta.kubernetes.io/aws-load-balancer-type: nlb`

`mysql.exposePrimary.externalTrafficPolicy`

Specifies whether Service should route external traffic to cluster-wide (`Cluster`) or node-local (`Local`) endpoints; it can influence the load balancing effectiveness.

Value type	Example
S string	`Cluster`

`mysql.exposePrimary.internalTrafficPolicy`

Specifies whether Service should route internal traffic to cluster-wide (`Cluster`) or node-local (`Local`) endpoints; it can influence the load balancing effectiveness.

Value type	Example
S string	`Cluster`

`mysql.exposePrimary.labels`

Labels are key-value pairs attached to objects.

Value type	Example
label	`rack: rack-22`

`mysql.exposePrimary.loadBalancerSourceRanges`

The range of client IP addresses from which the load balancer should be reachable (if not set, there are no limitations).

Value type	Example
S string	`10.0.0.0/8`

`mysql.expose.enabled`

Enable or disable exposing Percona Server for MySQL nodes with dedicated IP addresses. This setting exposes every Pod in your cluster.

Value type	Example
boolean	`true`

`mysql.expose.type`

The Kubernetes Service Type used for exposure.

Value type	Example
S string	`ClusterIP`

`mysql.expose.allocateLoadBalancerNodePorts`

Determines if Kubernetes should allocate NodePorts for the LoadBalancer Service. This option is relevant only when the Service type is set to `LoadBalancer`.

Value type	Example
boolean	`false`

`mysql.expose.annotations`

The Kubernetes annotations.

Value type	Example

string	`service.beta.kubernetes.io/aws-load-balancer-backend-protocol: tcp`

`mysql.expose.externalTrafficPolicy`

Specifies whether Service should route external traffic to cluster-wide (`Cluster`) or node-local (`Local`) endpoints; it can influence the load balancing effectiveness.

Value type	Example
string	`Cluster`

`mysql.expose.internalTrafficPolicy`

Specifies whether Service should route internal traffic to cluster-wide (`Cluster`) or node-local (`Local`) endpoints; it can influence the load balancing effectiveness.

Value type	Example
string	`Cluster`

`mysql.expose.labels`

Labels are key-value pairs attached to objects.

Value type	Example
label	`rack: rack-22`

`mysql.expose.loadBalancerSourceRanges`

The range of client IP addresses from which the load balancer should be reachable (if not set, there are no limitations).

Value type	Example

S string	`10.0.0.0/8`

`mysql.volumeSpec.emptyDir`

Starts a Pod with an empty temporary directory on the Kubernetes node. This directory exists as long as the Pod runs. Data is deleted when the Pod is deleted or moved to another node.

Value type	Example
S string	`{}`

`mysql.volumeSpec.hostPath.path`

Specifies a path on the host node's filesystem that will be mounted into your Pod when the Pod starts. Enables Pods to share files or access the host resources. Data persists as long as it exists on the host, independent of the Pod. Using the hostPath volume type presents many security risks.

Value type	Example
S string	`/data`

`mysql.volumeSpec.hostPath.type`

Specifies a type for the hostPath volume.

Value type	Example
S string	`Directory`

`mysql.volumeSpec.persistentVolumeClaim.storageClassName`

Requests a specific storage class for the Persistent Volume Claim. This will cause the PVC to match the right storage class if the cluster has StorageClasses enabled by the admin.

Value type	Example

S string	`standard`

`mysql.volumeSpec.persistentVolumeClaim.accessModes`

Specify a specific access mode for a Persistent Volume. Kubernetes uses volume access modes to match PersistentVolumeClaims and PersistentVolumes.

Value type	Example
S string	`"ReadWriteOnce"`

`mysql.volumeSpec.persistentVolumeClaim.resources.requests.storage`

The Kubernetes PersistentVolumeClaim size for the Percona Server for MySQL.

Value type	Example
S string	`2Gi`

`mysql.gracePeriod`

Specifies the maximum time, in seconds, the Operator allows for a pod to shut down gracefully after receiving a termination signal before it is forcefully killed. This ensures critical cleanup tasks, like flushing data, can complete.

Value type	Example
1 int	`600`

`mysql.configuration`

The `my.cnf` file options to be passed to Percona Server for MySQL instances.

Value type	Example
S string	

```
[mysqld]
max_connections=250
```

`mysql.sidecars.image`

Image for the custom sidecar container for Percona Server for MySQL Pods.

Value type	Example
S string	`busybox`

`mysql.sidecars.command`

Command for the custom sidecar container for Percona Server for MySQL Pods.

Value type	Example
[:] array	`["sleep", "30d"]`

`mysql.sidecars.name`

Name of the custom sidecar container for Percona Server for MySQL Pods.

Value type	Example
S string	`my-sidecar-1`

`mysql.sidecars.volumeMounts.mountPath`

Mount path of the custom sidecar container volume for Replica Set Pods.

Value type	Example
S string	`/volume1`

`mysql.sidecars.resources.requests.memory`

The Kubernetes memory requests for a Percona Server for MySQL sidecar container.

Value type	Example
S string	`16M`

`mysql.sidecars.volumeMounts.name`

Name of the custom sidecar container volume for Replica Set Pods.

Value type	Example
S string	`sidecar-volume-claim`

`mysql.sidecarVolumes`

Volume specification for the custom sidecar container volume for Percona Server for MySQL Pods.

Value type	Example
subdoc	

`mysql.sidecarPVCs`

Persistent Volume Claim for the custom sidecar container volume for Replica Set Pods |

Value type	Example
subdoc	

HAProxy subsection

The `proxy.haproxy` subsection in the deploy/cr.yaml file contains configuration options for the HAProxy service.

`proxy.haproxy.enabled`

Enables or disables load balancing with HAProxy Services.

Value type	Example
boolean	`true`

`proxy.haproxy.size`

The number of the HAProxy Pods to provide load balancing. Safe configuration should have 2 or more. This setting is required.

Value type	Example
int	`3`

`proxy.haproxy.image`

HAProxy Docker image to use.

Value type	Example
string	`percona/perconalab-xtradb-cluster-operator:1.2.0-haproxy`

`proxy.haproxy.imagePullPolicy`

The policy used to update images.

Value type	Example

string	`Always`

`proxy.haproxy.schedulerName`

The name of a Kubernetes scheduler used to assign HAProxy Pods to Kubernetes nodes. The `default-scheduler` means `kube-scheduler` is used. You can define your custom schedulers here.

Value type	Example
string	`default-scheduler`

`proxy.haproxy.priorityClassName`

The name of the Kubernetes PriorityClass, which is a way to assign priority levels to pods, helping the scheduler decide which pods to schedule first and which ones to evict last when resources are tight.

Value type	Example
string	`high-priority`

`proxy.haproxy.nodeSelector`

Kubernetes nodeSelector. It enables you to define node labels thereby ensuring that Pods will be scheduled onto nodes that have each of the labels you specify.

Value type	Example
label	`disktype: ssd`

`proxy.haproxy.serviceAccountName`

The Kubernetes Service Account for the HAProxy Pods.

Value type	Example
S string	`percona-server-mysql-operator-orchestrator`

`proxy.haproxy.podDisruptionBudget.maxUnavailable`

The number of unavailable Pods your cluster can tolerate during voluntary disruption. It can be either an absolute value or a percentage.

To learn more, see podDisruptionBudgets.

Value type	Example
1 int	1

`proxy.haproxy.podDisruptionBudget.minAvailable`

The number of Pods that must remain available during voluntary disruption. It can be either an absolute value or a percentage.

To learn more, see podDisruptionBudgets.

Value type	Example
1 int	0

`proxy.haproxy.runtimeClassName`

Specifies the name of the RuntimeClass resource used to define and select the container runtime configuration.

Value type	Example
S string	`image-rc`

`proxy.haproxy.tolerations`

Specifies the Kubernetes tolerations applied to HAProxy Pods allowing them to be scheduled on nodes with matching taints. Tolerations enable the Pod to tolerate specific node conditions, such as resource constraints being temporary unreachable, without being evicted immediately.

Value type	Example
subdoc	`node.alpha.kubernetes.io/unreachable`

`proxy.haproxy.imagePullSecrets.name`

Specifies the Kubernetes imagePullSecrets for the HAProxy image.

Value type	Example
string	`my-secret-1`

`proxy.haproxy.resources.requests.memory`

The Kubernetes memory requests for the main HAProxy container.

Value type	Example
string	`1G`

`proxy.haproxy.resources.requests.cpu`

Kubernetes CPU requests for the main HAProxy container.

Value type	Example
string	`600m`

`proxy.haproxy.resources.limits.memory`

Kubernetes memory limits for the main HAProxy container.

Value type	Example
S string	`1G`

`proxy.haproxy.resources.limits.cpu`

Kubernetes CPU limits for the main HAProxy container.

Value type	Example
S string	`700m`

`proxy.haproxy.sidecarResources.CONTAINER-NAME.requests.memory`

The Kubernetes memory requests for an HAProxy sidecar container (for example, `mysql-monit`).

Value type	Example
S string	`64Mi`

`proxy.haproxy.sidecarResources.CONTAINER-NAME.requests.cpu`

Kubernetes CPU requests for an HAProxy sidecar container (for example, `mysql-monit`).

Value type	Example
S string	`50m`

`proxy.haproxy.sidecarResources.CONTAINER-NAME.limits.memory`

Kubernetes memory limits for an HAProxy sidecar container (for example, `mysql-monit`).

Value type	Example
S string	`128Mi`

`proxy.haproxy.sidecarResources.CONTAINER-NAME.limits.cpu`

Kubernetes CPU limits for an HAProxy sidecar container (for example, `mysql-monit`).

Value type	Example
S string	`100m`

`proxy.haproxy.env.name`

Name of an environment variable for HAProxy.

Value type	Example
S string	`HA_CONNECTION_TIMEOUT`

`proxy.haproxy.env.value`

Value of an environment variable for HAProxy.

Value type	Example
S string	`"1000"`

`proxy.haproxy.envFrom.secretRef.name`

Name of a Secret with environment variables for HAProxy.

Value type	Example
S string	`haproxy-env-secret`

`proxy.haproxy.startupProbe.initialDelaySeconds`

The number of seconds to wait before performing the startup probe.

Value type	Example
1 int	`15`

`proxy.haproxy.startupProbe.timeoutSeconds`

The number of seconds after which the startup probe times out.

Value type	Example
1 int	`43200`

`proxy.haproxy.startupProbe.periodSeconds`

How often to perform the startup probe. Measured in seconds.

Value type	Example
1 int	`10`

`proxy.haproxy.startupProbe.successThreshold`

The number of successful probes required to mark the container successful.

Value type	Example
1 int	`1`

`proxy.haproxy.startupProbe.failureThreshold`

The number of failed probes required to mark the container unready.

Value type	Example
1 int	`1`

`proxy.haproxy.readinessProbe.timeoutSeconds`

Number of seconds after which the readiness probe times out.

Value type	Example
1 int	3

`proxy.haproxy.readinessProbe.periodSeconds`

How often (in seconds) to perform the readiness probe.

Value type	Example
1 int	5

`proxy.haproxy.readinessProbe.successThreshold`

Minimum consecutive successes for the readiness probe to be considered successful after having failed.

Value type	Example
1 int	3

`proxy.haproxy.readinessProbe.failureThreshold`

When the readiness probe fails, Kubernetes will try this number of times before marking the Pod Unready.

Value type	Example
1 int	1

`proxy.haproxy.livenessProbe.timeoutSeconds`

Number of seconds after which the liveness probe times out.

Value type	Example
1 int	3

`proxy.haproxy.livenessProbe.periodSeconds`

How often (in seconds) to perform the liveness probe.

Value type	Example
1 int	5

`proxy.haproxy.livenessProbe.successThreshold`

Minimum consecutive successes for the liveness probe to be considered successful after having failed.

Value type	Example
1 int	3

`proxy.haproxy.livenessProbe.failureThreshold`

When the liveness probe fails, Kubernetes will try this number of times before marking the Pod Unready.

Value type	Example
1 int	1

`proxy.haproxy.gracePeriod`

Specifies the maximum time, in seconds, the Operator allows for a pod to shut down gracefully after receiving a termination signal before it is forcefully killed. This ensures critical cleanup tasks, like flushing data, can complete.

Value type	Example
1 int	`30`

`proxy.haproxy.configuration`

The custom HAProxy configuration file contents.

Value type	Example
S string	

`proxy.haproxy.antiAffinityTopologyKey`

The Operator topology key node anti-affinity constraint.

Value type	Example
S string	`kubernetes.io/hostname`

`proxy.haproxy.affinity.advanced`

If available it makes a topologyKey node affinity constraint to be ignored.

Value type	Example
subdoc	

`proxy.haproxy.topologySpreadConstraints.labelSelector.matchLabels`

The Label selector for the Kubernetes Pod Topology Spread Constraints.

Value type	Example
label	`app.kubernetes.io/name: percona-server`

`proxy.haproxy.topologySpreadConstraints.maxSkew`

The degree to which Pods may be unevenly distributed under the Kubernetes Pod Topology Spread Constraints.

Value type	Example
int	1

`proxy.haproxy.topologySpreadConstraints.topologyKey`

The key of node labels for the Kubernetes Pod Topology Spread Constraints.

Value type	Example
string	`kubernetes.io/hostname`

`proxy.haproxy.topologySpreadConstraints.whenUnsatisfiable`

What to do with a Pod if it doesn't satisfy the Kubernetes Pod Topology Spread Constraints.

Value type	Example
string	`DoNotSchedule`

`proxy.haproxy.expose.type`

The Kubernetes Service Type used for HAProxy exposure.

Value type	Example

string	`ClusterIP`

`proxy.haproxy.expose.allocateLoadBalancerNodePorts`

Determines if Kubernetes should allocate NodePorts for the LoadBalancer Service. This option is relevant only when the Service type is set to `LoadBalancer`.

Value type	Example
boolean	`false`

`proxy.haproxy.expose.annotations`

The Kubernetes annotations for HAProxy.

Value type	Example
string	`service.beta.kubernetes.io/aws-load-balancer-backend-protocol: tcp`

`proxy.haproxy.expose.externalTrafficPolicy`

Specifies whether Service for HAProxy should route external traffic to cluster-wide (`Cluster`) or node-local (`Local`) endpoints; it can influence the load balancing effectiveness.

Value type	Example
string	`Cluster`

`proxy.haproxy.expose.internalTrafficPolicy`

Specifies whether Service for HAProxy should route internal traffic to cluster-wide (`Cluster`) or node-local (`Local`) endpoints; it can influence the load balancing effectiveness.

Value type	Example
string	`Cluster`

`proxy.haproxy.expose.labels`

Labels are key-value pairs attached to objects for HAProxy.

Value type	Example
label	`rack: rack-22`

`proxy.haproxy.expose.loadBalancerIP`

The static IP-address for the load balancer. This field is deprecated in Kubernetes 1.24+ and removed from the Operator starting with version 0.10.0. If you have defined it, refer to the Kubernetes documentation for recommendations.

Value type	Example
string	`127.0.0.1`

`proxy.haproxy.expose.loadBalancerSourceRanges`

The range of client IP addresses from which the load balancer should be reachable (if not set, there is no limitations) |

Value type	Example
string	`10.0.0.0/8`

`proxy.haproxy.containerSecurityContext`

A custom Kubernetes Security Context for a Container used for the HAProxy installation.

Value type	Example
subdoc	

```
privileged: false
runAsUser: 1001
runAsGroup: 1001
```

`proxy.haproxy.podSecurityContext`

A custom Kubernetes Security Context for a Pod to be used instead of the default one.

Value type	Example
subdoc	`fsGroup: 1001` `supplementalGroups: [1001, 1002, 1003]`

Router subsection

The `proxy.router` subsection in the deploy/cr.yaml file contains configuration options for the MySQL Router, which can act as a proxy for Group replication.

`proxy.router.enabled`

Enables or disables MySQL Router.

Value type	Example
boolean	`false`

`proxy.router.size`

The number of the Router Pods to provide routing to MySQL Servers. This setting is required.

Value type	Example
int	`3`

`proxy.router.image`

Router Docker image to use.

Value type	Example
S string	`perconalab/percona-server-mysql-operator:1.2.0-router`

`proxy.router.imagePullPolicy`

The policy used to update images.

Value type	Example
S string	`Always`

`proxy.router.runtimeClassName`

Specifies the name of the RuntimeClass resource used to define and select the container runtime configuration.

Value type	Example
S string	`image-rc`

`proxy.router.tolerations`

Specifies the Kubernetes tolerations applied to Router Pods allowing them to be scheduled on nodes with matching taints. Tolerations enable the Pod to tolerate specific node conditions, such as temporary unreachability or resource constraints, without being evicted immediately.

Value type	Example
subdoc	`node.alpha.kubernetes.io/unreachable`

`proxy.router.imagePullSecrets.name`

Specifies the Kubernetes imagePullSecrets for the Router image.

Value type	Example
S string	`my-secret-1`

`proxy.router.initContainer.image`

An alternative init image for MySQL Router Pods.

Value type	Example
S string	`perconalab/percona-server-mysql-operator:1.2.0`

`proxy.router.initContainer.containerSecurityContext`

A custom Kubernetes Security Context for a Container for the image used for MySQL Router Pods installation.

Value type	Example

subdoc	`privileged: false`
	`runAsUser: 1001`
	`runAsGroup: 1001`

`proxy.router.initContainer.resources.requests.memory`

The Kubernetes memory requests for an image used for MySQL Router Pods installation.

Value type	Example
S string	`100M`

`proxy.router.initContainer.resources.requests.cpu`

Kubernetes CPU requests for an image used for MySQL Router Pods installation.

Value type	Example
S string	`100m`

`proxy.router.initContainer.resources.limits.memory`

Kubernetes memory limits for an image used for MySQL Router Pods installation.

Value type	Example
S string	`200M`

`proxy.router.initContainer.resources.limits.cpu`

Kubernetes CPU limits for an image used for MySQL Router Pods installation.

Value type	Example
S string	`200m`

`proxy.router.env.name`

Name of an environment variable for MySQL Router Pods. Read more about defining environment variables in Kubernetes documentation.

Value type	Example
S string	`MY_ENV`

`proxy.router.env.value`

Value of an environment variable for MySQL Router Pods.

Value type	Example

S string	`"1000"`

`proxy.router.envFrom.secretRef.name`

Name of a Secret or a ConfigMap, key/values of which are used as environment variables for MySQL Router Pods.

Value type	Example
S string	`my-env-secret`

`proxy.router.podDisruptionBudget.maxUnavailable`

The number of unavailable Pods your cluster can tolerate during voluntary disruption. It can be either an absolute value or a percentage.

To learn more, see podDisruptionBudgets.

Value type	Example
1 int	1

`proxy.router.podDisruptionBudget.minAvailable`

The number of Pods that must remain available during voluntary disruption. It can be either an absolute value or a percentage.

To learn more, see podDisruptionBudgets.

Value type	Example
1 int	0

`proxy.router.ports.name`

The name for a custom or an existing port for the MySQL Router Service.

Value type	Example
S string	`http`

`proxy.router.ports.port`

The port exposed by the MySQL Router service to the outside world or other components.

Value type	Example
S string	`8443`

`proxy.router.ports.targetPort`

The port inside the Pod/container where MySQL Router is actually listening. When a client connects to the external port, Kubernetes forwards that traffic to the `targetPort` value on the backend Pod. A zero (0) value means Kubernetes uses the default internal port or does not do the remapping.

Value type	Example
S string	`0`

`proxy.router.resources.requests.memory`

The Kubernetes memory requests for MySQL Router container.

Value type	Example
S string	`256M`

`proxy.router.resources.requests.cpu`

Kubernetes CPU requests for MySQL Router container.

Value type	Example

string	`100m`

`proxy.router.resources.limits.memory`

Kubernetes memory limits for MySQL Router container.

Value type	Example
string	`256M`

`proxy.router.resources.limits.cpu`

Kubernetes CPU limits for MySQL Router container.

Value type	Example
string	`200m`

`proxy.router.affinity.antiAffinityTopologyKey`

The Operator topology key node anti-affinity constraint.

Value type	Example
string	`kubernetes.io/hostname`

`proxy.router.affinity.advanced`

In cases where the Pods require complex tuning the advanced option turns off the `topologyKey` effect. This setting allows the standard Kubernetes affinity constraints of any complexity to be used.

Value type	Example
subdoc	

`proxy.router.topologySpreadConstraints.labelSelector.matchLabels`

The Label selector for the Kubernetes Pod Topology Spread Constraints.

Value type	Example
label	`app.kubernetes.io/name: percona-server`

`proxy.router.topologySpreadConstraints.maxSkew`

The degree to which Pods may be unevenly distributed under the Kubernetes Pod Topology Spread Constraints.

Value type	Example
int	1

`proxy.router.topologySpreadConstraints.topologyKey`

The key of node labels for the Kubernetes Pod Topology Spread Constraints.

Value type	Example
string	`kubernetes.io/hostname`

`proxy.router.topologySpreadConstraints.whenUnsatisfiable`

What to do with a Pod if it doesn't satisfy the Kubernetes Pod Topology Spread Constraints.

Value type	Example
string	`DoNotSchedule`

`proxy.router.gracePeriod`

Specifies the maximum time, in seconds, the Operator allows for a pod to shut down gracefully after receiving a termination signal before it is forcefully killed. This ensures critical cleanup tasks, like flushing data, can complete.

Value type	Example
1 int	`30`

`proxy.router.configuration`

Custom configuration options to be passed to MySQL Router.

Value type	Example

S string	|
	`[default]`
	`logging_folder=/tmp/router/log`
	`[logger]`
	`level=DEBUG`

`proxy.router.expose.type`

The Kubernetes Service Type used for MySQL Router instances exposure.

Value type	Example
S string	`ClusterIP`

`proxy.router.expose.allocateLoadBalancerNodePorts`

Determines if Kubernetes should allocate NodePorts for the LoadBalancer Service. This option is relevant only when the Service type is set to `LoadBalancer`.

Value type	Example
boolean	`false`

`proxy.router.expose.annotations`

The Kubernetes annotations for MySQL Router.

Value type	Example
string	`service.beta.kubernetes.io/aws-load-balancer-backend-protocol: tcp`

`proxy.router.expose.externalTrafficPolicy`

Specifies whether Service for MySQL Router should route external traffic to cluster-wide (`Cluster`) or node-local (`Local`) endpoints; it can influence the load balancing effectiveness.

Value type	Example
string	`Cluster`

`proxy.router.expose.internalTrafficPolicy`

Specifies whether Service for MySQL Router should route internal traffic to cluster-wide (`Cluster`) or node-local (`Local`) endpoints; it can influence the load balancing effectiveness.

Value type	Example
string	`Cluster`

`proxy.router.expose.labels`

Labels are key-value pairs attached to objects for MySQL Router.

Value type	Example

label	`rack: rack-22`

`proxy.router.expose.loadBalancerIP`

The static IP-address for the load balancer. This field is deprecated in Kubernetes 1.24+ and removed from the Operator starting with version 0.10.0. If you have defined it, refer to the Kubernetes documentation for recommendations.

Value type	Example
string	`127.0.0.1`

`proxy.router.expose.loadBalancerSourceRanges`

The range of client IP addresses from which the load balancer should be reachable (if not set, there is no limitations) |

Value type	Example
string	`10.0.0.0/8`

Orchestrator section

The `orchestrator` section in the deploy/cr.yaml file contains configuration options for the Orchestrator - a replication topology manager, used if asynchronous replication is turned on.

`orchestrator.enabled`

Enables or disables the Orchestrator.

Value type	Example
boolean	`true`

`orchestrator.size`

The number of the Orchestrator Pods to provide load balancing. This setting is required.

Value type	Example
1 int	`3`

`orchestrator.image`

Orchestrator Docker image to use.

Value type	Example
S string	`perconalab/percona-server-mysql-operator:1.2.0-orchestrator`

`orchestrator.imagePullPolicy`

The policy used to update images.

Value type	Example
S string	`Always`

`orchestrator.runtimeClassName`

Specifies the name of the RuntimeClass resource used to define and select the container runtime configuration.

Value type	Example
S string	`image-rc`

`orchestrator.schedulerName`

The name of a Kubernetes scheduler used to assign Orchestrator Pods to Kubernetes nodes. The `default-scheduler` means `kube-scheduler` is used. You can define your custom schedulers here.

Value type	Example
S string	`default-scheduler`

`orchestrator.priorityClassName`

The name of the Kubernetes PriorityClass, which is a way to assign priority levels to pods, helping the scheduler decide which pods to schedule first and which ones to evict last when resources are tight.

Value type	Example
S string	`high-priority`

`orchestrator.nodeSelector`

Kubernetes nodeSelector. It enables you to define node labels thereby ensuring that Pods will be scheduled onto nodes that have each of the labels you specify.

Value type	Example
label	`disktype: ssd`

`orchestrator.startupProbe.initialDelaySeconds`

The number of seconds to wait before performing the startup probe.

Value type	Example
1 int	`15`

`orchestrator.startupProbe.timeoutSeconds`

The number of seconds after which the startup probe times out.

Value type	Example
1 int	`43200`

`orchestrator.startupProbe.periodSeconds`

How often to perform the startup probe. Measured in seconds.

Value type	Example
1 int	`10`

`orchestrator.startupProbe.successThreshold`

The number of successful probes required to mark the container successful.

Value type	Example
1 int	`1`

`orchestrator.startupProbe.failureThreshold`

The number of failed probes required to mark the container unready.

Value type	Example
1 int	`1`

`orchestrator.readinessProbe.timeoutSeconds`

Number of seconds after which the readiness probe times out.

Value type	Example
1 int	3

`orchestrator.readinessProbe.periodSeconds`

How often (in seconds) to perform the readiness probe.

Value type	Example
1 int	5

`orchestrator.readinessProbe.successThreshold`

Minimum consecutive successes for the readiness probe to be considered successful after having failed.

Value type	Example
1 int	3

`orchestrator.readinessProbe.failureThreshold`

When the readiness probe fails, Kubernetes will try this number of times before marking the Pod Unready.

Value type	Example
1 int	1

`orchestrator.livenessProbe.timeoutSeconds`

Number of seconds after which the liveness probe times out.

Value type	Example

1 int	3

`orchestrator.livenessProbe.periodSeconds`

How often (in seconds) to perform the liveness probe.

Value type	Example
1 int	5

`orchestrator.livenessProbe.successThreshold`

Minimum consecutive successes for the liveness probe to be considered successful after having failed.

Value type	Example
1 int	3

`orchestrator.livenessProbe.failureThreshold`

When the liveness probe fails, Kubernetes will try this number of times before marking the Pod Unready.

Value type	Example
1 int	1

`orchestrator.env.name`

Name of an environment variable for Orchestrator Pods. Read more about defining environment variables in Kubernetes documentation.

Value type	Example
S string	`MY_ENV`

`orchestrator.env.value`

Value of an environment variable for Orchestrator Pods.

Value type	Example
string	`"1000"`

`orchestrator.envFrom.secretRef.name`

Name of a Secret or a ConfigMap, key/values of which are used as environment variables for Orchestrator Pods.

Value type	Example
string	`my-env-secret`

`orchestrator.tolerations`

Specifies the Kubernetes tolerations applied to Orchestrator Pods allowing them to be scheduled on nodes with matching taints. Tolerations enable the Pod to tolerate specific node conditions, such as temporary unreachability or resource constraints, without being evicted immediately.

Value type	Example
subdoc	`node.alpha.kubernetes.io/unreachable`

`orchestrator.imagePullSecrets.name`

Specifies the Kubernetes imagePullSecrets for the Orchestrator image.

Value type	Example
string	`my-secret-1`

`orchestrator.serviceAccountName`

The Kubernetes Service Account for the Orchestrator Pods.

Value type	Example
string	`percona-server-mysql-operator-orchestrator`

`orchestrator.initContainer.image`

An alternative init image for Orchestrator Pods.

Value type	Example
string	`perconalab/percona-server-mysql-operator:1.2.0`

`orchestrator.initContainer.containerSecurityContext`

A custom Kubernetes Security Context for a Container for the image used for Orchestrator Pods installation.

Value type	Example

subdoc	`privileged: false`
	`runAsUser: 1001`
	`runAsGroup: 1001`

`orchestrator.initContainer.resources.requests.memory`

The Kubernetes memory requests for an image used for Orchestrator Pods installation.

Value type	Example
string	`100M`

`orchestrator.initContainer.resources.requests.cpu`

Kubernetes CPU requests for an image used for Orchestrator Pods installation.

Value type	Example
S string	`100m`

`orchestrator.initContainer.resources.limits.memory`

Kubernetes memory limits for an image used for Orchestrator Pods installation.

Value type	Example
S string	`200M`

`orchestrator.initContainer.resources.limits.cpu`

Kubernetes CPU limits for an image used for Orchestrator Pods installation.

Value type	Example
S string	`200m`

`orchestrator.affinity.antiAffinityTopologyKey`

The Operator topology key node anti-affinity constraint.

Value type	Example
S string	`kubernetes.io/hostname`

`orchestrator.affinity.advanced`

In cases where the Pods require complex tuning the advanced option turns off the `topologyKey` effect. This setting allows the standard Kubernetes affinity constraints of any complexity to be used.

Value type	Example
subdoc	

`orchestrator.topologySpreadConstraints.labelSelector.matchLabels`

The Label selector for the Kubernetes Pod Topology Spread Constraints.

Value type	Example
label	`app.kubernetes.io/name: percona-server`

`orchestrator.topologySpreadConstraints.maxSkew`

The degree to which Pods may be unevenly distributed under the Kubernetes Pod Topology Spread Constraints.

Value type	Example
int	1

`orchestrator.topologySpreadConstraints.topologyKey`

The key of node labels for the Kubernetes Pod Topology Spread Constraints.

Value type	Example
string	`kubernetes.io/hostname`

`orchestrator.topologySpreadConstraints.whenUnsatisfiable`

What to do with a Pod if it doesn't satisfy the Kubernetes Pod Topology Spread Constraints.

Value type	Example

string	`DoNotSchedule`

`orchestrator.gracePeriod`

Specifies the maximum time, in seconds, the Operator allows for a pod to shut down gracefully after receiving a termination signal before it is forcefully killed. This ensures critical cleanup tasks, like flushing data, can complete.

Value type	Example
int	`30`

`orchestrator.expose.type`

The Kubernetes Service Type used to expose the Orchestrator HTTP API (port `3000`). The default `ClusterIP` keeps the API reachable only inside the cluster.

When you change this option to `LoadBalancer` or `NodePort`, the Orchestrator API becomes reachable from outside the cluster. Starting with Operator version 1.2.0, when `spec.crVersion` is `1.2.0` or higher, all API requests must authenticate. To learn more about authentication, refer to the Orchestrator HTTP API authentication section.

Value type	Example
string	`ClusterIP`

`orchestrator.expose.allocateLoadBalancerNodePorts`

Determines if Kubernetes should allocate NodePorts for the LoadBalancer Service. This option is relevant only when the Service type is set to `LoadBalancer`.

Value type	Example
boolean	`false`

`orchestrator.expose.annotations`

The Kubernetes annotations for the Orchestrator.

Value type	Example
string	`service.beta.kubernetes.io/aws-load-balancer-backend-protocol: tcp`

`orchestrator.expose.externalTrafficPolicy`

Specifies whether Service for the Orchestrator should route external traffic to cluster-wide (`Cluster`) or node-local (`Local`) endpoints; it can influence the load balancing effectiveness.

Value type	Example
string	`Cluster`

`orchestrator.expose.internalTrafficPolicy`

Specifies whether Service for the Orchestrator should route internal traffic to cluster-wide (`Cluster`) or node-local (`Local`) endpoints; it can influence the load balancing effectiveness.

Value type	Example
string	`Cluster`

`orchestrator.expose.labels`

Labels are key-value pairs attached to objects for the Orchestrator.

Value type	Example
label	`rack: rack-22`

`orchestrator.expose.loadBalancerSourceRanges`

The range of client IP addresses from which the load balancer should be reachable (if not set, there is no limitations).

Value type	Example
string	`10.0.0.0/8`

`orchestrator.containerSecurityContext`

A custom Kubernetes Security Context for a Container used for the Orchestrator installation.

Value type	Example

subdoc	`privileged: false`
	`runAsUser: 1001`
	`runAsGroup: 1001`

`orchestrator.podSecurityContext`

A custom Kubernetes Security Context for a Pod to be used instead of the default one.

Value type	Example
subdoc	`fsGroup: 1001` `supplementalGroups: [1001, 1002, 1003]`

`orchestrator.podDisruptionBudget.maxUnavailable`

The number of unavailable Pods your cluster can tolerate during voluntary disruption. It can be either an absolute value or a percentage.

To learn more, see podDisruptionBudgets.

Value type	Example
int	1

`orchestrator.podDisruptionBudget.minAvailable`

The number of Pods that must remain available during voluntary disruption. It can be either an absolute value or a percentage.

To learn more, see podDisruptionBudgets.

Value type	Example
1 int	0

`orchestrator.resources.requests.memory`

The Kubernetes memory requests for an Orchestrator container.

Value type	Example
S string	`128M`

`orchestrator.resources.requests.cpu`

Kubernetes CPU requests for an Orchestrator container.

Value type	Example
S string	`100m`

`orchestrator.resources.limits.memory`

Kubernetes memory limits for an Orchestrator container.

Value type	Example
S string	`256M`

`orchestrator.resources.limits.cpu`

Kubernetes CPU limits for an Orchestrator container.

Value type	Example
string	`200m`

`orchestrator.volumeSpec.persistentVolumeClaim.resources.requests.storage`

The Kubernetes PersistentVolumeClaim size for the Orchestrator |

Value type	Example
string	`1Gi`

`orchestrator.configuration`

Custom Orchestrator options to merge into the Orchestrator configuration. For example, `FailMasterPromotionOnLagMinutes` or `RecoveryPeriodBlockSeconds` are used to tune failover behavior per cluster. The value must be a JSON object passed as a string. You cannot override keys managed by the Operator, such as Raft topology, topology TLS, HTTP authentication, failover hooks, alias detection queries, and storage paths. Changing them is silently ignored.

Updating this field triggers a rolling restart of the Orchestrator Pods.

Value type	Example
string	`'{"FailMasterPromotionOnLagMinutes": 10}'`

PMM section

The `pmm` section in the deploy/cr.yaml file contains configuration options for Percona Monitoring and Management.

`pmm.enabled`

Enables or disables monitoring Percona Server for MySQL with PMM.

Value type	Example
boolean	`false`

`pmm.image`

PMM client Docker image to use.

Value type	Example
string	`percona/pmm-client:3.8.1`

`pmm.imagePullPolicy`

The policy used to update images.

Value type	Example
string	`Always`

`pmm.mysqlParams`

Enables to pass MySQL parameters to PMM. For example, to change the number of tables (from the default of 1000) beyond which per-table statistics collection is disabled.

Value type	Example
string	`"--disable-tablestats-limit=2000"`

`pmm.readinessProbe.initialDelaySeconds`

The number of seconds to wait before performing the first readiness probe.

Value type	Example
1 int	`15`

`pmm.readinessProbe.timeoutSeconds`

The number of seconds after which the readiness probe times out.

Value type	Example
1 int	`15`

`pmm.readinessProbe.periodSeconds`

How often to perform the readiness probe. Measured in seconds.

Value type	Example
1 int	`30`

`pmm.readinessProbe.successThreshold`

The number of successful probes required to mark the container successful.

Value type	Example
1 int	`1`

`pmm.readinessProbe.failureThreshold`

The number of failed probes required to mark the container unready.

Value type	Example
1 int	`5`

`pmm.livenessProbe.initialDelaySeconds`

The number of seconds to wait before performing the first liveness probe.

Value type	Example
1 int	`300`

`pmm.livenessProbe.timeoutSeconds`

The number of seconds after which the liveness probe times out.

Value type	Example
1 int	`5`

`pmm.livenessProbe.periodSeconds`

How often to perform the liveness probe. Measured in seconds.

Value type	Example
1 int	`10`

`pmm.livenessProbe.successThreshold`

The number of successful probes required to mark the container successful.

Value type	Example
1 int	`1`

`pmm.livenessProbe.failureThreshold`

The number of failed probes required to mark the container unhealthy.

Value type	Example
1 int	`3`

`pmm.resources.requests.memory`

The Kubernetes memory requests for a PMM container.

Value type	Example
S string	`150M`

`pmm.resources.requests.cpu`

Kubernetes CPU requests for a PMM container.

Value type	Example
S string	`300m`

`pmm.resources.limits.memory`

Kubernetes memory limits for a PMM container.

Value type	Example
S string	`256M`

`pmm.resources.limits.cpu`

Kubernetes CPU limits for a PMM container.

Value type	Example
S string	`400m`

`pmm.serverHost`

Address of the PMM Server to collect data from the cluster.

Value type	Example
S string	`monitoring-service`

`pmm.customClusterName`

A custom name to define for a cluster. PMM Server uses this name to properly parse the metrics and display them on dashboards. Using a custom name is useful for clusters deployed in different data centers - PMM Server connects them and monitors them as one deployment. Another use case is for clusters deployed with the same name in different namespaces - PMM treats each cluster separately.

Value type	Example
S string	`cluster1-custom`

Backup section

The `backup` section in the deploy/cr.yaml file contains the following configuration options for the regular Percona XtraDB Cluster backups.

`backup.enabled`

Enables or disables making backups.

Value type	Example
boolean	`true`

`backup.encryptionKeySecret.name`

The name of the Kubernetes Secret that stores the key used to encrypt backups before they are stored in object storage. When set, all backup storages use this key unless you specify a different Secret for a specific storage.

Value type	Example
S string	`my-encryption-key`

`backup.encryptionKeySecret.key`

The key within the Secret that holds the encryption key value. Defaults to `encryptionKey`. This is a global setting and applies to all storages unless overridden for a specific storage.

Value type	Example
S string	`encryptionKey`

`backup.pitr.enabled`

Enables or disables point-in-time recovery functionality.

Value type	Example
boolean	`true`

`backup.binlogServer.size`

Controls the number of Percona Binarylog Server Pods for binlog collection. Defaults to 1. The Custom Resource allows only `1` (larger values are rejected). To learn more, see Point-in-time recovery.

Value type	Example
1 int	`1`

`backup.binlogServer.image`

The Docker image to use to deploy Percona Binarylog Server.

Value type	Example
S string	`perconalab/percona-binlog-server:0.2.1`

`backup.binlogServer.imagePullPolicy`

The policy the `kubelet` uses to pull the Percona Binarylog Server image.

Value type	Example
S string	`Always`

`backup.binlogServer.imagePullSecrets.name`

The name of a Secret the `kubelet` uses to authenticate to a private image registry. You can list several Secrets as an array of objects with the `name` key.

Value type	Example
S string	`my-secret-1`

`backup.binlogServer.serverId`

The unique server ID that Percona Binarylog Server uses when connecting to MySQL as a replication client for binlog collection.

Value type	Example
1 int	`100`

`backup.binlogServer.storage.s3.bucket`

The name of the bucket on AWS S3 or S3-compatible storage where binlogs are streamed.

Value type	Example
S string	`S3-BACKUP-BUCKET-NAME-HERE`

`backup.binlogServer.storage.s3.credentialsSecret`

The Kubernetes Secret for binlog storage. It should contain `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` keys.

Value type	Example
S string	`ps-cluster1-s3-credentials`

`backup.binlogServer.storage.s3.endpointUrl`

The URL to access the bucket on S3-compatible storage. Not needed for AWS S3.

Value type	Example
S string	`https://s3.amazonaws.com`

`backup.binlogServer.storage.s3.prefix`

The path prefix (folder) in the bucket where binlogs are stored. You cannot change the prefix after you configured the Binlog Server.

Value type	Example
S string	`PREFIX_NAME`

`backup.binlogServer.storage.s3.region`

The region of the bucket. Required for Amazon S3 and for S3-compatible storage.

Value type	Example
S string	`us-west-2`

`backup.binlogServer.connectTimeout`

Timeout in seconds for establishing a connection to MySQL.

Value type	Example
1 int	`30`

`backup.binlogServer.readTimeout`

The maximum time in seconds the Binlog Server waits to read data from the MySQL instance.

Value type	Example
1 int	`30`

`backup.binlogServer.writeTimeout`

The maximum time in seconds the Binlog Server waits to write data to a remote server.

Value type	Example
1 int	`30`

`backup.binlogServer.idleTime`

The maximum time in seconds the Binlog Server stays in idle mode before trying to reconnect.

Value type	Example
1 int	`30`

`backup.binlogServer.affinity.antiAffinityTopologyKey`

The Kubernetes topologyKey constraint for node anti-affinity on Percona Binarylog Server Pods.

Value type	Example
string	`kubernetes.io/hostname`

`backup.binlogServer.affinity.advanced`

In cases where the pods require complex tuning the advanced option turns off the topologykey effect. This setting allows the standard Kubernetes affinity constraints of any complexity to be used.

Value type	Example
subdoc	

`backup.binlogServer.checkpointInterval`

How often the Binlog Server writes checkpoints - the snapshot of its state.

Value type	Example
string	`30s`

`backup.binlogServer.checkpointSize`

Defines the size threshold for checkpoints. Default is 16M.

Value type	Example
string	`16M`

`backup.binlogServer.containerSecurityContext`

A custom Kubernetes security context for a container for the Percona Binlog Server container.

Value type	Example

subdoc	`privileged: false`
	`runAsUser: 1001`
	`runAsGroup: 1001`

`backup.binlogServer.env`

Custom environment variables for the Percona Binarylog Server container.

Value type	Example
subdoc	`[]`

`backup.binlogServer.envFrom`

The source for the bulk-import of all key-value pairs from a ConfigMap or Secret as environment variables for the Percona Binarylog Server container.

Value type	Example
subdoc	`[]`

`backup.binlogServer.initContainer.image`

The container image for the Binlog Server init container.

Value type	Example
string	`perconalab/percona-server-mysql-operator:main`

`backup.binlogServer.initContainer.containerSecurityContext`

A custom Kubernetes security context for a container for the Binlog Server init container.

Value type	Example

subdoc	`privileged: false`
	`runAsUser: 1001`
	`runAsGroup: 1001`

`backup.binlogServer.initContainer.resources.requests.memory`

Kubernetes memory requests for the Binlog Server init container.

Value type	Example
string	`200M`

`backup.binlogServer.initContainer.resources.requests.cpu`

Kubernetes CPU requests for the Binlog Server init container.

Value type	Example
string	`200m`

`backup.binlogServer.initContainer.resources.limits.memory`

Kubernetes memory limits for the Binlog Server init container.

Value type	Example
string	`100M`

`backup.binlogServer.initContainer.resources.limits.cpu`

Kubernetes CPU limits for the Binlog Server init container.

Value type	Example
string	`100m`

`backup.binlogServer.initImage`

An alternative init image for the Percona Binarylog Server Pod. Leave empty to use the default one.

Value type	Example
S string	`""`

`backup.binlogServer.logLevel`

Log level for the Percona Binarylog Server. Defaults to `info`.

Value type	Example
S string	`info`

`backup.binlogServer.podSecurityContext`

A custom Kubernetes Pod security context for Percona Binarylog Server Pods.

Value type	Example

subdoc	`fsGroup: 1001`
	`supplementalGroups:`
	`  - 1001`

`backup.binlogServer.resources.requests.memory`

Kubernetes memory requests for the Percona Binarylog Server container.

Value type	Example
S string	`200M`

`backup.binlogServer.resources.requests.cpu`

Kubernetes CPU requests for the Percona Binarylog Server container.

Value type	Example
S string	`200m`

`backup.binlogServer.resources.limits.memory`

Kubernetes memory limits for the Percona Binarylog Server container.

Value type	Example
S string	`100M`

`backup.binlogServer.resources.limits.cpu`

Kubernetes CPU limits for the Percona Binarylog Server container.

Value type	Example
S string	`100m`

`backup.binlogServer.rewriteFileSize`

Specifies the maximum binlog file size for rewrite. Defaults to `128M`. Set to an empty string to use the default.

Value type	Example
S string	`128M`

`backup.binlogServer.sslMode`

The TLS mode for the Binlog Server connection to MySQL. Defaults to `verify_identity`.

Value type	Example
S string	`verify_identity`

`backup.sourcePod`

Specifies the MySQL instance Pod to take a backup from. When defined, takes precedence, regardless of the cluster type (async or group-replication) and topology. Applies both to scheduled and on-demand backups.

Asynchronous replication clusters that consist of more than one Pod and have the Orchestrator disabled must have the `sourcePod` defined for the Operator to make backups. Otherwise, the Operator fails to start a backup and reports an error.

Value type	Example
S string	`ps-cluster1-mysql-0`

`backup.image`

The Percona XtraBackup Docker image to use for the backup.

Value type	Example
S string	`percona/percona-server-mysql-operator:1.2.0-backup`

`backup.imagePullPolicy`

The policy used to update images.

| **Value** | string | | **Example** | `Always` |

`backup.imagePullSecrets.name`

The Kubernetes ImagePullSecret.

Value type	Example

string	`my-secret-1`

`backup.initContainer.image`

An alternative init image for Percona XtraBackup Pods.

Value type	Example
string	`perconalab/percona-server-mysql-operator:1.2.0`

`backup.initContainer.containerSecurityContext`

A custom Kubernetes Security Context for a Container for the image used for Percona XtraBackup Pods installation.

Value type	Example

subdoc	`privileged: false`
	`runAsUser: 1001`
	`runAsGroup: 1001`

`backup.initContainer.resources.requests.memory`

The Kubernetes memory requests for an image used for Percona XtraBackup Pods installation.

Value type	Example
string	`100M`

`backup.initContainer.resources.requests.cpu`

Kubernetes CPU requests for an image used for Percona XtraBackup Pods installation.

Value type	Example
string	`100m`

`backup.initContainer.resources.limits.memory`

Kubernetes memory limits for an image used for Percona XtraBackup Pods installation.

Value type	Example
S string	`200M`

`backup.initContainer.resources.limits.cpu`

Kubernetes CPU limits for an image used for Percona XtraBackup Pods installation.

Value type	Example
S string	`200m`

`backup.containerSecurityContext`

A custom Kubernetes Security Context for the `xtrabackup` container to be used instead of the default one.

Value type	Example
subdoc	`privileged: true`

`backup.backoffLimit`

The number of retries to make a backup (by default, 6 retries are made).

Value type	Example
1 int	`6`

`backup.storages.STORAGE-NAME.type`

The cloud storage type used for backups. The following types are supported: `s3`, `gcs` and `azure`.

Value type	Example
S string	`s3`

`backup.storages.STORAGE-NAME.verifyTLS`

Enable or disable verification of the storage server TLS certificate. Disabling it may be useful e.g. to skip TLS verification for private S3-compatible storage with a self-issued certificate.

Value type	Example
boolean	`true`

`backup.storages.STORAGE-NAME.encryptionKeySecret.name`

The name of the Kubernetes Secret that contains the key used to encrypt backups for this storage location. When set, it takes precedence over `backup.encryptionKeySecret.name`.

Value type	Example
S string	`my-encryption-key`

`backup.storages.STORAGE-NAME.encryptionKeySecret.key`

The key within the Secret that holds the encryption key value. Defaults to `encryptionKey`. It takes precedence over the `backup.encryptionKeySecret.key` setting.

Value type	Example
S string	`encryptionKey`

`backup.storages.STORAGE-NAME.nodeSelector`

Kubernetes nodeSelector.

Value type	Example
label	`disktype: ssd`

`backup.storages.STORAGE-NAME.resources.requests.memory`

The Kubernetes memory requests for a Percona XtraBackup container.

Value type	Example
string	`1G`

`backup.storages.STORAGE-NAME.resources.requests.cpu`

Kubernetes CPU requests for a Percona XtraBackup container.

Value type	Example
string	`600m`

`backup.storages.STORAGE-NAME.affinity.nodeAffinity`

The Operator node affinity constraint.

Value type	Example
subdoc	

`backup.storages.STORAGE-NAME.topologySpreadConstraints.labelSelector.matchLabels`

The Label selector for the Kubernetes Pod Topology Spread Constraints.

Value type	Example
label	`app.kubernetes.io/name: percona-server`

`backup.storages.STORAGE-NAME.topologySpreadConstraints.maxSkew`

The degree to which Pods may be unevenly distributed under the Kubernetes Pod Topology Spread Constraints.

Value type	Example
int	1

`backup.storages.STORAGE-NAME.topologySpreadConstraints.topologyKey`

The key of node labels for the Kubernetes Pod Topology Spread Constraints.

Value type	Example
string	`kubernetes.io/hostname`

`backup.storages.STORAGE-NAME.topologySpreadConstraints.whenUnsatisfiable`

What to do with a Pod if it doesn't satisfy the Kubernetes Pod Topology Spread Constraints.

Value type	Example
string	`DoNotSchedule`

`backup.storages.STORAGE-NAME.tolerations`

Kubernetes Pod tolerations.

Value type	Example
subdoc	

`backup.storages.STORAGE-NAME.schedulerName`

The Kubernetes Scheduler.

Value type	Example
string	`mycustom-scheduler`

`backup.storages.STORAGE-NAME.priorityClassName`

The Kubernetes Pod priority class.

Value type	Example
string	`high-priority`

`backup.storages.STORAGE-NAME.containerSecurityContext`

A custom Kubernetes Security Context for a Container to be used instead of the default one.

Value type	Example
subdoc	`privileged: true`

`backup.storages.STORAGE-NAME.podSecurityContext`

A custom Kubernetes Security Context for a Pod to be used instead of the default one.

Value type	Example
subdoc	`fsGroup: 1001` `supplementalGroups: [1001, 1002, 1003]`

`backup.storages.STORAGE-NAME.runtimeClassName`

Specifies the name of the RuntimeClass resource used to define and select the container runtime configuration for backup and restore jobs associated with the specific storage.

Value type	Example
S string	`image-rc`

`backup.storages.STORAGE-NAME.containerOptions.env`

The environment variables set as key-value pairs for the backup container.

Value type	Example
subdoc	`- name: CUSTOM_ENV` `value: “some-value”`

`backup.storages.STORAGE-NAME.containerOptions.args.xtrabackup`

Custom command line options for the `xtrabackup` Percona XtraBackup tool.

Value type	Example
subdoc	`- “–someflag=abc”`

`backup.storages.STORAGE-NAME.containerOptions.args.xbcloud`

Custom command line options for the `xbcloud` Percona XtraBackup tool.

Value type	Example
subdoc	`- “–someflag=abc”`

`backup.storages.STORAGE-NAME.containerOptions.args.xbstream`

Custom command line options for the `xbstream` Percona XtraBackup tool

Value type	Example
subdoc	`- "–someflag=abc"`

`backup.storages.STORAGE-NAME.annotations`

The Kubernetes annotations.

Value type	Example
label	`testName: scheduled-backup`

`backup.storages.STORAGE-NAME.labels`

Labels are key-value pairs attached to objects.

Value type	Example
label	`backupWorker: 'True'`

`backup.storages.STORAGE-NAME.s3.bucket`

The Amazon S3 bucket name for backups.

Value type	Example
string	

`backup.storages.STORAGE-NAME.s3.region`

The AWS region to use. Please note **this option is mandatory** for Amazon and all S3-compatible storages.

Value type	Example
S string	`us-west-2`

`backup.storages.STORAGE-NAME.s3.prefix`

The path (sub-folder) to the backups inside the bucket.

Value type	Example
S string	`""`

`backup.storages.STORAGE-NAME.s3.credentialsSecret`

The Kubernetes secret for backups. It should contain `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` keys.

Value type	Example
S string	`my-cluster-name-backup-s3`

`backup.storages.STORAGE-NAME.s3.endpointUrl`

The endpoint URL of the S3-compatible storage to be used (not needed for the original Amazon S3 cloud)

Value type	Example
S string	

`backup.storages.STORAGE-NAME.azure.container`

The name of the Microsoft Azure blob container for backups.

Value type	Example

string	`""`

`backup.storages.STORAGE-NAME.azure.prefix`

The path (sub-folder) to the backups inside the container.

Value type	Example
string	`""`

`backup.storages.STORAGE-NAME.azure.endpointUrl`

The endpoint URL of the S3-compatible storage to be used (not needed for the original Amazon S3 cloud)

Value type	Example
string	`https://accountName.blob.core.windows.net`

`backup.storages.STORAGE-NAME.azure.credentialsSecret`

The Kubernetes secret for backups. It should contain the `AZURE_STORAGE_ACCOUNT_NAME` and the `AZURE_STORAGE_ACCOUNT_KEY` keys.

Value type	Example
string	`my-cluster-name-backup-azure`

`backup.storages.STORAGE-NAME.gcs.bucket`

The name of the storage bucket.

Value type	Example
string	`bucket-name`

`backup.storages.STORAGE-NAME.gcs.prefix`

The path to the data directory in the bucket. If undefined, backups are stored in the bucket's root directory.

Value type	Example
S string	`prefix-name`

`backup.storages.STORAGE-NAME.gcs.endpointUrl`

The URL to access the data in Google Cloud Storage.

Value type	Example
S string	`"https://storage.googleapis.com"`

`backup.storages.STORAGE-NAME.gcs.credentialsSecret`

The Kubernetes secret for backups. It should contain the `ACCESS_KEY_ID` and the `SECRET_ACCESS_KEY` keys.

Value type	Example
S string	`ps-cluster1-gcp-credentials`

`backup.schedule.name`

Name of the scheduled backup.

Value type	Example
S string	`sat-night-backup`

`backup.schedule.schedule`

Scheduled time of the backup, specified in the crontab format.

Value type	Example
S string	`0 0 \* \* 6`

`backup.schedule.keep`

The amount of most recent backups to store. Older backups are automatically deleted. Set `keep` to zero or completely remove it to disable automatic deletion of backups. Note that `keep` is ignored for incremental backups.

Value type	Example
1 int	`3`

`backup.schedule.storageName`

The name of the storage for the backups configured in the `storages` subsection.

Value type	Example
S string	`s3-us-west`

`backup.schedule.type`

The backup type. Supported values: `full`, `incremental`. When undefined, defaults to `full`.

Value type	Example
S string	`full`

Percona Toolkit section

The `toolkit` section in the deploy/cr.yaml file contains configuration options for Percona Toolkit.

`toolkit.image`

Percona Toolkit client Docker image to use.

Value type	Example
S string	`percona/pmm-client:3.8.1`

`toolkit.imagePullPolicy`

The policy used to update images.

Value type	Example
S string	`Always`

`toolkit.resources.requests.memory`

The Kubernetes memory requests for a Percona Toolkit container.

Value type	Example
S string	`150M`

`toolkit.resources.requests.cpu`

Kubernetes CPU requests for a Percona Toolkit container.

Value type	Example
S string	`100m`

`toolkit.resources.limits.memory`

Kubernetes memory limits for a Percona Toolkit container.

Value type	Example
S string	`256M`

`toolkit.resources.limits.cpu`

Kubernetes CPU limits for a Percona Toolkit container

Value type	Example
S string	`400m`

ClusterSet Resource options

The `PerconaServerMySQLClusterSet` Custom Resource defines cross-site replication between Group Replication clusters as a MySQL InnoDB ClusterSet. Use the short name `ps-clusterset`.

This document describes all available options. For status fields and conditions, see Custom resource statuses.

`apiVersion`

Specifies the API version of the Custom Resource.

Value type	Example
S string	`ps.percona.com/v1`

`kind`

Defines the type of resource: `PerconaServerMySQLClusterSet`.

`metadata`

The metadata section identifies the ClusterSet object. It includes the following keys:

- `name` — Name of the ClusterSet resource.
- `namespace` — Namespace where the ClusterSet Custom Resource and the `mysqlshell-runner` Pod run.
- `finalizers` — ensure safe deletion of resources in Kubernetes under certain conditions. This subsection includes the following finalizers:
- `percona.com/clusterset-dissolve` — Runs `.dissolve()` on the InnoDB ClusterSet before the Custom Resource is deleted. Underlying clusters continue as standalone InnoDB Clusters.

`spec`

This section contains the ClusterSet configuration.

`spec.primaryCluster`

Name of the InnoDB cluster that should serve writes. Must match exactly one `spec.clusters[].innodbClusterName`. Must contain only alphanumeric characters (max 63).

Editing this field triggers a **planned switchover** when the current primary is reachable. For a **forced failover** when the current primary is unreachable, you must also set the `spec.unsafeFlags.forcedFailover` to `true`.

Value type	Example
S string	`pscluster1`

The `spec.unsafeFlags` subsection

Groups opt-in flags for destructive ClusterSet operations. Both default to `false`.

`spec.unsafeFlags.forcedFailover`

When `true`, permits `forcePrimaryCluster()` if you change `spec.primaryCluster` while the current primary is unreachable.

Value type	Example
boolean	`false`

Warning

Forced failover can promote a replica while the old primary may still accept writes, causing split-brain and lost transactions. Set this only when you are certain the old primary cannot recover.

`spec.unsafeFlags.forcedClusterRemoval`

When `true`, permits `removeCluster(..., {force: true})` for an unreachable replica removed from `spec.clusters[]`.

Value type	Example
boolean	`false`

> **Warning**
>
> Forced removal abandons unreplicated transactions on the removed cluster. The cluster may require manual cleanup or a full rebuild.

`spec.sslMode`

SSL mode for ClusterSet async replication channels between primary and replica clusters.

Value	Meaning
`AUTO`	TLS is enabled if the instance supports it; otherwise disabled. **Default.**
`DISABLED`	TLS is disabled for replication channels.
`REQUIRED`	TLS is required for replication channels.
`VERIFY_CA`	Like `REQUIRED`, plus verify the peer certificate against configured CA certificates. Primary and replica certificates must be signed by the same CA.
`VERIFY_IDENTITY`	Like `VERIFY_CA`, plus verify the peer certificate matches the connection host (SAN must match `spec.clusters[].endpoints[].host`).

Value type	Example
S string	`AUTO`

`spec.credentialsSecret.name`

The name of a Secret in the same namespace that holds the `clusterset` user password.

Value type	Example
S string	`ps-cluster1-secrets`

`spec.credentialsSecret.key`

The field in the Secret that holds the `clusterset` user password value. This password must match on every participating cluster. Defaults to `clusterset`.

Value type	Example
S string	`clusterset`

The `spec.clusters` subsection

List of InnoDB clusters that participate in the ClusterSet. Minimum 1 entry (primary only at bootstrap); maximum 10. Each cluster name must be unique.

`spec.clusters[].innodbClusterName`

Logical name of the InnoDB cluster within the ClusterSet. Used in `spec.primaryCluster`, status, and MySQL Shell AdminAPI calls. The name must match the value from `status.innodbClusterName` of a corresponding `PerconaServerMySQL` Custom Resource.

Alphanumeric only, max 63 characters. Immutable in practice — rename by remove and re-add.

Value type	Example
S string	`pscluster1`

`spec.clusters[].endpoints`

List of `host:port` pairs the controller uses to reach MySQL members of this cluster. Minimum 1, maximum 9 per cluster. The controller uses the first reachable endpoint. Multiple entries improve resilience to single-member failures.

Each `host` must be unique across **all** clusters in the ClusterSet. It can be the IP address or a DNS name reachable from the `mysqlshell-runner` Pod.

The `port` is MySQL port. Defaults to `3306`.

Value type	Example
≡ subdoc	

```
    - host: ps-cluster1-mysql-primary.default.svc.cluster.local
      port: 3306
```

`spec.createReplicaClusterOptions.recoveryMethod`

Preferred method for seeding the first member of a new replica cluster when it joins the ClusterSet.

Supported values:

- `clone` — Physical snapshot via MySQL CLONE plugin. **Default.** Best for small datasets and low-latency networks.
- `incremental` — Apply GTID delta only. Use after restoring a backup from the primary into the replica. Best for large datasets or WAN links.

When you omit this field, the Operator uses `clone`.

Value type	Example
S string	`clone`

`spec.mysqlshellRunner.image`

Container image for the `mysqlshell-runner` Pod and ClusterSet Jobs. Must contain `mysqlsh` on `PATH`. The mysqlsh major version must match the MySQL endpoints (8.0 endpoints → 8.0 image, 8.4 endpoints → 8.4 image).

Value type	Example
S string	`percona/percona-server:8.4.10-10.1`

Backup Resource options for Percona Server for MySQL

The `PerconaServerMySQLBackup` Custom Resource is used to define and manage backups for a Percona Server for MySQL cluster. This CR allows you to specify the backup configuration, including the cluster to back up and the storage location.

This document describes all available options that you can use to customize your backups.

`apiVersion`

Specifies the API version of the Custom Resource. `ps.percona.com` indicates the group, and `v1alpha1` is the version of the API.

`kind`

Defines the type of resource being created: `PerconaServerMySQLBackup`.

`metadata`

The metadata part of the `deploy/backup/backup.yaml` contains metadata about the resource, such as its name and other attributes. It includes the following keys:

- `finalizers` ensure safe deletion of resources in Kubernetes under certain conditions. This subsection includes the following finalizers:
  - `percona.com/delete-backup` - deletes the backup resource after the backup data is deleted from storage.
- `name` - The name of the backup resource used to identify it in your deployment. You also use the backup name for the restore operation.

`spec`

This subsection includes the configuration of a backup resource.

`sourcePod`

Specifies the MySQL instance to take a backup from. When defined, takes precedence, regardless of the cluster type (async or group-replication) and topology. Overrides the `sourcePod` value if defined in the `deploy/cr.yaml`.

Value type	Example
S string	`ps-cluster1-mysql-0`

`clusterName`

Specifies the name of the Percona Server for MySQL cluster to back up.

Value type	Example
S string	`ps-cluster1`

`storageName`

Specifies the name of the storage where to save a backup. It must match the name you specified in the `spec.backup.storages` subsection of the `deploy/cr.yaml` file.

If the cluster or the matching storage entry defines `encryptionKeySecret`, the Operator encrypts this backup automatically. See Encrypted backups.

Value type	Example
S string	`s3-us-west`

`type`

Specifies the backup type. Supported values: `full`, `incremental`. When undefined, defaults to `full`.

Value type	Example

S string	`full`

`incrementalBaseBackupName`

Specifies the **full backup** to serve as the base for the incremental chain. This option is only valid when `type` is set to `incremental`. When undefined, the Operator uses the most recent full backup as the base for the incremental chain. Both the specified base backup and the incremental backup must be on the same storage.

Value type	Example
S string	`ps-backup`

`containerOptions.env`

The environment variables set as key-value pairs for the backup job.

Value type	Example
≡ subdoc	`- name: VERIFY_TLS` `value: "false"`

`containerOptions.args.xtrabackup`

Custom command line options for the `xtrabackup` Percona XtraBackup tool.

Value type	Example
≡ subdoc	`- "–someflag=abc"`

`containerOptions.args.xbcloud`

Custom command line options for the `xbcloud` Percona XtraBackup tool.

Value type	Example

subdoc	- “–someflag=abc”

containerOptions.args.xbstream

Custom command line options for the `xbstream` Percona XtraBackup tool

Value type	Example
subdoc	- “–someflag=abc”

Restore Resource options for Percona Server for MySQL

A Restore resource is a Kubernetes object that tells the Operator how to restore your database from a specific backup. The `deploy/backup/restore.yaml` file is a template for creating restore resources. It defines the `PerconaServerMySQLRestore` resource.

This document describes all available options that you can use to customize a restore.

`apiVersion`

Specifies the API version of the Custom Resource. `ps.percona.com` indicates the group, and `v1alpha1` is the version of the API.

`kind`

Defines the type of resource being created: `PerconaServerMySQLRestore`.

`metadata`

The metadata part of the `deploy/backup/restore.yaml` contains metadata about the resource, such as its name and other attributes. It includes the following keys:

- `name` - The name of the restore object used to identify it in your deployment. You use this name to track the restore operation status and view information about it.

`spec`

This section includes the configuration of a restore resource.

`clusterName`

Specifies the name of the Percona Server for MySQL cluster to restore.

Value type	Example

S string	`ps-cluster1`

`backupName`

Specifies the name of a backup to be used for a restore. This backup should be from the same cluster.

When the backup was encrypted, the Operator uses the `encryptionKeySecret` from the cluster Custom Resource automatically. The key used for the restore must match the key used to encrypt the backup.

Value type	Example
S string	`backup1`

`pitr` section

This subsection contains configuration options for **point-in-time recovery**. When present, the Operator restores the base backup (from `backupName` or `backupSource.destination`) and then replays binary logs until the requested **GTID** or **timestamp**. You must enable point-in-time recovery and configure a Binlog Server in the cluster; see Point-in-time recovery for details.

`pitr.type`

The type of point-in-time recovery. Supported values are:

Value	Meaning
`gtid`	Restore just before the specified GTID target (`pitr.gtid`).
`date`	Restore just before the specified timestamp (`pitr.date`).

Value type	Example
S string	`gtid`

`pitr.gtid`

The exact GTID set for point-in-time recovery, specified in the format “aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee:nnn”. Used when `pitr.type` is `gtid`.

Value type	Example
S string	`3E11FA47-71CA-11E1-9E33-C80AA9429562:1-5`

`pitr.date`

Timestamp string used when `pitr.type` is `date`. Specified in the format `yyyy-mm-dd hh:mm:ss`.

Value type	Example
S string	`2026-03-30 14:32:00`

`pitr.force`

Forces the `mysql` client to run with the `--force` flag and this silently ignores all SQL errors during binlog replay.

Warning: This might result in data loss if underlying replication or data integrity errors are ignored.

Value type	Example
boolean	`false`

The `pitr.backupSource` subsection

This subsection contains the Binlog Server settings that the Operator uses for point-in-time recovery when restoring to a cluster that does not have its own Binlog Server — for example, when restoring to a new Kubernetes environment. When specified, the Operator starts a temporary Binlog Server from these settings, uses it to locate the required binlogs, and removes it when the restore completes.

`pitr.backupSource.binlogServer.size`

Controls the number of Percona Binarylog Server Pods for binlog collection. Defaults to 1. This is the only allowed number. All greater values will be rejected. To learn more, see Point-in-time recovery.

Value type	Example

1 int	`1`

`pitr.backupSource.binlogServer.image`

The Docker image to use to deploy Percona Binarylog Server.

Value type	Example
S string	`perconalab/percona-binlog-server:0.2.1`

`pitr.backupSource.binlogServer.serverId`

The unique server ID that Percona Binarylog Server uses when connecting to MySQL as a replication client for binlog collection.

Value type	Example
1 int	`101`

`pitr.backupSource.binlogServer.storage.s3.bucket`

The name of the bucket on AWS S3 or S3-compatible storage where binlogs are streamed. You can use the same values as defined in `spec.backup.pitr.binlogServer` on the source cluster Custom Resource. For a restore to a new cluster, copy these settings from the source cluster, including the binlog `prefix` that points to the existing binlog folder.

Value type	Example
S string	`S3-BINLOG-BUCKET-NAME-HERE`

`pitr.backupSource.binlogServer.storage.s3.credentialsSecret`

The Kubernetes Secret for binlog storage. It should contain `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` keys.

Value type	Example
S string	`ps-cluster1-s3-credentials`

`pitr.backupSource.binlogServer.storage.s3.endpointUrl`

The URL to access the bucket on S3-compatible storage. Not needed for AWS S3.

Value type	Example
S string	`https://s3.amazonaws.com`

`pitr.backupSource.binlogServer.storage.s3.prefix`

Specifies the path prefix (folder) in the bucket where binlogs are stored. After configuring the Binlog Server, this prefix cannot be changed.

Important:
If you are restoring to a new cluster and both the source and target clusters use the same binlog storage, be sure to specify a different prefix for the target cluster to avoid conflicts.

Value type	Example
S string	`PREFIX_NAME`

`pitr.backupSource.binlogServer.storage.s3.region`

The region of the bucket. Required for Amazon S3 and for S3-compatible storage.

Value type	Example
S string	`us-west-2`

`pitr.backupSource.binlogServer.connectTimeout`

Timeout in seconds for establishing a connection to MySQL.

Value type	Example
1 int	`30`

`pitr.backupSource.binlogServer.readTimeout`

The maximum time in seconds the Binlog Server waits to read data from the MySQL instance.

Value type	Example
1 int	`30`

`pitr.backupSource.binlogServer.writeTimeout`

The maximum time in seconds the Binlog Server waits to write data to a remote server.

Value type	Example
1 int	`30`

`pitr.backupSource.binlogServer.idleTime`

The maximum time in seconds the Binlog Server stays in idle mode before trying to reconnect.

Value type	Example
1 int	`30`

`pitr.backupSource.binlogServer.logLevel`

The verbosity level for Binlog Server logs. Supported values are: `info` (default), `warning`, `error`, `debug`.

Value type	Example
S string	`info`

The `backupSource` subsection

Contains the configuration options to restore from a backup made in a different cluster, namespace, or Kubernetes environment.

`backupSource.destination`

Specifies the path to the backup on the storage

Value type	Example
S string	`s3://bucket-name/backup-destination/`

`backupSource.storage.encryptionKeySecret`

References a Kubernetes Secret that contains the key used to decrypt an encrypted backup. Required when restoring an encrypted backup from object storage on a cluster that does not have access to the source cluster's encryption key Secret. The key used for the restore must match the key used to encrypt the backup.

Value type	Example
subdoc	`name: my-encryption-key` `key: encryptionKey`

`backupSource.storage.encryptionKeySecret.name`

The name of the Kubernetes Secret that stores the backup encryption key.

Value type	Example
S string	`my-encryption-key`

`backupSource.storage.encryptionKeySecret.key`

The key within the Secret that holds the encryption key value. Defaults to `encryptionKey`.

Value type	Example
S string	`encryptionKey`

`backupSource.storage.s3.bucket`

Specifies the name of the bucket where the backup that you wish to restore from is saved.

Value type	Example
S string	

`backupSource.storage.s3.credentialsSecret`

Specifies the Secrets object name with the credentials to access the storage with a backup.

Value type	Example
S string	`ps-cluster1-s3-credentials`

`backupSource.storage.s3.region`

The AWS region to use. Please note **this option is mandatory** for Amazon and all S3-compatible storages.

Value type	Example
S string	`us-west-2`

`backupSource.storage.type`

Specifies the type of the backup storage. Available options: `s3`, `azure`.

Value type	Example
S string	`s3`

`containerOptions.env`

The environment variables set as key-value pairs for the restore job.

Value type	Example
subdoc	`- name: VERIFY_TLS` `value: “false”`

containerOptions.args.xtrabackup

Custom command line options for the `xtrabackup` Percona XtraBackup tool.

Value type	Example
subdoc	`- “–someflag=abc”`

containerOptions.args.xbcloud

Custom command line options for the `xbcloud` Percona XtraBackup tool.

Value type	Example
subdoc	`- “–someflag=abc”`

containerOptions.args.xbstream

Custom command line options for the `xbstream` Percona XtraBackup tool

Value type	Example
subdoc	`- “–someflag=abc”`

Custom resource statuses

Status fields show the current state of a Custom Resource. The Operator sets these fields in the `.status` section of a Custom Resource. You do not edit the status.

Use status values to confirm progress, detect failures, and decide when it is safe to run the next action (for example, start a restore after a backup succeeds, or perform switchover after the ClusterSet is `Ready`).

How to view custom resource statuses

To check the status of custom resources, use the `kubectl get` or `kubectl describe` commands. See how to use them to get the quick overview, in-depth details, and targeted queries.

Get a quick overview

List your resources and check their high-level status:

```
kubectl get ps <cluster-name> -n <namespace>
kubectl get ps-clusterset <clusterset-name> -n <namespace>
kubectl get ps-backup <backup-name> -n <namespace>
kubectl get ps-restore <restore-name> -n <namespace>
```

Sample output for `PerconaServerMySQL`

```
NAME          REPLICATION          ENDPOINT                              STATE   MYSQL
HAPROXY   AGE
ps-cluster1   group-replication    ps-cluster1-haproxy.default:3306      ready   3
1         27m
```

View full details

See all status details, conditions, and events:

```
kubectl get ps <cluster-name> -n <namespace> -o yaml
kubectl describe ps <cluster-name> -n <namespace>

kubectl get ps-clusterset <clusterset-name> -n <namespace> -o yaml
kubectl describe ps-clusterset <clusterset-name> -n <namespace>

kubectl get ps-backup <backup-name> -n <namespace> -o yaml
kubectl get ps-restore <restore-name> -n <namespace> -o yaml
```

Check for the `.status` field in the output to find the current state, readiness, messages, and conditions.

Query a status field directly

You can extract specific status fields using `jsonpath`.

Example 1. Get the InnoDB cluster name (for ClusterSet configuration):

```
kubectl get ps <cluster-name> -n <namespace> \
  -o jsonpath='{.status.innodbClusterName}{"\n"}'
```

Sample output

```
pscluster1
```

Example 2. Check whether the ClusterSet is ready:

```
kubectl get ps-clusterset <clusterset-name> -n <namespace> \
  -o jsonpath='{range .status.conditions[?(@.type=="Ready")]}{.status}{": "}
{.reason}{"\n"}{end}'
```

Sample output

```
True: ClusterSetHealthy
```

Example 3. Get the backup destination path:

```
kubectl get ps-backup <backup-name> -n <namespace> \
  -o jsonpath='{.status.destination}{"\n"}'
```

Sample output

```
s3://<my-bucket>/ps-cluster1-2026-06-29-08:06:10-full
```

PerconaServerMySQL status

The main cluster state is recorded in the `status.state` field. Component-level states are recorded in the `status.mysql`, `status.haproxy`, `status.router`, `status.orchestrator`, and `status.binlogserver` sections.

Common fields:

- `status.state` — overall cluster state
- `status.host` — connection endpoint
- `status.innodbClusterName` — InnoDB cluster name reported by MySQL Shell
- `status.mysql.ready` / `status.mysql.size` — number of ready MySQL Pods and the desired size of the database cluster
- `status.conditions` — detailed condition list with reason and message

Cluster state values

`status.state` values are:

Value	Meaning
`""`	The Operator has not set a state yet.
`initializing`	The Operator is creating or reconciling the cluster.
`stopping`	The Operator is stopping or scaling down cluster components.
`paused`	The cluster is paused.
`ready`	The cluster is up and healthy.

`error`	The Operator detected an error; check conditions and events.

The same state values apply to each component subsection (`status.mysql.state`, `status.haproxy.state`, etc.).

Conditions

Conditions show more detail about cluster state changes. You can see them in `status.conditions[]`.

Common condition fields:

- `type` — condition type
- `status` — condition status (`True`, `False`, or `Unknown`)
- `reason` — short reason string
- `message` — human-readable details

`status.conditions[].type` values:

Value	Meaning
`Initializing`	The cluster or a component is starting up.
`Ready`	The cluster or a component is ready.
`Error`	The Operator detected an error (for example, full cluster crash).
`InnoDBClusterBootstrapped`	The InnoDB Cluster metadata exists and Group Replication is formed.
`AwaitingExternalBootstrap`	The cluster is configured with `spec.mysql.bootstrap.mode: manual` and waits for an external actor (typically the ClusterSet controller) to bootstrap Group Replication.
`ClusterSetReplicationRunning`	The cluster is a REPLICA member of an InnoDB ClusterSet and async replication from the primary cluster is active.

`status.conditions[].status` values:

Value	Meaning
`True`	The condition is currently true.
`False`	The condition is currently false.
`Unknown`	The Operator has not determined the condition yet.

The Operator sets `reason` and `message` as free-form strings. Common reasons include `ManualBootstrapRequested`, `ClusterSetReplicationRunning`, `FullClusterCrashDetected`, and state names such as `Initializing` and `Ready`.

PerconaServerMySQLClusterSet status

ClusterSet progress and topology are reflected in `status.primaryCluster`, `status.clusters`, and `status.conditions`.

Common fields are:

- `status.primaryCluster` — the cluster currently serving as the ClusterSet primary (observed; may lag `spec.primaryCluster` briefly during switchover)
- `status.primaryClusterEndpoint` — host:port of the global primary instance
- `status.clusters` — map of member clusters keyed by `innodbClusterName`, mirroring MySQL Shell `.status()` output
- `status.conditions` — ClusterSet-level conditions
- `status.lastObservedAt` — timestamp of the last successful status refresh
- `status.lastObservedGeneration` — generation last reconciled

Per-member cluster status

Each entry in `status.clusters[<innodbClusterName>]` contains:

Field	Meaning
`clusterRole`	`PRIMARY` or `REPLICA` as observed by MySQL Shell

`globalStatus`	Member health in the ClusterSet
`primary`	Host:port of the cluster's local group replication primary

`globalStatus` values (from MySQL Shell):

Value	Meaning
`OK`	Cluster is healthy and replicating as expected.
`OK_NOT_REPLICATING`	Cluster is reachable but not currently replicating (transitional or misconfigured).
`NOT_OK`	Cluster has a problem; check MySQL Shell status and events.
`INVALIDATED`	Cluster was fenced off after forced failover; may have divergent GTIDs. Requires running `rejoinCluster()` manually if GTIDs are compatible. Otherwise, the INVALIDATED cluster must be removed and recreated to rejoin the ClusterSet.
`UNKNOWN`	Status could not be determined.

Conditions

`status.conditions[].type` values:

Value	Meaning
`Ready`	ClusterSet is formed and replicas are replicating from the specified primary.
`ClusterSetBootstrapped`	The primary cluster is configured as a ClusterSet.
`MySQLShellRunnerReady`	The `mysqlshell-runner` Pod is running and ready.
`SwitchoverInProgress`	`spec.primaryCluster` differs from the observed primary and a switchover Job is running or pending.
`ErrorReconcile	An error occurred during reconciliation

`ReplicaManagementFailure`	A replica add/remove Job failed; see the Job logs.
`ClusterSetDissolving`	The Custom Resource is being deleted and the dissolve finalizer is running.

Common `Ready` condition reasons:

Reason	Meaning
`ClusterSetHealthy`	MySQL Shell reports overall ClusterSet status as healthy.
`ClusterSetNotHealthy`	One or more members are not healthy; see `status.clusters` and condition message.
`PrimaryUnreachable`	Primary cluster is not reachable
`AccessDenied`	Incorrect password configured on the replica site

Events

The ClusterSet controller emits Kubernetes events you can view with `kubectl describe ps-clusterset`:

Event reason	Meaning
`ClusterSetBootstrapped`	ClusterSet was created on the primary cluster.
`ClusterSetPrimarySwitched`	Planned switchover completed.
`ClusterSetPrimaryForcedSwitched`	Forced failover completed.
`ClusterSetMemberAdded`	A cluster was added to the ClusterSet.
`ClusterSetMemberRemoved`	A cluster was removed from the ClusterSet.
`ClusterSetHealthDegraded`	Overall ClusterSet health dropped from healthy to unhealthy.

PerconaServerMySQLBackup status

Backup progress and results are in `status.state`. You also get destination and timing details that help you validate backups and incremental chains.

Common fields:

- `status.state` — backup job state
- `status.type` — backup type (`full` or `incremental`)
- `status.destination` — backup path or URL
- `status.completed` — completion timestamp
- `status.stateDescription` — error details when the backup fails
- `status.backupSource` — source pod or path used for the backup
- `status.conditions` — additional conditions

Backup state values

`status.state` values are:

Value	Meaning
`""`	Backup is created but not processed yet.
`Starting`	Backup is starting.
`Running`	Backup is in progress.
`Succeeded`	Backup completed successfully.
`Error`	Backup failed to start.
`Failed`	Backup started but failed during execution.

Conditions

Value	Meaning

`BackupLeaseAcquired`	The backup job holds the cluster backup lease and is authorized to run.

PerconaServerMySQLRestore status

Restore progress and results are in `status.state`. Use these fields to confirm when a restore starts, finishes, or fails.

Common fields:

- `status.state` — restore job state
- `status.stateDescription` — error details when the restore fails
- `status.completed` — completion timestamp
- `status.backupSource` — backup metadata used for the restore (when restoring from an existing backup object)

Restore state values

`status.state` values are:

Value	Meaning
`""`	Restore is created but not processed yet.
`Starting`	Restore is starting.
`Running`	Restore is in progress.
`Succeeded`	Restore completed successfully.
`Error`	Restore failed to start.
`Failed`	Restore started but failed during execution.

Certified images

Percona certified images

This page lists Percona's certified Docker images that you can use with Percona Operator for MySQL based on Percona Server for MySQL 1.2.0.

To find images for a specific Operator version, see Retrieve Percona certified images

Images released with the Operator version 1.2.0:

Image	Digest
percona/percona-server-mysql-operator:1.2.0	76deed6a6daca2846fba6046b6646e25d22bca2d6d58a0b567e0a2c1240fe103
percona/percona-server-mysql-operator:1.2.0 (ARM64)	e0dd1e6bf1fd90b2149290500997ee791101f4500b5162bcea5694b1b9d7ab58
percona/percona-server:8.4.10-10.1	d24391a363426239220c35b9707d6b26ce7522ac27abe55689baebfb48bd9fb3
percona/percona-server:8.4.10-10.1 (ARM64)	70f6c4d01b5807737cdd423ab32af1feb1d00513b5420a84361773b167aeca87
percona/percona-server:8.0.46-37.1	1552a3b1860bb12325755295fb833890eb920c34fc212dbadb34c59baf0c378a
percona/percona-server:8.0.46-37.1 (ARM64)	e41dfb981de432da1a117077e7230008422ab49d6c0e9dd8d17a03d7ffde5dc0
percona/percona-xtrabackup:8.4.0-6.1	d135aadaae9e2f947cb2002f982f7b4c6e177b1c7e3d543ef7795aea999feedd
percona/percona-xtrabackup:8.4.0-6.1 (ARM64)	fcf2b3fc20cfbfa6d47ec60cecd881f915beacdae838f994dda984baea825293
percona/percona-xtrabackup:8.0.35-36.1	5416f19ac087e4d8e06eeff63cf1e075dc3b4e45577abbb9c3d87ae1ea3816c5

percona/percona-xtrabackup:8.0.35-36.1 (ARM64)	b604421b30bdd7ddfbc716220a8f8f32077c12a762d1eae622dcf4e5085b115a
percona/percona-mysql-router:8.4.10-10.1	7edd16793022a518842aa6e709582af9175ee9ab145faf11e930f8175f16c588
percona/percona-mysql-router:8.4.10-10.1 (ARM64)	e3be4405858a4198ecb1265365763de33dc6991bdb811bbc10803a92d06606a0
percona/percona-mysql-router:8.0.46-37.1	826cd7a26e918fc5b75cad6ba19d74b508dd1a73f93e49cfac727f6f40b74a02
percona/percona-mysql-router:8.0.46-37.1 (ARM64)	d7a835fff71c6fa881d35cc54fb126c4027ccd9692575c6b95c2cd6329915b90
percona/percona-orchestrator:3.2.6-22	384bcf3121f50e9536fc911532386b9b485a52701e1e99ada51a5faf91d667c3
percona/percona-orchestrator:3.2.6-22 (ARM64)	7ed29d0cde9396687fa42e1640ac2554125fd3a1e2a9f8d70667bbd00b386703
percona/haproxy:2.8.18-1	09e4d2ce9e65dc4aec9195e818e6da2041aea1a2bdb04f868d8c42ee81090dbf
percona/haproxy:2.8.18-1 (ARM64)	563d84d64e1668cb4f6b2202fbe15a18e0733b48fe963383bb877c9e3a8a0abd
percona/percona-toolkit:3.7.1-4	0865492cb600f1092618de63e91ae480606a0459d86e5eb876c7e7fb6e6cdda1
percona/percona-toolkit:3.7.1-4 (ARM64)	c0ebb11297c3bd83bbbc7acd9195dfa754e7e9be9da70800d146381a7d652129
percona/pmm-client:3.8.1	a92cfb7f912bd85d8245575c3ee5c423664ad2baedb674d159a87b113dbd4de2
percona/pmm-client:3.8.1 (ARM64)	3fe427c0666337df7613824da5f3b5fb7397e849f70402ac557c1324c5d996e6
percona/percona-server-mysql-operator:1.2.0-	53336b6ccbe463c255f7ca9caaf9390ceba2b9c7672b91998cbe5f44d84dc4e4

binlog-server-0.3.1	
percona/percona-server-mysql-operator:1.2.0-binlog-server-0.3.1 (ARM64)	361695d2b77443e67aa7f2858e9345cff08967afa3e2a8820d4c5981963fa858

Find images for previous versions

Retrieve Percona certified images

When preparing for the upgrade, you must have the list of compatible images for a specific Operator version and the database version you wish to update to. You can either manually find the images in the list of certified images or you can get this list by querying the **Version Service** server.

What is the Version Service?

The **Version Service** is a centralized repository that the Percona Operator for MySQL connects to at scheduled times to get the latest information on compatible versions and valid image paths. This service is a crucial part of the automatic upgrade process, and it is enabled by default. Its landing page, `check.percona.com`, provides more details about the service itself.

How to query the Version Service

You can manually query the Version Service using the `curl` command. The basic syntax is:

```
curl https://check.percona.com/versions/v1/ps-operator/<operator-version>/<ps-version> | jq -r '.versions[].matrix'
```

where:

- `<operator-version>` is the version of the Percona Operator for MySQL you are using.
- `<ps-version>` is the version of Percona Server for MySQL you want to get images for. This part is optional and helps filter the results. It can be a specific Percona Server for MySQL version (e.g. 8.4), a recommended version (e.g. 8.4-recommended), or the latest available version (e.g. 8.4-latest).

For example, to retrieve the list of images for the Operator version `0.11.0` and the latest version of Percona Server for MySQL 8.4, use the following command:

```
curl https://check.percona.com/versions/v1/ps-operator/0.11.0/8.4-latest | jq -r '.versions[].matrix'
```

Sample output

```
"pmm": {
    "3.3.1": {
      "imagePath": "percona/pmm-client:3.3.1",
```

```
      "imageHash": "29a9bb1c69fef8bedc4d4a9ed0ae8224a8623fd3eb8676ef40b13fd044188cb4",
      "imageHashArm64":
"50bccba4cb2c33bd3f6c5e37553bb70345de3f328b23a64ecfa63893f9025c83",
      "status": "recommended",
      "critical": true
    }
  },
  "haproxy": {
    "2.8.15": {
      "imagePath": "percona/haproxy:2.8.15",
      "imageHash": "49e6987a1c8b27e9111ae1f1168dd51f2840eb6d939ffc157358f0f259819006",
      "imageHashArm64": "",
      "status": "recommended",
      "critical": false
    }
  },
  "backup": {
    "8.4.0-3": {
      "imagePath": "percona/percona-xtrabackup:8.4.0-3",
      "imageHash": "01071522753ad94e11a897859bba4713316d08e493e23555c0094d68da223730",
      "imageHashArm64": "",
      "status": "available",
      "critical": false
    }
  },
  "operator": {
    "0.11.0": {
      "imagePath": "percona/percona-server-mysql-operator:0.11.0",
      "imageHash": "3068b1a4d81b5da7676e071040d3b44ff9fec5532cbfabb17f9c7612e8c9d35d",
      "imageHashArm64":
"b50f215869ca3d9f6a52561171851e8ffa033ca30d0276556e220ad448418b61",
      "status": "recommended",
      "critical": false
    }
  },
  "mysql": {
    "8.4.5-5": {
      "imagePath": "percona/percona-server:8.4.5-5",
      "imageHash": "9628b1e598c0ec13c2a6b834caa1642bf77f2b4a2d7620b1ba2d8aaf2b708133",
      "imageHashArm64": "",
      "status": "available",
      "critical": false
    }
  },
  "router": {
    "8.4.5": {
      "imagePath": "percona/percona-mysql-router:8.4.5",
      "imageHash": "e890ecc49c297cc8882b54ba457ff4d25da896cb11269989fa072aa338d620c1",
      "imageHashArm64": "",
      "status": "available",
      "critical": false
    }
  },
  "orchestrator": {
    "3.2.6-17": {
      "imagePath": "percona/percona-orchestrator:3.2.6-17",
      "imageHash": "c1871ddc6ff3eaca7bb03c3aa11db880ae02d623db1203d0858f8566f56ea5f7",
      "imageHashArm64": "",
```

```
      "status": "recommended",
      "critical": false
    }
  },
  "toolkit": {
    "3.7.0": {
      "imagePath": "percona/percona-toolkit:3.7.0",
      "imageHash": "17ef2b69a97fa546d1f925c74ca09587ac215085c392761bb4d51f188baa6c0e",
      "imageHashArm64": "",
      "status": "recommended",
      "critical": false
    }
  }
```


To narrow down the search and check the Percona Server for MySQL images available for a specific Operator version (`0.11.0` in the following example), use the following command:

```
curl -s https://check.percona.com/versions/v1/ps-operator/0.11.0 | jq -r
'.versions[0].matrix.mysql | to_entries[] | "\(.key)\t\(.value.imagePath)\t\
(.value.status)"'
```

Sample output

```
8.0.32-24   percona/percona-server:8.0.32-24    available
8.0.33-25   percona/percona-server:8.0.33-25    available
8.0.36-28   percona/percona-server:8.0.36-28    available
8.0.40-31   percona/percona-server:8.0.40-31    available
8.0.42-33   percona/percona-server:8.0.42-33    recommended
8.4.5-5     percona/percona-server:8.4.5-5      available
```

Versions compatibility

Below you can find the versions of software components and platforms that have been tested and are confirmed to work with each specific Operator version. Other software and platform versions might work, but we have not tested them and cannot guarantee their compatibility.

Cluster components

Operator	Percona Server for MySQL	Percona XtraBackup	HA Proxy	MySQL Router	Orchestrator	Percona Toolkit
1.2.0	8.4.10-10.1, 8.0.46-37.1	8.4.0-6.1, 8.0.35-36.1	2.8.18-1	8.4.10, 8.0.46	3.2.6-22	3.7.1-4
1.1.0	8.4.8-8.1, 8.0.45-36.1	8.4.0-5.1, 8.0.35-35.1	2.8.18-1	8.4.8, 8.0.45	3.2.6-20	3.7.1
1.0.0	8.4.6-6.1, 8.0.43-34.1	8.4.0-4.1, 8.0.35-34.1	2.8.15	8.4.6-6.1, 8.0.43-34.1	3.2.6-18	3.7.0-2

Platforms

Operator	GKE	EKS	Openshift	Minikube
1.2.0	1.33 - 1.35	1.33 - 1.36	4.18 - 4.22	1.38.1
1.1.0	1.32 - 1.35	1.33 - 1.35	4.18 - 4.21	1.38.1
1.0.0	1.31 - 1.33	1.31 - 1.34	4.16 - 4.20	1.37.0

Legal

Copyright and licensing information

Documentation licensing

Percona Operator for MySQL based on Percona Server for MySQL documentation is (C)2009-2023 Percona LLC and/or its affiliates and is distributed under the Creative Commons Attribution 4.0 International License.

Trademark policy

This Trademark Policy is to ensure that users of Percona-branded products or services know that what they receive has really been developed, approved, tested and maintained by Percona. Trademarks help to prevent confusion in the marketplace, by distinguishing one company's or person's products and services from another's.

Percona owns a number of marks, including but not limited to Percona, XtraDB, Percona XtraDB, XtraBackup, Percona XtraBackup, Percona Server, and Percona Live, plus the distinctive visual icons and logos associated with these marks. Both the unregistered and registered marks of Percona are protected.

Use of any Percona trademark in the name, URL, or other identifying characteristic of any product, service, website, or other use is not permitted without Percona's written permission with the following three limited exceptions.

First, you may use the appropriate Percona mark when making a nominative fair use reference to a bona fide Percona product.

Second, when Percona has released a product under a version of the GNU General Public License ("GPL"), you may use the appropriate Percona mark when distributing a verbatim copy of that product in accordance with the terms and conditions of the GPL.

Third, you may use the appropriate Percona mark to refer to a distribution of GPL-released Percona software that has been modified with minor changes for the sole purpose of allowing the software to operate on an operating system or hardware platform for which Percona has not yet released the software, provided that those third party changes do not affect the behavior, functionality, features, design or performance of the software. Users who acquire this Percona-branded software receive substantially exact implementations of the Percona software.

Percona reserves the right to revoke this authorization at any time in its sole discretion. For example, if Percona believes that your modification is beyond the scope of the limited license granted in this Policy or that your use of the Percona mark is detrimental to Percona, Percona will revoke this authorization. Upon revocation, you must immediately cease using the applicable Percona mark. If you do not immediately cease using the Percona mark upon revocation, Percona may take action to protect its rights and interests in the Percona mark. Percona does not grant any license to use any Percona mark for any other modified versions of Percona software; such use will require our prior written permission.

Neither trademark law nor any of the exceptions set forth in this Trademark Policy permit you to truncate, modify or otherwise use any Percona mark as part of your own brand. For example, if XYZ creates a modified version of the Percona Server, XYZ may not brand that modification as “XYZ Percona Server” or “Percona XYZ Server”, even if that modification otherwise complies with the third exception noted above.

In all cases, you must comply with applicable law, the underlying license, and this Trademark Policy, as amended from time to time. For instance, any mention of Percona trademarks should include the full trademarked name, with proper spelling and capitalization, along with attribution of ownership to Percona Inc. For example, the full proper name for XtraBackup is Percona XtraBackup. However, it is acceptable to omit the word “Percona” for brevity on the second and subsequent uses, where such omission does not cause confusion.

In the event of doubt as to any of the conditions or exceptions outlined in this Trademark Policy, please contact trademarks@percona.com for assistance and we will do our very best to be helpful.

Release Notes

Percona Operator for MySQL Release Notes

- Percona Operator for MySQL 1.2.0 (2026-07-03)
- Percona Operator for MySQL 1.1.0 (2026-04-17)
- Percona Operator for MySQL 1.0.0 (2025-11-17)
- Percona Operator for MySQL 0.12.0 (2025-09-23)
- Percona Operator for MySQL 0.11.0 (2025-09-01)
- Percona Operator for MySQL 0.10.0 (2025-06-04)
- Percona Operator for MySQL 0.9.0 (2025-02-11)
- Percona Operator for MySQL 0.8.0 (2024-07-16)
- Percona Operator for MySQL 0.7.0 (2024-03-25)
- Percona Operator for MySQL 0.6.0 (2023-09-05)
- Percona Operator for MySQL 0.5.0 (2023-03-30)
- Percona Operator for MySQL 0.4.0 (2023-01-30)
- Percona Operator for MySQL 0.3.0 (2022-09-29)
- Percona Operator for MySQL 0.2.0 (2022-06-30)
- Percona Distribution for MSQL Operator based on Percona Server for MySQL* 0.1.0 (2022-01-25)

Percona Operator for MySQL 1.2.0 (2026-07-03)

Installation

Percona Operator for MySQL brings production-grade automation to MySQL deployments on Kubernetes. It handles provisioning, scaling, backups, failover, and upgrades using declarative Custom Resources, thus reducing manual effort and human error. With built-in support for Percona XtraBackup, proxies such as HAProxy or MySQL Router, and Percona Toolkit, it ensures resilient, secure, and performant MySQL clusters.

Release highlights

This release delivers key features such as cross-site replication, encrypted backups, and automated storage resizing, which harden the Operator for production use. Additionally, it improves security with Orchestrator API authentication and Vault encryption secret validation, and includes a bunch of improvements for day-to-day operations.

Cross-site replication for Group Replication clusters

Until now, each Percona Server for MySQL cluster stood on its own. That worked well for high availability within a cluster where group replication keeps cluster nodes in sync. But to scale and span MySQL geographically for disaster recovery, migration or read distribution, you had to build and maintain custom tooling outside the Operator to interconnect sites and replicate data between them.

Cross-site replication closes that gap by linking Group Replication clusters into a MySQL InnoDB ClusterSet. You manage the topology declaratively from Kubernetes with a new `PerconaServerMySQLClusterSet` Custom Resource where you declare which clusters participate, which one is the primary, and how replicas receive data when they join the ClusterSet.

The Operator automates the ClusterSet provisioning and lifecycle management in a similar way it does for a standalone Percona Server for MySQL cluster. It bootstraps the ClusterSet, adds or removes replica clusters, refreshes status, and runs switchover when you change the primary. Because management runs over the MySQL protocol, replica clusters can run in the same Kubernetes cluster, a different cluster, or outside Kubernetes entirely.

With cross-site replication, you gain:

- Faster recovery — promote a live replica in a second region during an outage instead of restoring from backup alone.
- Regional read scaling — serve read traffic locally while writes stay on the primary.
- Unified operations — one Custom Resource and the same declarative model you already use for a single cluster.

Learn more about the workflow, setup and usage in the cross-cluster replication documentation.

Encrypted backups

You can now encrypt your backup data with Percona Operator for MySQL, so backup files stored in S3, GCS, or Azure are protected at rest. This helps you meet compliance requirements without adding external encryption tools or changing how you run scheduled and on-demand backups.

Encrypted backups are supported for both Group Replication and asynchronous replication cluster types.

The Operator uses Percona XtraBackup's native encryption and automates the full workflow for you. You create a Kubernetes Secret with the encryption key and reference it in the cluster Custom Resource either on the global level or per storage. The storage-level setting takes precedence. The Operator uses that Secret to encrypt the backup stream during backup and to decrypt it automatically during in-place restore on the same cluster.

For a restore on a new cluster or in a different Kubernetes environment, provide the same encryption key in the restore resource, because the Operator has no access to Secrets from the source cluster. **The key used for the restore must match the key used to encrypt the backup**. If the key is lost, the backup cannot be recovered.

The default encryption algorithm is **AES256**. If your security policy requires a different cipher or you want to tune performance, you can override the algorithm and related settings through Percona XtraBackup options in `containerOptions.args.xtrabackup`.

To learn more about encrypted backups, read our documentation.

Automatic storage resizing

Starting with version 1.2.0, the Operator can automatically resize Persistent Volume Claims (PVCs) for Percona Server for MySQL Pods based on your configured thresholds. The Operator monitors storage usage and when it exceeds the defined threshold, triggers resizing until it reaches the maximum storage size.

This improvement gives you:

- Fewer outages from full disks because storage grows with demand
- Less guesswork in capacity planning and fewer last-minute fixes
- Lower operational effort for developers and platform engineers
- Cost control by expanding only when needed
- A more predictable environment so teams can focus on delivery

To enable automatic storage resizing, edit the Custom Resource manifest as follows:

```
spec:
  storageScaling:
    enableVolumeScaling: true
    autoscaling:
      enabled: true
      triggerThresholdPercent: 80
      growthStep: 2Gi
      maxSize: "10Gi"
```

Learn more about the workflow and troubleshooting tips in our documentation.

Delegate PVC resizing to an external autoscaler

You can now configure the Operator to use an external storage autoscaler instead of its own resizing logic. This ability may be useful for organizations needing centralized, advanced, or cross-application scaling policies.

To use an external autoscaler, set the `spec.storageScaling.enableExternalAutoscaling` option to `true` in the Custom Resource manifest.

```
spec:
  storageScaling:
    enableExternalAutoscaling: true
```

Connection details exposed in the user Secret

Connecting applications to your cluster no longer means hunting down endpoints and credentials across Services and Secrets. With this release, the Operator creates and maintains a dedicated Secret for a `root` user that gives your application developers and administrators a single, reliable source of connection information. The Secret name follows the format `<cluster-name>-psuser-root` (`ps-cluster1-psuser-root` by default).

The Secret is owned by the Operator and updated on every reconciliation, so you always get current values without manual upkeep. The `host`, `port`, and `uri` fields point to the direct MySQL primary Service and are automatically updated when failover moves the primary to another Pod.

If you have enabled HAProxy or MySQL Router, the Secret also includes `proxy-host`, `proxy-port`, and `proxy-uri` values, so applications can connect through your proxy without extra configuration. These fields are generic — they do not reveal which proxy technology is in use.

When you expose the proxy with a `LoadBalancer` Service type and an external IP is assigned, the Secret adds `proxy-external-host`, `proxy-external-port`, and `proxy-external-uri` for connections from outside the cluster.

You can mount this Secret in your application Pods, reference it in your CI/CD pipeline, or use it with External Secrets. However you consume credentials today, you now have everything in one place.

Configure binlog settings in the restore object for point-in-time recovery

You can now specify binlog storage directly in the `PerconaServerMySQLRestore` object when performing point-in-time recovery to a new cluster. Instead of configuring Binlog Server on the target cluster just to replay binary logs from the source environment, you define the binlog location alongside your existing `backupSource` settings. This simplifies the cross-cluster point-in-time recovery configuration as a single restore manifest carries everything needed for the operation.

When you trigger the restore, the Operator starts a temporary Binlog Server using those settings, uses it to locate and fetch the required binlogs from the object storage, and removes it when the restore completes.

This improvement makes the restore path straightforward. By separating binlog collection on the source cluster from binlog access during restore on the target, the Operator makes disaster recovery and migration scenarios much easier to plan and execute.

Point-in-time recovery remains a tech preview feature. We do not recommend using it in production environments yet, but we encourage you to try this workflow in staging or test clusters and share your feedback.

Read more about point-in-time recovery and restore to a new cluster in our documentation.

Disable NodePort allocation for LoadBalancer Services

By default, Kubernetes assigns a `NodePort` behind every `LoadBalancer` Service, even when the underlying load balancer does not require it.

With this release, you can now turn off `NodePort` allocation for the `LoadBalancer` Services created by the Operator. Update your cluster Custom Resource with the following configuration:

```
spec:
  expose:
    type: LoadBalancer
    allocateLoadBalancerNodePorts: false
```

This setting keeps the `LoadBalancer` Service behavior intact while preventing Kubernetes from reserving `NodePorts`. It's useful when:

- Your load balancer implementation doesn't rely on NodePorts
- You want to reduce the attack surface by avoiding unintended node level exposure
- You run many LoadBalancer services and want to avoid port exhaustion

Independent resource configuration for HAProxy sidecar containers

HAProxy Pods include the `mysql-monit` sidecar, a lightweight DNS-polling container that monitors MySQL endpoints. Previously, it inherited resource requests and limits from the main HAProxy container. Since the sidecar needs far fewer resources than HAProxy itself, it was over-provisioned.

Now you can configure the resource requests and limits for each HAProxy sidecar container independently. Use the `spec.proxy.haproxy.sidecarResources` subsection in the Custom Resource and specify the required values for each sidecar:

```
proxy:
  haproxy:
    sidecarResources:
      mysql-monit:
        requests:
          cpu: 50m
          memory: 64Mi
        limits:
          cpu: 100m
          memory: 128Mi
```

This improvement helps you use cluster resources more efficiently and thereby lower your infrastructure costs.

Vault encryption secret validation

The Operator now validates the Vault Secret when you enable data-at-rest encryption with HashiCorp Vault. If the Secret is missing or invalid, you see a clear error in the cluster status right away, so you can fix the configuration without digging through Pod logs.

With this validation, you can now deploy Vault and the database cluster together, such as using Helm integration charts. You can define the Secret reference up front and let the Operator wait until the Secret is present and valid before continuing cluster setup.

Read more about configuring data-at-rest encryption with Vault.

Improve operational efficiency with the support for concurrent cluster reconciliation

Reconciliation is a Kubernetes mechanism to keep your cluster in sync with its desired state. Previously, the Operator ran only one reconciliation loop at a time. This sequential processing meant that other clusters managed by the same Operator had to wait for the current reconciliation to complete before receiving updates.

With this release, the Operator supports concurrent reconciling and can process several clusters simultaneously. You can define the maximum number of concurrent reconciles with the `MAX_CONCURRENT_RECONCILES` environment variable for the Operator deployment.

This enhancement significantly improves scalability and responsiveness, especially in multi-cluster environments.

Configurable DNS suffix for Operator connections

You can now specify a custom DNS suffix that the Operator uses when it generates service names. This is useful when the Operator runs in a vcluster or in a cluster with a custom DNS configuration. In those setups, the default `cluster.local` suffix can cause incorrect domain name resolution and, as a result, failed connections to external services such as PMM. By setting the `clusterServiceDNSSuffix` in the Custom Resource to your cluster's DNS suffix, the Operator generates hostnames that match your DNS configuration. This ensures the service discoverability and correct communication between workloads.

Secure Orchestrator API access with authentication

The Orchestrator HTTP API is the control plane for your MySQL clusters with asynchronous replication. It handles failovers, topology changes, and other critical operations.

Starting with this release, the Orchestrator API requires authentication. Clients must present valid user credentials to connect, and unauthenticated requests are rejected.

The Operator handles authentication automatically for its own API calls, reading credentials from the mounted secret, so day-to-day cluster management continues to work as before. For custom integrations or scripts that call the Orchestrator API directly, you will need to include the appropriate credentials.

This change hardens your deployment by ensuring only authorized callers can initiate failover or topology operations. As a result, you have a stronger protection against accidental or malicious disruption of your MySQL cluster.

Fine-tune Orchestrator failover behavior

Until now, you could not adjust the Orchestrator behavior beyond the Operator defaults. This release makes the `spec.orchestrator.configuration` field effective, so you can tune failover settings per cluster directly in the Custom Resource.

Provide Orchestrator options as a JSON string. For example, with the `FailMasterPromotionOnLagMinutes` option, you can control how long Orchestrator waits before promoting a new primary when replication is lagging. Using the `RecoveryPeriodBlockSeconds` option, define how long to block recovery after a failover.

```
spec:
  orchestrator:
    enabled: true
    configuration: '{"FailMasterPromotionOnLagMinutes": 30,
"RecoveryPeriodBlockSeconds": 300}'
```

The Operator merges your settings into the Orchestrator configuration. When you change the `spec.orchestrator.configuration` field, Orchestrator Pods restart in a rolling fashion to apply the new settings.

Options that the Operator manages itself such as Raft topology, topology TLS, HTTP authentication, failover hooks, alias detection queries, and storage paths are reserved and ignored if you set them, so cluster automation continues to work reliably.

Improved monitoring for clusters in multi-region or multi-namespace deployments in PMM

Now you can define a custom name for your clusters deployed in different data centers. This name helps Percona Management and Monitoring (PMM) Server to correctly recognize clusters as connected and monitor them as one deployment. Similarly, PMM Server identifies clusters deployed with the same names in different namespaces as separate ones and correctly displays performance metrics for you on dashboards.

To assign a custom name, define this configuration in the Custom Resource manifest for your cluster:

```
spec:
  pmm:
    customClusterName: testClusterName
```

Percona Server for MySQL clusters 8.4 have NUMA disabled by default

`innodb_numa_interleave` is a MySQL configuration variable that spreads memory allocation across multiple NUMA (Non Uniform Memory Access) nodes. MySQL 8.4 turns this setting on by default to improve performance on servers with NUMA architecture, particularly when using a large InnoDB buffer pool.

In Kubernetes, containers are not allowed to modify memory allocation policies due to common security restrictions. When MySQL tries to enable NUMA interleaving, Kubernetes blocks the request, which can lead to warnings or confusing log messages.

Because most cloud servers for Kubernetes use servers with only a single NUMA node, this MySQL setting provides no performance benefit in containerized environments. To keep MySQL behavior predictable and reduce unnecessary noise, the Operator now sets `innodb_numa_interleave` to `OFF` by default when it deploys Percona Server for MySQL 8.4. This reduces false alarms, and ensures MySQL runs smoothly in hardened Kubernetes environments.

You can still enable NUMA interleaving for your Percona Server for MySQL by passing custom MySQL options through the Operator. Be sure you understand the implications, as this requires a Kubernetes environment that allows NUMA policies.

Documentation updates

Updated the README in the Helm chart to include the all available options.

Deprecation, rename and removal

Deprecated options

The `spec.enableVolumeExpansion` option is deprecated. It remains working for backward compatibility but it will be removed in version 1.5.0. Use the `spec.storageScaling.enableVolumeScaling` option instead.

Known limitations

Binlog Server image incompatibility notice

The new version of the Binlog Server image included in this release is not compatible with the previous version.

To upgrade the Binlog Server you must start a fresh binlog collection. Here's how to do it:

1. Stop point-in-time recovery binlog collection
2. Create a new full backup
3. Change the binlog storage settings in your cluster configuration. It can be a new bucket or a new prefix for the existing bucket
4. Update the Binlog Server image
5. Re-enable point-in-time recovery

Considerations for using OpenShift 4.22

Starting with OpenShift 4.22, the way images with not fully qualified names are pulled has changed for repositories that share the same repository name on DockerHub and Red Hat Marketplace. By default the tags are pulled from Red Hat Marketplace. Specifying not fully qualified image names may result in the `ImagePullBackOff` error.

- **OLM installation:** Images are provided with the fully qualified names and are pulled from the Red Hat Marketplace/DockerHub registry.
- **Manual install/update with default manifests:** Images must use the `docker.io` registry prefix to guarantee successful download from the DockerHub `percona-server-mysql-operator` repository.

For manual installation or update, follow the instructions below:

1. Clone the Operator repository:

   ```
   git clone -b v1.2.0 https://github.com/percona/percona-server-mysql-operator
   cd percona-server-mysql-operator
   ```

2. Edit the `deploy/bundle.yaml` file.
   - Locate the Deployment custom resource for the Operator.
   - Update the `spec.image` field to

   ```
   docker.io/percona/percona-server-mysql-operator:1.2.0
   ```

3. Apply the updated `deploy/bundle.yaml` file:

   ```
   oc apply --server-side -f deploy/bundle.yaml
   ```

4. Install Percona Server for MySQL:

   ```
   oc create -f deploy/cr.yaml
   ```

For the upgrade steps, follow the Update documentation.

CRD changes

- A new CRD `PerconaServerMySQLClusterSet` is added

Changelog

New Features

- K8SPS-127 - Added automatic storage scaling that monitors PVC usage and triggers resizing when usage crosses your configured threshold. This helps prevent outages from full disks and reduces manual capacity planning by expanding storage only when it is needed.
- K8SPS-409 - Added encrypted backups to ensure backup data stored in S3, GCS, or Azure is protected at rest with a customer-controlled key.
- K8SPS-496 - Added the ability to disable NodePort allocation for LoadBalancer Services created by the Operator when your load balancer does not need them.
- K8SPS-508 - Added cross-site replication for Group Replication clusters with declarative topology management from Kubernetes via a new `PerconaServerMySQLClusterSet` Custom Resource.
- K8SPS-512 - Added HAProxy version reporting in the cluster status so proxy version information is available from the start of the cluster lifecycle. This ensures telemetry and monitoring integrations receive accurate HAProxy version data instead of an empty value.
- K8SPS-627 - Added the `pmm.customClusterName` option so you can assign a custom name to clusters monitored by Percona Monitoring and Management (PMM). PMM can now correctly group source and replica clusters across regions or namespaces as a single deployment on dashboards.
- K8SPS-689 - Added a dedicated root user Secret that exposes connection details for your cluster in a single, operator-maintained resource.

Improvements

- K8SPS-19 - Enabled authentication for the Orchestrator HTTP API so only authorized clients can trigger failovers or topology changes. The Operator handles credentials automatically for its own calls, while custom scripts that access the API directly must provide valid credentials.
- K8SPS-434 - Introduce the ability to configure the maximum number of concurrent reconciles via an environment variable so that the Operator can continue reconciling other clusters while an upgrade is in progress. This improves responsiveness in multi-cluster environments where a long-running upgrade previously stalled all other reconciliation work.

- K8SPS-46 - Added the ability to pass custom Orchestrator configuration to the Operator without changing Operator defaults.
- K8SPS-487 - Improved Vault encryption Secret handling by validating the Secret when data-at-rest encryption is enabled and reporting clear status errors when it is missing or invalid. This lets you deploy Vault and the database cluster together and see configuration problems immediately instead of digging through Pod logs.
- K8SPS-497 - Improved graceful shutdown of the pt-heartbeat sidecar by handling SIGTERM in the entrypoint script. MySQL Pods now terminate faster during deletion or rolling updates instead of waiting for the full termination grace period while heartbeat keeps running.
- K8SPS-503 - Improved Group Replication bootstrap logging to reflect clone-based recovery when `group_replication_clone_threshold` requires it. Bootstrap logs now match the actual recovery method, making troubleshooting less confusing for users.
- K8SPS-525 - Added timestamps to failed backup Pod logs so you can tell whether a log entry belongs to the failed backup job or an earlier run. This makes it easier to correlate backup failures with the correct attempt when reviewing Pod or sidecar logs.
- K8SPS-716 - Added ability to specify binlog storage through the `backupSource` field in the `PerconaServerMySQLRestore` object. You can configure point-in-time recovery to a new cluster from a single restore manifest without pre-configuring binlog storage on the target cluster.
- K8SPS-742 - Added independent resource configuration for the `mysql-monit` sidecar container in HAProxy Pods via `spec.proxy.haproxy.sidecarResources`. You can set appropriate CPU and memory limits for the lightweight monitoring sidecar without over-provisioning it and save on infrastructure costs.

Bug Fixes

- K8SPS-296 - Improved the logging behavior for unhandled errors and stack traces that appeared when changing passwords in asynchronous replication clusters. Password rotation now completes cleanly without noisy reconciler errors in the Operator logs.
- K8SPS-454 - Fixed an issue where MySQL cluster primary selection failed after a minor version upgrade, leaving all instances read-only with no primary. The Operator now handles post-upgrade topology recovery correctly so the cluster regains a writable primary. (Thank you Alexander Kuleshov for reporting this issue)
- K8SPS-607 - Fixed reconciler errors that flooded Operator logs after deleting a `PerconaServerMySQLRestore` object. The restore controller now stops reconciling gracefully once the restore resource is removed.

- K8SPS-638 - Fixed an issue where the Group Replication plugin and related configuration were loaded even when `spec.mysql.clusterType` was set to `async`. Asynchronous clusters no longer show Group Replication settings in Pod specs, logs, or configuration that were never requested.
- K8SPS-680 - Fixed an issue where the `mysql-0` Pod could remain stuck in Terminating state until the termination grace period expired when deleting a cluster. Cluster deletion now completes promptly on affected platforms.
- K8SPS-682 - Fixed an issue where the default enablement of NUMA interleaving in MySQL 8.4 caused container deployment errors under hardened security policies. The Operator now correctly restricts this option inside host VMs where MPOL_INTERLEAVE limits are active to guarantee pod initialization stability.
- K8SPS-683 - Fixed the SmartUpdate behavior so that the Operator performs an explicit switchover before updating the primary Pod instead of deleting it directly. This applies to both Group Replication and asynchronous replication and reduces disruption during rolling upgrades.
- K8SPS-686 - Fixed an issue where scheduled backups could start while a cluster was still initializing after a restore with `unsafeFlags.backupNonReadyCluster` enabled. Backups now wait until the database is ready again after restore completes.
- K8SPS-699 - Fixed incremental backup restore when a failed backup exists in the chain by verifying the incremental chunk's completeness when building the restore chain. Failed or incomplete incremental backups are now skipped to ensure the restore succeeds.
- K8SPS-707 - Improved validation for the `spec.incrementalBaseBackupName` field to prevent specifying an incremental backup as the base one. The Operator now rejects invalid base backup references instead of creating an unsupported nested incremental chain.
- K8SPS-709 - Fixed binlog server crashes when an S3-compatible bucket accumulated a very large number of objects over sustained operation. The Operator handles large binlog inventories more reliably so point-in-time recovery does not require manual bucket cleanup and Pod recreation.
- K8SPS-739 - Fixed password propagation for MySQL Router clusters by using `Router.Size` instead of `HAProxy.Size` in the passwords-propagated check. Router-based deployments now receive credentials on all database Pods as intended. (Thank you user @jackiesre721 for contribution)
- K8SPS-741 - Fixed a race when a partially pre-created user Secret caused the Operator to copy incomplete credentials to the internal Secret before auto-generating missing system user passwords. MySQL Pods now start reliably even when the user-supplied Secret contains only a subset of required keys.

- K8SPS-744 - Fixed an issue where a custom `replica_parallel_workers` value in the cluster configuration was overridden to the default during bootstrap. The Operator now preserves your configured parallel applier thread count instead of resetting it to default value.
- K8SPS-751 - Fixed Version Service request errors when `spec.upgradeOptions.apply` was not set in the Custom Resource. Clusters with telemetry enabled no longer generate repeated HTTP 500 errors against the version check endpoint on every reconcile loop.
- K8SPS-755 - Fixed AutoRecovery bootstrap order so the Operator does not bootstrap `mysql-0` first when another Pod was the Group Replication primary before a full cluster crash. Recovery now respects the correct member order to avoid data loss after quorum loss.
- K8SPS-760 - Fixed an Operator panic when `spec.backup` was omitted from the Custom Resource entirely. Clusters without a backup section now reconcile normally instead of crashing the controller. (Thank you Wilfried ROSET for contributing to this issue)

Supported software

The Operator was developed and tested with the following software:

- Percona Server for MySQL 8.4.10-10.1
- Percona Server for MySQL 8.0.46-37.1
- XtraBackup 8.4.0-6.1
- XtraBackup 8.0.35-36.1
- MySQL Router 8.4.10
- MySQL Router 8.0.46
- HAProxy 2.8.18-1
- Orchestrator 3.2.6-22
- Percona Toolkit 3.7.1-4
- PMM Client 3.8.1
- Cert Manager 1.20.2
- Percona Binlog Server 0.3.1

Other options may also work but have not been tested.

Supported platforms

Percona Operators are designed for compatibility with all CNCF-certified Kubernetes distributions. Our release process includes targeted testing and validation on major cloud provider platforms and OpenShift, as detailed below:

- Google Kubernetes Engine (GKE) 1.33 - 1.35
- Amazon Elastic Kubernetes Service (EKS) 1.33 - 1.36
- OpenShift 4.18 - 4.22
- Minikube 1.38.1 with Kubernetes v1.35.1

This list only includes the platforms on which the Percona Operators are specifically tested as part of the release process. Compatibility with other Kubernetes flavors and versions depends on the backward compatibility provided by Kubernetes itself.

Percona certified images

Find Percona's certified Docker images that you can use with Percona Operator for MySQL based on Percona Server for MySQL in the following table:

Image	Digest
percona/percona-server-mysql-operator:1.2.0	76deed6a6daca2846fba6046b6646e25d22bca2d6d58a0b567e0a2c1240fe103
percona/percona-server-mysql-operator:1.2.0 (ARM64)	e0dd1e6bf1fd90b2149290500997ee791101f4500b5162bcea5694b1b9d7ab58
percona/percona-server:8.4.10-10.1	d24391a363426239220c35b9707d6b26ce7522ac27abe55689baebfb48bd9fb3
percona/percona-server:8.4.10-10.1 (ARM64)	70f6c4d01b5807737cdd423ab32af1feb1d00513b5420a84361773b167aeca87
percona/percona-server:8.0.46-37.1	1552a3b1860bb12325755295fb833890eb920c34fc212dbadb34c59baf0c378a
percona/percona-server:8.0.46-37.1 (ARM64)	e41dfb981de432da1a117077e7230008422ab49d6c0e9dd8d17a03d7ffde5dc0

percona/percona-xtrabackup:8.4.0-6.1	d135aadaae9e2f947cb2002f982f7b4c6e177b1c7e3d543ef7795aea999feedd
percona/percona-xtrabackup:8.4.0-6.1 (ARM64)	fcf2b3fc20cfbfa6d47ec60cecd881f915beacdae838f994dda984baea825293
percona/percona-xtrabackup:8.0.35-36.1	5416f19ac087e4d8e06eeff63cf1e075dc3b4e45577abbb9c3d87ae1ea3816c5
percona/percona-xtrabackup:8.0.35-36.1 (ARM64)	b604421b30bdd7ddfbc716220a8f8f32077c12a762d1eae622dcf4e5085b115a
percona/percona-mysql-router:8.4.10-10.1	7edd16793022a518842aa6e709582af9175ee9ab145faf11e930f8175f16c588
percona/percona-mysql-router:8.4.10-10.1 (ARM64)	e3be4405858a4198ecb1265365763de33dc6991bdb811bbc10803a92d06606a0
percona/percona-mysql-router:8.0.46-37.1	826cd7a26e918fc5b75cad6ba19d74b508dd1a73f93e49cfac727f6f40b74a02
percona/percona-mysql-router:8.0.46-37.1 (ARM64)	d7a835fff71c6fa881d35cc54fb126c4027ccd9692575c6b95c2cd6329915b90
percona/percona-orchestrator:3.2.6-22	384bcf3121f50e9536fc911532386b9b485a52701e1e99ada51a5faf91d667c3
percona/percona-orchestrator:3.2.6-22 (ARM64)	7ed29d0cde9396687fa42e1640ac2554125fd3a1e2a9f8d70667bbd00b386703
percona/haproxy:2.8.18-1	09e4d2ce9e65dc4aec9195e818e6da2041aea1a2bdb04f868d8c42ee81090dbf
percona/haproxy:2.8.18-1 (ARM64)	563d84d64e1668cb4f6b2202fbe15a18e0733b48fe963383bb877c9e3a8a0abd
percona/percona-toolkit:3.7.1-4	0865492cb600f1092618de63e91ae480606a0459d86e5eb876c7e7fb6e6cdda1

percona/percona-toolkit:3.7.1-4 (ARM64)	c0ebb11297c3bd83bbbc7acd9195dfa754e7e9be9da70800d146381a7d652129
percona/pmm-client:3.8.1	a92cfb7f912bd85d8245575c3ee5c423664ad2baedb674d159a87b113dbd4de2
percona/pmm-client:3.8.1 (ARM64)	3fe427c0666337df7613824da5f3b5fb7397e849f70402ac557c1324c5d996e6
percona/percona-server-mysql-operator:1.2.0-binlog-server-0.3.1	53336b6ccbe463c255f7ca9caaf9390ceba2b9c7672b91998cbe5f44d84dc4e4
percona/percona-server-mysql-operator:1.2.0-binlog-server-0.3.1 (ARM64)	361695d2b77443e67aa7f2858e9345cff08967afa3e2a8820d4c5981963fa858

Find images for previous versions

Percona Operator for MySQL 1.1.0 (2026-04-17)

Installation

Percona Operator for MySQL brings production-grade automation to MySQL deployments on Kubernetes. It handles provisioning, scaling, backups, failover, and upgrades using declarative Custom Resources, thus reducing manual effort and human error. With built-in support for Percona XtraBackup, proxies such as HAProxy or MySQL Router and Percona Toolkit, it ensures resilient, secure, and performant MySQL clusters.

Release highlights

This release focuses on backups improvements enabling more efficient and flexible data protection strategies. It also includes a handful of improvements and bug fixes.

Point-in-time recovery (tech preview)

This release introduces point-in-time recovery, giving you precise control over how far back you restore your MySQL cluster. Instead of recovering only to the moment a backup was taken, you can now roll the database forward to a specific transaction or timestamp. This is invaluable when you need to undo a bad migration, recover right before someone dropped the wrong table, or meet tighter RPO requirements with minimal data loss.

Point-in-time recovery works the same way in both asynchronous and group replication clusters, ensuring consistent recovery behavior regardless of your topology. It uses the Percona Binlog Server to collect binary logs and the `mysqlbinlog` client to apply them during the restore. Read more about the workflow in our documentation.

Point-in-time recovery is released as a **tech preview**. We do not recommend using it in production environments yet. However, we strongly encourage you to try it out in staging or test clusters and share your feedback. Your input will directly shape how we refine and finalize this capability in future releases.

Incremental backups (tech preview)

This release introduces incremental backups for MySQL clusters, giving you a faster, more efficient, and more cost effective way to protect your data. Instead of creating a full backup every time, the Operator now captures only the changes since the previous backup, significantly reducing backup size, storage usage, and data transfer overhead. Incremental backups also lower the load on your cluster, helping you maintain performance even during frequent backup operations. This feature works seamlessly with all supported backup storages and integrates with both scheduled and on-demand backup jobs.

Incremental backups are released as a **tech preview** feature and we don't recommend them for production environments yet. However, we encourage you to try them out and leave your feedback. This will help us shape the future of this functionality.

Learn more about incremental backups in our documentation.

Backup compression

Percona Operator for MySQL now supports backup compression, giving you faster, lighter, and more cost efficient backups.

Compressed backups are smaller in size, which means they stream to object storage quicker and consume less storage space as compared to uncompressed ones. In practice, this reduces both storage and data transfer costs.

Compression works for full and incremental backups, and you can use it for both on demand and scheduled backup workflows. Configure compression globally or per backup storage directly in the Custom Resource . You can also override the compression algorithm for a specific on-demand backup.

During the restore, the Operator automatically detects whether a backup is compressed and decompresses it as part of the restore workflow, so no additional steps are required from you.

Currently the Operator supports only the `zstd` compression algorithm. There is a known limitation for the `lz4` compression algorithm and it will be lifted after the PXB-3568 is resolved.

To learn more about compressed backups, read our documentation.

Configure file descriptor limit for HAProxy

When HAProxy performs external MySQL health checks, it tries to close every file descriptor (FD) up to the system limit before executing the check script. Some systems set this limit extremely high, so HAProxy ends up looping through millions or even billions of numbers. This causes heavy CPU use, long delays and timeouts.

This release sets a safe file descriptor limit in the entrypoint script before HAProxy container starts. The default FD limit value is `1048576`.

You can change this limit with the `HA_RLIMIT_NOFILE` environment variable in the Custom Resource. The value is checked, and if it is invalid, the Operator falls back to the default value. If the value is too large, the Operator uses the hard limit file descriptor value. This makes external checks fast, stable, and predictable.

Configurable reconnect attempts for clusters with asynchronous replication

In some cases, the default number of attempts for a replica to reconnect to its source is not sufficient, causing the cluster to become stuck. You can now fine-tune this behavior using the following environment variables:

- `ASYNC_SOURCE_RETRY_COUNT` - controls the number of reconnection attempts
- `ASYNC_SOURCE_CONNECT_RETRY` - adjusts the reconnect timeout for MySQL Pods

Specify new values directly in the Custom Resource to match your environment's needs.

Read more about other environment variables in our documentation.

This makes cluster management more predictable and avoids unexpected stalls during replication interruptions.

Documentation updates

- Completed the install on Openshift documentation with instructions how to install using OLM
- Added a tutorial how to upgrade the Operator on OpenShift
- Enhanced Helm install documentation explaining how to install the Operator with customized parameters and apply custom naming to releases

CRD changes

- The `.status.storage.s3.storageClass` has now the type `string`
- The `v1.^.status.storage.s3.storageClass` field is removed
- Added the `incrementalBaseBackupName` option. This option can be set only when the backup configuration has `spec.type` set to `incremental`, and its value must refer to an existing full backup.

Changelog

New features

- K8SPS-410 - Added the ability to make incremental backups to save only the changes since the previous backup.
- K8SPS-642 - Introduced Point-in-Time Recovery (PITR) support for enhanced data protection. This feature enables the binlog server to collect binary logs, allowing users to restore their database to any specific timestamp.

Improvements

- K8SPS-69 - Updated the readiness probe to fail if replication threads have stopped. This prevents application traffic from being routed to nodes that are no longer receiving updates from the primary.
- K8SPS-96 - Added support for backup compression using Percona XtraBackup. This enhancement significantly reduces the storage footprint and transfer time for all database backups.
- K8SPS-215 - Improved restoration logic by automatically removing unnecessary PVCs during asynchronous replication restores. This ensures a clean environment and prevents resource conflicts when bringing pods back online.
- K8SPS-435 - Prevented scheduled backups from running if the database is in an unhealthy state. This change avoids redundant system overhead and failed backup attempts when the cluster cannot reliably provide data.
- K8SPS-467 - Set the default temporary path for PMM agents to /tmp/pmm to improve platform compatibility. This update specifically resolves file system permission issues encountered on platforms like OpenShift.

- K8SPS-595 - Replaced Operator panics with structured error handling when invalid storage configurations are detected. Users will now receive clear diagnostic messages instead of operator crashes, facilitating easier troubleshooting.
- K8SPS-601 - Reclassified status changes and replication warnings as “Normal” event types in Kubernetes. This update reduces monitoring noise by distinguishing expected operational transitions from critical failures.
- K8SPS-622 - Made the async replication retry count configurable via the Custom Resource. Users can now tune the `SOURCE_RETRY_COUNT` parameter to better match their specific network conditions and stability requirements.
- K8SPS-666 - Optimized how file descriptors are managed in the HAProxy entrypoint to avoid performance bottlenecks. This fix prevents HAProxy from attempting to close thousands of unused file descriptors during health checks.

Fixed bugs

- K8SPS-530 - Fixed an issue where the `percona/delete-backup` finalizer would block the deletion of backups stuck in the starting state. Users can now immediately remove pending or failed backup resources without waiting for a timeout.
- K8SPS-616 - Removed the requirement for a backup image when backups are explicitly disabled in the configuration. This simplifies the Custom Resource definition by eliminating unnecessary parameters for unused features.
- K8SPS-623 - Resolved an “Access Denied” error when performing GCS backups to private buckets. The Operator now correctly validates and uses credentials to authenticate storage access during the backup process.
- K8SPS-635 - Fixed a bug where annotations for MySQL, HAProxy, and Router were not correctly applied to all components. This ensure that custom metadata and third-party integrations work consistently across the entire cluster deployment.
- K8SPS-653 - Corrected a validation error where the Operator would default to asynchronous replication even when not explicitly requested. The Operator now adheres to the intended cluster type, preventing unexpected deployment failures.
- K8SPS-661 - Resolved an issue where primary nodes retained stale replication configurations after a Kubernetes node restart. This fix ensures that primary nodes are always initialized with the most current cluster state. (Thank you Alexander Khozya for reporting this issue)

- K8SPS-669 - Enabled automated recovery for clusters that have suffered a complete loss of quorum. The Operator can now detect and rebuild the cluster automatically, significantly reducing recovery time is case of failures.
- K8SPS-684 - Resolved a connection drop issue that occurred whenever the HAProxy configuration was reloaded. This ensures stable and uninterrupted application connectivity during proxy maintenance or scaling events.

Supported software

The Operator was developed and tested with the following software:

- Percona Server for MySQL 8.4.8-8.1
- Percona Server for MySQL 8.0.45-36.1
- XtraBackup 8.4.0-5.1
- XtraBackup 8.0.35-35.1
- MySQL Router 8.4.8
- MySQL Router 8.0.45
- HAProxy 2.8.18-1
- Orchestrator 3.2.6-20
- Percona Toolkit 3.7.1
- PMM Client 3.7.0
- Cert Manager 1.19.1
- Percona Binlog Server 0.2.1

Other options may also work but have not been tested.

Supported platforms

Percona Operators are designed for compatibility with all CNCF-certified Kubernetes distributions. Our release process includes targeted testing and validation on major cloud provider platforms and OpenShift, as detailed below:

- Google Kubernetes Engine (GKE) 1.32 - 1.35
- Amazon Elastic Kubernetes Service (EKS) 1.33 - 1.35

- Azure Kubernetes Service (AKS) 1.33 - 1.35
- OpenShift 4.18.36 - 4.21.8
- Minikube 1.38.1 based on Kubernetes v1.35.1

This list only includes the platforms on which the Percona Operators are specifically tested as part of the release process. Compatibility with other Kubernetes flavors and versions depends on the backward compatibility provided by Kubernetes itself.

Percona certified images

Find Percona's certified Docker images that you can use with Percona Operator for MySQL based on Percona Server for MySQL in the following table:

Image	Digest
percona/percona-server-mysql-operator:1.1.0	c44447339098998367dcc84028ad065a3872c9b8b1239d5b338e90f28f984107
percona/percona-server-mysql-operator:1.1.0 (ARM64)	55ffe11625f6453c67e394c5cc757746d8a5c58022cd6285af5a5072f850ff4e
percona/percona-server:8.4.8-8.1	5203959150b5afe769f04f17453ae22d63ea889dfbbdef6acc013e999cc31552
percona/percona-server:8.4.8-8.1 (ARM64)	831ac32171d0b794826e276745cf3e7526b63c62447fdb266b9e308e7028319e
percona/percona-server:8.0.45-36.1	27845b5451a1b707df70e832d51bcd3c384e6cb3cc02799a7bc01ec6cd578c2d
percona/percona-server:8.0.45-36.1 (ARM64)	1dae69344a18d21800fd65752a1450ddaac31b50d733d3eb1507f28ec824172d
percona/percona-xtrabackup:8.4.0-5.1	1c7e20fac192f70de2233e471a9243ba9a65399e9667cf954f5fc5afba1b9aa4
percona/percona-xtrabackup:8.4.0-5.1 (ARM64)	dd6089277586865a7debfbe2e8b3a894632f9240fd1aec8c77c4f71cb9750b90

percona/percona-xtrabackup:8.0.35-35.1	f44162b0bede26d74f23c1cd3ab58efb0f43529f7fe09d221e3ad7b473463f0b
percona/percona-xtrabackup:8.0.35-35.1 (ARM64)	68908749139e3904d81a21a0cbb7ff85b08dc7a65d9d063b17dfff278ad81107
percona/percona-mysql-router:8.4.8	c087433f2824a0d53e297d13eb8db995b3b6cb6c90491d894a0494bb1ccbf6bd
percona/percona-mysql-router:8.4.8 (ARM64)	c087433f2824a0d53e297d13eb8db995b3b6cb6c90491d894a0494bb1ccbf6bd
percona/percona-mysql-router:8.0.45	0b1a1ba2005eb7be254d2e7b851802c33081f428d4f8524ed5f62a354499fa2e
percona/percona-mysql-router:8.0.45 (ARM64)	0b1a1ba2005eb7be254d2e7b851802c33081f428d4f8524ed5f62a354499fa2e
percona/percona-orchestrator:3.2.6-20	13ae84e75279201da09ed8cd1936a5306fa2f29b2cc74478974f16a5d1e05e6c
percona/percona-orchestrator:3.2.6-20 (ARM64)	1d0f9b414fcadebe6d3c2e6778e58e0f17b42c64a4293f03efc744b1d27bef99
percona/haproxy:2.8.18-1	fa668cec0a541ce862ecd8fd781df4631e837e085dde6b5ae2a4bb678cc84024
percona/haproxy:2.8.18-1 (ARM64)	8271dafc8db4d1a4e5a446b3618ec8bed95837c7e9066cce0898ffefdf6b36f1
percona/percona-toolkit:3.7.1-3	7fd2092b0ac8addf44163a5a7e1999acf5ae34ccffe77aeb005aa6ea8a8cfc5d
percona/percona-toolkit:3.7.1-3 (ARM64)	afa0a0c7826433071da8f16077830977522240f03959e0b57db14ecad2357cca
percona/pmm-client:3.7.0	3ddfd925a9f6bb0daee88021e0310f8375f550afd70b7f2d0980509b7f3fb777
percona/pmm-client:3.7.0 (ARM64)	2701042087701c70666fd88383bfac22368c339a033b15e364bd6dd417ad1922

percona/percona-server-mysql-operator:1.1.0-binlog-server-0.2.1	9ffb7db0094dc0c39f6ec794b70a33db6e4cd19aa4ca38cd6a9b05bc5bbea63c
percona/percona-server-mysql-operator:1.1.0-binlog-server-0.2.1 (ARM64)	97ad1244d08773eb6e6121c4b45424d88d2a624fbd26eb40ab907d054999e154

Find images for previous versions

Percona Operator for MySQL 1.0.0 (2025-11-17)

Installation

Percona Operator for MySQL brings production-grade automation to MySQL deployments on Kubernetes. It handles provisioning, scaling, backups, failover, and upgrades using declarative Custom Resources, thus reducing manual effort and human error. With built-in support for Percona XtraBackup, proxies such as HAProxy or MySQL Router and Percona Toolkit, it ensures resilient, secure, and performant MySQL clusters.

Release highlights

This release marks the **General Availability (GA) of Percona Operator for MySQL using Percona Server for MySQL with the group replication type**. The asynchronous replication has the tech preview status and we don't recommend using it in production yet.

With the GA status of the Operator, you can confidently deploy and run it in production environments, benefiting from long-term maintenance and enterprise-grade reliability.

Alternatively, you may opt for Percona Operator for MySQL based on Percona XtraDB Cluster, which is production-ready and contains everything you need to quickly and consistently deploy and scale MySQL clusters in a Kubernetes-based environment, on-premises or in the cloud.

This release focuses on stability and bug fixing, ensuring the Operator is ready for production use. Additionally, it introduces these improvements:

Seamless Operator lifecycle management on OpenShift OLM

The Operator images are passing official certification for OpenShift. When passed, this unlocks full support for the Operator Lifecycle Manager (OLM) so that you can install, upgrade and manage the Operator's lifecycle directly from the OpenShift console.

What this means for you:

- Simplified installation: Deploy Operators directly from the OpenShift UI with just a few clicks.
- Streamlined updates: Stay current with automatic or manual updates via OLM.

- Enterprise-grade assurance: Certified images meet Red Hat's security and compatibility standards.
- Better integration: Leverage OpenShift-native workflow for lifecycle management, RBAC, and monitoring.
- Scalable operations: Simplify cluster-wide rollouts and reduce manual overhead.

All OpenShift-related features will become available to users as soon as certification is confirmed. Whether you're a platform engineer, DBA, or architect, this advancement will bring you closer to a secure, scalable, and policy-driven infrastructure.

Streamlined custom configuration usage for backup and restore processes

In previous version we have added the ability to fine-tune backups and restores by defining `xtrabackup`, `xbstream` and `xbcloud` settings globally via the Custom Resource manifest, or individually via a specific backup / restore manifest.

In this release we improved how the Operator applies these settings: now individual configuration always takes precedence over global settings.

With this improvement you have maximum flexibility: you can define consistent default settings for the entire cluster, but still tailor individual backup or restore operations as needed. This way you can optimize performance, troubleshoot, or customize specific scenarios without affecting the global configuration.

Increased timeouts for read, write and clone operations inside MySQL cluster

To improve reliability of clone operations in asynchronous MySQL clusters, especially when transferring large datasets, we've increased the default timeouts for read, write and clone operations to 3600 seconds. This change helps prevent premature failures caused by network delays or slow disk I/O during large data transfers.

The following timeouts are now set to 3600s by default:

- `BOOTSTRAP_CLONE_TIMEOUT`
- `BOOTSTRAP_READ_TIMEOUT`
- `BOOTSTRAP_WRITE_TIMEOUT`

You can fine-tune the timeout for cloning in your custom resource (CR) using environment variables:

```
spec:
  mysql:
    env:
      - name: BOOTSTRAP_CLONE_TIMEOUT
        value: "3600"
      - name: BOOTSTRAP_READ_TIMEOUT
        value: "3600"
      - name: BOOTSTRAP_WRITE_TIMEOUT
        value: "3600"
```

This update ensures smoother provisioning and bootstrapping of new database nodes, especially in environments with large datasets or variable network conditions.

Deprecation, rename and removal

- Changed paths for example configuration files for backups and restores. They are now stored in the `deploy/backup/` folder. Adjust your automation workflow with this new path, if needed.
- The Custom Resource options `spec.pmm.readinessProbes` and `spec.pmm.livenessProbes` have been renamed to the singular `spec.pmm.readinessProbe` and `spec.pmm.livenessProbe`, respectively. Please update your application configurations to use these new field names as needed.

Known limitations

If you defined several schedules for the same remote backup storage, be aware of the following limitations:

1. **Retention policy conflicts.** The Operator applies retention policies only to the first schedule in your configuration. For example, if you set daily backups to keep 5 copies and monthly backups to keep 3 copies, the Operator will only keep 5 total backups in storage, not 8 as you might expect. However, all backup objects will still appear in `kubectl get ps-backup` output.
2. **Concurrent backup conflicts.** When multiple schedules run simultaneously and write to the same storage path, backups can overwrite each other, resulting in incomplete or corrupted data.

To avoid these issues and ensure each schedule maintains its own retention policy, configure separate storage locations for each schedule. Refer to the documentation for more information and configuration steps.

Changelog

Improvements

- K8SPS-469 - Improved log message to display clearer and more informative error messages in case of authorization issues to a backup storage.
- K8SPS-537 - Extended the test suite for the automatic update process to include MySQL version 8.4.
- K8SPS-574 - Align readiness and liveness probe naming to be in the singular form to correspond to to the Kubernetes API structure

Fixed bugs

- K8SPS-491 - Percona Operator for MySQL now automatically generates the secrets object in the format `<cluster-name>-secrets`, if it's not explicitly defined in the Custom resource, preventing common startup errors.
- K8SPS-492 - Fixed the issue with the Operator sending the unsupported `Error` event type during the Group Replication cluster startup by sending the `Warning` event type instead.
- K8SPS-498 - Stopped unnecessary updates to the `resourceVersion` field of the cluster objects during its initialization.
- K8SPS-501 - Fixed the issue with the Operator failing to update the PVC when expanding database volumes by retrying the operation.
- K8SPS-517 - Fixed an issue that prevented MySQL clone operations from completing successfully due to a default 10-second read timeout, which caused "query interrupted" errors. This was resolved by increasing the default read/write timeouts to 3600 seconds (1 hour) for long-running operations and enhancing error handling for better reliability and debugging.
- K8SPS-518 - ConfigMap settings were fixed to ensure proper labels are applied when deploying clusters with various replication and router configurations.
- K8SPS-521 - Fixed an issue where mysql-shell would overwrite Group Replication options in my.cnf during cluster creation. The operator now parses my.cnf and explicitly passes user-defined settings (like group_replication_single_primary_mode and group_replication_paxos_single_leader) to dba.createCluster(), ensuring user customizations are respected.
- K8SPS-524 - Fixed the issue with successful backups displaying the incorrect state description. The state description field is irrelevant for successful backups and is not present.

- K8SPS-529 - Removed reconciliation of backups that entered the error state due to underlying configuration issues and keep their state for troubleshooting.
- K8SPS-533 - The individual configuration for `xbcloud`, `xbstream` and `xtrabackup` tools specified directly for a backup or a restore object now fully overrides any default arguments set in the cluster's custom resource.
- K8SPS-535 - Backup deletion operations no longer display an erroneous failure message when the backup is successfully removed from storage.
- K8SPS-539 - Fixed the issue with excessive CPU utilization of the `ps-entrypoint.sh` recovery by adding a backoff mechanism to the file existence check. This improves cluster operation in environments where CPU resources are a concern.
- K8SPS-548 - Improved Group Replication self-healing test and resolved a sporadic failure where a pod would not become ready after a full cluster crash.
- K8SPS-456,K8SPS-550 - Fixed the issue with service accounts defined for HAProxy Pods via Custom Resource not being applied.
- K8SPS-556 - Improved logging when telemetry server is unavailable by placing them to Debug level
- K8SPS-560 - Fixed the issue with scheduled backups failing due to conflicting job names when multiple backups run concurrently.
- K8SPS-564 - Fixed the issue with both HAProxy or Router being deployed when both are enabled by validating the configuration and either reporting the error or deploying only one proxy. This prevents unintended dual deployments.
- K8SPS-565 - Restores now complete successfully when using an auto-generated secrets name for clusters.
- K8SPS-598 - Fixed the issue with unneeded cluster restart after the Operator image update. The Operator now prevents these restarts by adding the logic to compare the Operator's new image version with the Custom Resource version, ensuring the init container image in the StatefulSet is only updated when necessary.

Supported software

The Operator was developed and tested with the following software:

- Percona Server for MySQL 8.4.6-6.1
- Percona Server for MySQL 8.0.43-34.1

- XtraBackup 8.4.0-4.1
- XtraBackup 8.0.35-34.1
- MySQL Router 8.4.6-6.1
- MySQL Router 8.0.43-34.1
- HAProxy 2.8.15
- Orchestrator 3.2.6-18
- Percona Toolkit 3.7.0-2
- PMM Client 3.4.1
- Cert Manager 1.19.1

Other options may also work but have not been tested.

Supported platforms

Percona Operators are designed for compatibility with all CNCF-certified Kubernetes distributions. Our release process includes targeted testing and validation on major cloud provider platforms and OpenShift, as detailed below:

- Google Kubernetes Engine (GKE) 1.31 - 1.33
- Amazon Elastic Container Service for Kubernetes (EKS) 1.31 - 1.34
- OpenShift 4.16 - 4.20
- Minikube 1.37.0 with Kubernetes v1.34.0

This list only includes the platforms on which the Percona Operators are specifically tested as part of the release process. Compatibility with other Kubernetes flavors and versions depends on the backward compatibility provided by Kubernetes itself.

Percona certified images

Find Percona's certified Docker images that you can use with Percona Operator for MySQL based on Percona Server for MySQL in the following table:

Image	Digest
percona/percona-server-	36d82324630c7b2030c6f96df8dc8433726c1236f915e790825a54571dbca7f3

mysql-operator:1.0.0	
percona/percona-server-mysql-operator:1.0.0 (ARM64)	fa9e3082d51d3c52f6cefbd1be129f4585effba7ca6221fd1234a481ddcd61a5
percona/percona-server:8.4.6-6.1	ea97c9df3e362728fc3819c28c841498f5a1765945b9556bc964b218b7d4dc97
percona/percona-server:8.0.43-34.1	315efeac572c48cc6f118bba7e0b2545ad396142f60328f2db9620ae0ad57e45
percona/percona-xtrabackup:8.4.0-4.1	840260525cf27e299b5edc7b48ad19caea03ad3ea7349000d0fc6de627b2fb10
percona/percona-xtrabackup:8.0.35-34.1	967bafa0823c90aa8fa9c25a9012be36b0deef64e255294a09148d77ce6aea68
percona/percona-mysql-router:8.4.6	e083c632c118cd4af472d9030a7900401f4b338e069c91996fe33747b77be985
percona/percona-mysql-router:8.0.43	3a420b803cd39c7c2a3ff414d45d1858df39599961339f5c02df0681f558ccdd
percona/percona-orchestrator:3.2.6-18	a8a70f8882925b0a1a46893376e29af73646117b22e1eeb5a0a89876a907651f
percona/haproxy:2.8.15	e64e468ac0ed2036ee164631469cc71821dcb84a6d568883f704d0eacaf84bb4
percona/percona-toolkit:3.7.0-2	b7a4a2ca71ebf2b35786ab614221cbefb032fd5dfb5c5a478efcdd23931dd70b
percona/pmm-client:3.4.1	1c59d7188f8404e0294f4bfb3d2c3600107f808a023668a170a6b8036c56619b
percona/pmm-client:3.4.1 (ARM64)	2d23ba3e6f0ae88201be15272c5038d7c38f382ad8222cd93f094b5a20b854a5

Find images for previous versions

Percona Operator for MySQL 0.12.0 (2025-09-23)

Installation

Percona Operator for MySQL allows users to deploy MySQL clusters with both asynchronous and group replication topology. This release includes various stability improvements and bug fixes, getting the Operator closer to the General Availability stage. Version 0.12.0 of the Percona Operator for MySQL is still **a tech preview release**, and it is **not recommended for production environments**.

As of today, we recommend using Percona Operator for MySQL based on Percona XtraDB Cluster, which is production-ready and contains everything you need to quickly and consistently deploy and scale MySQL clusters in a Kubernetes-based environment, on-premises or in the cloud.

Release highlights

Full MySQL 8.4 support now available

With this release, data-at-rest encryption is now supported for Percona Server for MySQL 8.4.

In the previous release, we have added support for Percona Server for MySQL 8.4 within the Operator. However, data-at-rest encryption was not yet available. That limitation has now been lifted, unlocking the full potential of Percona Server for MySQL's latest major version. Check our documentation for Percona Server for MySQL 8.4-specific setup instructions.

This improvement empowers you to take full advantage of Percona Server for MySQL 8.4's features while benefiting from seamless, automated lifecycle management provided by the Operator. Percona Server for MySQL 8.4 is now the default version for deploying a database cluster.

Ensure cluster availability with PodDisruptionBudgets

A PodDisruptionBudget (PDB) in Kubernetes helps keep your applications available during voluntary disruptions, such as deleting a deployment or draining a node for maintenance. A PDB sets a limit on how many Pods can be unavailable at the same time due to these voluntary actions.

With this release, you can now configure PodDisruptionBudgets for MySQL, HAProxy, MySQL Router, and Orchestrator Pods, thus ensuring your cluster remains available, even during disruptions or planned maintenance.

Fine-tune backup and restore operations

The Operator sets sensible defaults for backups and restores to ensure their smooth flow. If you need more control, you can fine-tune `xtrabackup`, `xbstream`, and `xbcloud` settings. You can do this globally via the `deploy/cr.yaml` Custom resource manifest or individually for a specific backup / restore operation via the respective `deploy/backup.yaml` or `deploy/restore.yaml` manifests. In either case, define your configuration in the `spec.containerOptions` subsection. For example:

```
spec:
  backup:
    storages:
      <STORAGE-NAME>:
         containerOptions:
           env:
           - name: CUSTOM_VAR
             value: "false"
           args:
             xtrabackup:
             - "--someflag=abc"
             xbcloud:
             - "--someflag=abc"
             xbstream:
             - "--someflag=abc"
```

Note that individual settings take precedence over the global ones. Read more about fine-tuning backups and restores and how the settings are applied in our documentation.

Monitor PMM Client health and status

Percona Monitoring and Management (PMM) is a great tool to monitor the health of your database cluster. Now you can also learn if PMM itself is healthy using probes - a Kubernetes diagnostics mechanism to check the health and status of containers. Use the `spec.pmm.readinessProbes.*` and `spec.pmm.livenessProbes.*` Custom Resource options to fine-tune Readiness and Liveness probes for PMM Client.

Define a source Pod for backups

You can now explicitly define from what MySQL instance Pod the Operator should make a backup. You can specify the Pod in the deploy/cr.yaml to apply it for all backups, both scheduled and on-demand. You can also override it for an on-demand backup in its resource manifest.

```
spec:
  backup:
      sourcePod: ps-cluster1-mysql-1
```

These options let you tailor your backup strategy to fit your organization's policies.

For asynchronous replication clusters, the Operator must know the cluster topology to run a backup. For this, either enable the Orchestrator in your deployment. Or specify the `sourcePod` value, if your cluster has more than one MySQL Pods.

Deprecation, rename and removal

- The `.spec.initImage` field has been replaced by the `.spec.initContainer` subsection, which follows Kubernetes best practices for defining containers that run before the main containers in a Pod. The `initContainer` feature is helpful for setup tasks such as:
- Initializing data
- Waiting for services to become available
- Setting permissions
- Pulling secrets or configuration files
- The default cluster name has been changed to `ps-cluster1` to prevent possible conflicts if you have custom resources of both Percona Operator for MySQL based on Percona Server for MySQL and Percona XtraDB Cluster in the same namespace.
- The API version in CRD has changed from `v1alpha` to `v1`. Read more about updates in Known limitations.

Known limitations

Due to the API version change, CRD updates are currently not supported. In order to update to version 0.12.0, you must manually delete the CRDs, apply new ones and recreate the cluster. To keep the data, do the following:

- check that the `percona.com/delete-mysql-pvc` finalizer is not enabled in `deploy/cr.yaml`
- don't delete PVCs manually
- Recreate the cluster with the same name. The Operator then automatically reuses the same PVCs.

Changelog

New features

- K8SPS-400 - Improved flexibility for backups and restores via adding support for custom options for `xtrabackup`, `xbstream`, and `xbcloud` binaries.
- K8SPS-405 - Users can now configure the LivenessProbe for the PMM Client container, allowing for custom timeouts and improved container health checks.
- K8SPS-413 - Add ability to set resources and `containerSecurityContext` for init containers.
- K8SPS-480 - Added support for data-at-rest encryption for MySQL 8.4.

Improvements

- K8SPS-172 - The operator now includes logs for all haproxy manipulations, providing better visibility for operations like adding, deleting, or downscaling.
- K8SPS-269 - All Kubernetes objects created by the Operator now have appropriate labels, including the Orchestrator configmap in async clusters. This improves object filtering and grouping.
- K8SPS-417 - Added ability to define PodDisruptionBudget, which helps manage voluntary disruptions to your cluster.
- K8SPS-427 - Simplified the Custom Resource (CR) validation logic by using Kubernetes validations for CR input.
- K8SPS-464 - The Operator will now automatically set the `crVersion` to the Operator's current version if it is not defined by the user.
- K8SPS-466 - Added ability to set global labels and annotations for all Kubernetes objects created by the Operator.
- K8SPS-478 - Improved bootstrapper behavior to determine if incremental recovery is possible and specify it when adding new instances to the existing cluster.
- K8SPS-488 - Switched to using API version `v1` in custom resource definitions

Bugs fixed

- K8SPS-371 - Added the ability to set a backup source Pod to ensure backups are made for clusters with asynchronous replication when the Orchestrator is disabled.

- K8SPS-374 - Fixed the issue with the Operator reporting the reconciliation error when an async cluster was being paused or recovered.
- K8SPS-378 - Fixed an issue where the cluster would remain in an unready state if the Orchestrator was scaled down to 1 Pod.
- K8SPS-430 - The Operator now updates TLS certificates when new Subject Alternative Names (SANs) are added to the CR.
- K8SPS-465 - Readiness and liveness probes have been added for HAProxy Pods to ensure their health.
- K8SPS-475 - Fixed an issue where the `exposePrimary.labels` field was incorrectly applied to the service selector. The exposed services now contain global labels together with the exposed labels and the selectors do not contain labels.
- K8SPS-494 - Fixed the issue with the constant update of the `resourceVersion` of the PerconaServerMySQL object after a cluster is created. The issue was caused by the Operator receiving stale objects during reconciliation, which resulted in the `InnoDBClusterBootstrapped` condition being set twice in every loop and constantly updating its last transition time. The Fix updates the status directly after setting the condition and waits for consistency with the API server.

Supported software

The Operator was developed and tested with the following software:

- Percona Server for MySQL 8.4.6-6
- Percona Server for MySQL 8.0.43-34
- XtraBackup 8.4.0-4
- XtraBackup 8.0.35-34
- MySQL Router 8.4.6-6
- MySQL Router 8.0.43-34
- HAProxy 2.8.15-1
- Orchestrator 3.2.6-18
- Percona Toolkit 3.7.0-2
- PMM Client 3.4.0
- Cert Manager 1.18.2

Other options may also work but have not been tested.

Supported platforms

Percona Operators are designed for compatibility with all CNCF-certified Kubernetes distributions. Our release process includes targeted testing and validation on major cloud provider platforms and OpenShift, as detailed below for Operator version 0.12.0:

- Google Kubernetes Engine (GKE) 1.30 - 1.33
- Amazon Elastic Container Service for Kubernetes (EKS) 1.31 - 1.33
- OpenShift 4.15 - 4.19
- Minikube 1.37.0 (based on Kubernetes 1.34.0)

This list only includes the platforms that the Percona Operators are specifically tested on as part of the release process. Other Kubernetes flavors and versions depend on backward compatibility offered by Kubernetes itself.

Percona certified images

Find Percona's certified Docker images that you can use with Percona Operator for MySQL based on Percona Server for MySQL in the following table:

Image	Digest
percona/percona-server-mysql-operator:0.12.0	a9d073efaee64250c7a97210232373909c951d0d6da3d67d68b36a212b5e5a5b
percona/percona-server-mysql-operator:0.12.0 (ARM64)	cb704b34cff91fd44e897d4aed0379ffb5c970625e8e5ecd7ffec2b12b0e5897
percona/percona-mysql-router:8.4.6	5b714f768c4cea30e85b31de7ab3958074814e8ccfbcad61db5109d3d80710b3
percona/percona-mysql-router:8.0.43	00047362aec0ee988e32f7133d41b723928a6ac98d38dcd10c18ca99e2718a53
percona/percona-orchestrator:3.2.6-18	6fa4c515363c2a89a13accb0a58ba66519329afc04ad31c400775ced96bc8c09

percona/percona-toolkit:3.7.0-2	17ef2b69a97fa546d1f925c74ca09587ac215085c392761bb4d51f188baa6c0e
percona/haproxy:2.8.15	e64e468ac0ed2036ee164631469cc71821dcb84a6d568883f704d0eacaf84bb4
percona/percona-xtrabackup:8.4.0-4.1	f9e859ffbc6a9db1b1e3c72a8545cfcb0c9d70271c3c3273007d033fed3772a6
percona/percona-xtrabackup:8.0.35-34.1	2dc127b08971051296d421b22aa861bb0330cf702b4b0246ae31053b0f01911e
percona/percona-server:8.4.6-6.1	dd0c67df12f5b13eac441dfa27b04a49ac449b6946adf100bc1ecebffd8074f4
percona/percona-server:8.4.5-5	61a6811372d919316640990922188005f92d58639a2472865d59e56b2a6050a1
percona/percona-server:8.0.43-34.1	d8a03675a2dd5d01a36ab0fe1c942091679bce90a241fb539cd31776ebf43aca
percona/percona-server:8.0.42-33	a7cbaa50c43483a07506f7cd5cccc4e587611f6500e1be5df0a93e339b9d3bc5
percona/percona-server:8.0.40-31	09276abecbc7c38ce9c5453da1728f3e7d81722c56e2837574ace3a021ee92f2
percona/percona-server:8.0.36-28	423acd206f94b34288d10ed041c3ba42543e26e44f3706621320504a010dd41f
percona/percona-server:8.0.33-25	14ef81039f2dfa5e19a9bf20e39aaf367aae4370db70899bc5217118d6fd2171
percona/pmm-client:3.4.0	9e8bf020be35eddc2a9e3a22e974234ce02c4818353e87e0522191df2743af3c
percona/pmm-client:3.4.0 (ARM64)	1f3e19db0a409e92ccd5238893000f4340bc309a44ebdc2a78eedf4d7948c766

Find images for previous versions

Percona Operator for MySQL 0.11.0 (2025-09-01)

Installation

Percona Operator for MySQL allows users to deploy MySQL clusters with both asynchronous and group replication topology. This release includes various stability improvements and bug fixes, getting the Operator closer to the General Availability stage. Version 0.11.0 of the Percona Operator for MySQL is still **a tech preview release**, and it is **not recommended for production environments**.

As of today, we recommend using Percona Operator for MySQL based on Percona XtraDB Cluster, which is production-ready and contains everything you need to quickly and consistently deploy and scale MySQL clusters in a Kubernetes-based environment, on-premises or in the cloud.

Release highlights

Support for MySQL 8.4

This release introduces support for Percona Server for MySQL 8.4.x. The Operator supports all major functionality for this latest major version except data-at-rest encryption. However, we do not recommended Percona Server for MySQL 8.4 for production environments yet.

Ensure data security with data at rest encryption

Data-at-rest encryption provides robust data protection by encrypting your database files on disk. Data is encrypted automatically, in real time, prior to writing to storage and decrypted when read from storage. The Operator uses the `keyring_vault` plugin to encrypt tablespace files and binlog. It integrates directly with HashiCorp Vault, giving you a secure and automated solution for managing encryption keys.

With this feature, you can meet your compliance requirements and protect sensitive data without the operational complexity. Learn how to configure it in our documentation

Note that data-at-rest encryption is currently not supported for Percona Server for MySQL 8.4.x. We plan to add it in future releases.

Support for `emptyDir` and `hostPath` volumes

You can now configure the Operator to use `emptyDir` or `hostPath` volumes for MySQL Pods, in addition to `persistentVolumeClaim` volumes. This extends the number of use cases for using the Operator, such as configuring additional storage for the data you don't need to persist when a Pod restarts, ephemeral workloads, testing CI/CD automation against a database and more.

Note the following key points for using volume types:

- Using `hostPath` can be risky in production, as it ties your Pod to a specific node and can lead to data loss if the node fails.
- `emptyDir` is not for persistent data.
- `persistentVolumeClaim` is the recommended way for persistent, portable storage in Kubernetes.

Improved security for user secrets with special characters in passwords

The Operator now generates stronger passwords using the combination of uppercase and lowercase letters, digits, and special characters like `! $ % & ( ) * + , - . < = > ? @ [ ] ^ _ { } ~ #`. These have been tested to ensure compatibility across SQL queries, shell scripts, YAML files, and connection strings.

The Operator excludes problematic characters such as `' " \ / : | ;`.

When you create passwords for user secrets yourself, be sure to stick to the approved character set to ensure your services run smoothly.

Customize connection to MySQL Router via configurable ports

You can now modify existing ports for the MySQL Router service, as well as add new custom ports. This ability enables you to fine-tune the connection to your Percona Server for MySQL cluster. For example, you can separate access to the database for different applications, so that each one connects to the same MySQL Router but gets a tailored experience based on the port.

Ability to resize the volume with Volume Expansion capability

Previously, users could resize the storage by deleting the cluster and manually adjusting the storage size in custom resources. Now users can leverage the Volume Expansion capability available in Kubernetes and resize their PVCs by just changing the value of the `resources.requests.storage` option in the PerconaServerMySQL custom resource. The Operator handles the resizing flow thus reducing the intervention required from the user.

Deprecation, rename and removal

- `.spec.pmm.runtimeClassName` field has been removed from the `crd.yaml` and code because it wasn't being used
- `.spec.backup.imagePullSecrets` will now be applied to the backup and restore jobs
- `.spec.proxy.haproxy.runtimeClassName` will be applied to the HAProxy Pods
- `.spec.pmm.serverUser` is removed as not used in PMM3

Changelog

New features

- K8SPS-126 - It is now possible to resize Persistent Volume Claims by patching the PerconaServerMySQL Custom Resource. Enable, volume expansion, change `persistentVolumeClaim.resources.requests.storage` and let the Operator do the scaling.
- K8SPS-421 - Added data-at-rest encryption support
- K8SPS-445 - Added MySQL 8.4 support

Improvements

- K8SPS-437 - Removed the `spec.pmm.serverUser` field as not used in PMM 3
- K8SPS-406 - Added possibility of adding custom parameters for PMM client via Custom Resource
- K8SPS-131 - Improve connection configuration by making router ports configurable
- K8SPS-265 - Added special symbols support in passwords
- K8SPS-319 - Improve labels by adding MySQL to the Operator name
- K8SPS-323 - Added support for primary Pod discovery through a Kubernetes Service (Thank you Marjus Cako for reporting this issue)
- K8SPS-336 - Added the ability to deploy the Operator with `hostPath` and `emptyDir` volume types
- K8SPS-357 - Improved cluster provisioning
- K8SPS-401 - Added examples of setting up backups on Azure into our CRs
- K8SPS-418 - Added the ability to specify the time for the Pod to shut down gracefully after receiving a termination signal before it is forcefully killed.
- K8SPS-414 - Added the ability to configure imagePullSecrets via the Custom Resource

- K8SPS-415 - Added the ability to configure runtimeClassName via the Custom Resource
- K8SPS-416 - Added the ability to configure tolerations via the Custom Resource

Bugs Fixed

- K8SPS-287 - Improved logging to include information about `allowUnsafeConfigurations` not set when a user tries to scale down a cluster to less than 3 Pods
- K8SPS-298 - Added an error to the logs about invalid configuration for deploying a cluster with asynchronous replication without a proxy.
- K8SPS-308 - Fixed the issue with smart update reporting errors for the cluster with async replication
- K8SPS-381 - Improved restores from Azure blob storage by removing a hardcoded slash
- K8SPS-394 - Improved the cluster behavior when a user tries to change a replication type on a running cluster. The cluster fails because this operation is not allowed on a running cluster. Added documentation with the recommended steps.
- K8SPS-396 - Improved the gr-self-healing tests by replacing assert with readiness check for chaos-daemon
- K8SPS-425 - Fixed the cluster booostrap process for a group replication clusters with MySQL 8.4

Supported software

The Operator was developed and tested with the following software:

- Percona Server for MySQL 8.4.5-5
- Percona Server for MySQL 8.0.42-33
- XtraBackup 8.4.0-3
- XtraBackup 8.0.35-33
- MySQL Router 8.4.5-5
- MySQL Router 8.0.42
- HAProxy 2.8.15
- Orchestrator 3.2.6-17
- Percona Toolkit 3.7.0

- PMM Client 3.3.1
- Cert Manager 1.18.2

Other options may also work but have not been tested.

Supported platforms

Percona Operators are designed for compatibility with all CNCF-certified Kubernetes distributions. Our release process includes targeted testing and validation on major cloud provider platforms and OpenShift, as detailed below for Operator version 0.11.0:

- Google Kubernetes Engine (GKE) 1.31 - 1.33
- Amazon Elastic Container Service for Kubernetes (EKS) 1.31 - 1.33
- OpenShift 4.15 - 4.19
- Minikube 1.36.0 (based on Kubernetes 1.33.1)

This list only includes the platforms that the Percona Operators are specifically tested on as part of the release process. Other Kubernetes flavors and versions depend on backward compatibility offered by Kubernetes itself.

Percona certified images

Find Percona's certified Docker images that you can use with Percona Operator for MySQL based on Percona Server for MySQL in the following table:

Image	Digest
percona/percona-server-mysql-operator:0.11.0	3068b1a4d81b5da7676e071040d3b44ff9fec5532cbfabb17f9c7612e8c9d35d
percona/percona-server-mysql-operator:0.11.0 (ARM64)	b50f215869ca3d9f6a52561171851e8ffa033ca30d0276556e220ad448418b61
percona/percona-mysql-router:8.4.5	e890ecc49c297cc8882b54ba457ff4d25da896cb11269989fa072aa338d620c1
percona/percona-mysql-router:8.0.42	2364ebb010cc94d7873367e06ad2fadba44f37b81fc78febf3f58a9e61fa5948

percona/percona-orchestrator:3.2.6-17	c1871ddc6ff3eaca7bb03c3aa11db880ae02d623db1203d0858f8566f56ea5f7
percona/percona-toolkit:3.7.0	17ef2b69a97fa546d1f925c74ca09587ac215085c392761bb4d51f188baa6c0e
percona/haproxy:2.8.15	49e6987a1c8b27e9111ae1f1168dd51f2840eb6d939ffc157358f0f259819006
percona/percona-xtrabackup:8.4.0-3	01071522753ad94e11a897859bba4713316d08e493e23555c0094d68da223730
percona/percona-xtrabackup:8.0.35-33	ae56852f0343726409633268cf8c6af92754ea2b5aacbecbec93fa980f8e724d
percona/percona-server:8.4.5-5	9628b1e598c0ec13c2a6b834caa1642bf77f2b4a2d7620b1ba2d8aaf2b708133
percona/percona-server:8.0.42-33	a7cbaa50c43483a07506f7cd5cccc4e587611f6500e1be5df0a93e339b9d3bc5
percona/percona-server:8.0.40-31	09276abecbc7c38ce9c5453da1728f3e7d81722c56e2837574ace3a021ee92f2
percona/percona-server:8.0.36-28	423acd206f94b34288d10ed041c3ba42543e26e44f3706621320504a010dd41f
percona/percona-server:8.0.33-25	14ef81039f2dfa5e19a9bf20e39aaf367aae4370db70899bc5217118d6fd2171
percona/percona-server:8.0.32-24	2107838f98d41172f37c7fc9689095e9ebd0a1af557b687396d92cf00f54ec3f
percona/pmm-client:3.3.1	29a9bb1c69fef8bedc4d4a9ed0ae8224a8623fd3eb8676ef40b13fd044188cb4
percona/pmm-client:3.3.1 (ARM64)	50bccba4cb2c33bd3f6c5e37553bb70345de3f328b23a64ecfa63893f9025c83

Find images for previous versions

Percona Operator for MySQL 0.10.0 (2025-06-04)

Installation

Percona Operator for MySQL allows users to deploy MySQL clusters with both asynchronous and group replication topology. This release includes various stability improvements and bug fixes, getting the Operator closer to the General Availability stage. Version 0.10.0 of the Percona Operator for MySQL is still **a tech preview release**, and it is **not recommended for production environments**.

As of today, we recommend using Percona Operator for MySQL based on Percona XtraDB Cluster, which is production-ready and contains everything you need to quickly and consistently deploy and scale MySQL clusters in a Kubernetes-based environment, on-premises or in the cloud.

Release highlights

PMM3 support

The Operator is natively integrated with PMM 3, enabling you to monitor the health and performance of your Percona Server MySQL deployment and at the same time enjoy enhanced performance, new features, and improved security that PMM 3 provides.

Note that the support for PMM2 is dropped. This means you must do the following to monitor your deployment further:

- transition to PMM 3 if you had PMM 2 to. The PMM documentation explains how to upgrade.
- run the Operator version 0.10.0. Check the Upgrade the Operator tutorial for the update steps.
- ensure that PMM 3 Server version must be equal to or newer than the PMM Client.

Support for deployments on OpenShift

OpenShift is a fully integrated Kubernetes-based platform enhanced with automation, security, and developer-friendly tools. You can now deploy Percona Operator for MySQL based on Percona Server for MySQL on OpenShift and benefit from its portability across hybrid clouds. The Operator also fully supports the Red Hat OpenShift lifecycle which ensures its security and reliability.

Follow our installation guide to install the Operator on OpenShift.

Added labels to identify the version of the Operator

CRD is compatible with the last 3 Operator versions. To know which Operator version is attached to it, we've added lables to all Custom Resource Definitions. The labels help you identify the current Operator version and decide if you need to update the CRD.

To view the labels, run: `kubectl get crd perconaservermysqls.ps.percona.com --show-labels`

Improved configuration validation during cluster deployment

The Operator now enforces these mandatory parameters to have values when it deploys the database cluster:

- `.spec.mysql.size`
- `.spec.proxy.haproxy.size`
- `.spec.proxy.router.size`
- `.spec.orchestrator.size`
- `.spec.backup.pitr.binlogServer.size`

If any of the following configuration options are empty, the deployment fails.

This improved validation ensures that every cluster is deployed with the necessary settings for stability and functionality.

Changelog

New features

- K8SPS-393 - Added support for PMM v3
- K8SPS-110 - Added support for OpenShift

Improvements

- K8SPS-135 - Use MD5 hashing for stored configuration
- K8SPS-320 - Added labels to TLS and user secret objects created by the Operator

- K8SPS-357, K8SPS-423 - Added the `state-monitor` utility to read MySQL state during startup. It is a valuable tool to improve cluster provisioning
- K8SPS-382 - Removed the `loadBalancerIP` Service type as deprecated
- K8SPS-392 - Added the ability to increase timeout for the CLONE operation while bootstrapping a cluster (Thank you Alexander Kuleshov for reporting this issue)
- K8SPS-426 - Added Labels for Custom Resource Definitions (CRD) to identify the Operator version attached to them

Bugs Fixed

- K8SPS-212 - Improved the Custom Resource validation during a cluster deployment when the `.mysql.clusterType` is set to `async`. The validation rules verify that HAProxy and Orchestrator are enabled, while MySQL Router is disabled for async deployments. The corresponding log message is also printed. This helps ensure your cluster is configured according to the requirements for this replication type.
- K8SPS-221 - Fixed a bug with bootstrapping the cluster after crash when the `clusterType` is set to `group-replication`. The fix uses the `state-monitor` utility that checks MySQL state and proceeds with bootstrappins based on the database state.
- K8SPS-299 - Fixed the issue with the Operator failing to initialize the cluster when the size for MySQL is absent in Custom Resource manifest by making the parameters that affect cluster operation mandatory for deployment. If any option has no value, the Operator fails to deploy the cluster
- K8SPS-365 - Fixed the issue with `clusterType` set to `group-replication` failing after the upgrade of the MySQL image. The issue was fixed by removing the excessive restart of the `mysql` container after adding a pod to the cluster.
- K8SPS-375 - Improved the cluster startup process by handling reconciling errors
- K8SPS-379 - Automated the ClusterRole generation when installing the Operator in a cluster-wide mode
- K8SPS-387 - Added the `wait_for_delete` function to ensure that the cluster or its components are fully cleaned up before performing operations like restoration, scaling, or re-deployment.

Deprecation and removal

The `loadBalancerIP` type for the Service objects is deprecated in Kubernetes v1.24+. It is removed from the HAProxy and Router subsections of the `deploy/cr.yaml` Custom Resource manifest. Please refer to Kubernetes documentation for recommendations how to proceed if you have defined this type before.

Supported software

The Operator was developed and tested with the following software:

- Percona Server for MySQL 8.0.42-33
- Orchestrator 3.2.6-17
- MySQL Router 8.0.42
- XtraBackup 8.0.35-33
- Percona Toolkit 3.7.0
- HAProxy 2.8.14
- PMM Client 3.2.0

Other options may also work but have not been tested.

Supported platforms

Percona Operators are designed for compatibility with all CNCF-certified Kubernetes distributions. Our release process includes targeted testing and validation on major cloud provider platforms and OpenShift, as detailed below for Operator version 0.10.0:

- Google Kubernetes Engine (GKE) 1.30 - 1.32
- Amazon Elastic Container Service for Kubernetes (EKS) 1.30 - 1.32
- OpenShift 4.15 - 4.18
- Minikube 1.36.0 (based on Kubernetes 1.33.0)

This list only includes the platforms that the Percona Operators are specifically tested on as part of the release process. Other Kubernetes flavors and versions depend on backward compatibility offered by Kubernetes itself.

Percona certified images