Operator for MongoDB

Documentation

1.22.0 (February 25, 2026)

Table of Contents

1 Percona Operator for MongoDB

The Percona Operator for MongoDB is a Kubernetes-native application that uses custom resources to manage the lifecycle of Percona Server for MongoDB clusters. It works as a controller, monitoring the desired database state defined by you (through YAML files) and ensuring your MongoDB deployment matches that state automatically.

The Operator simplifies and automates tasks related to MongoDB cluster management such as:

- Provisioning and scaling:
    - Automatically creates MongoDB clusters on Kubernetes.
    - Dynamically scales your MongoDB instances up or down based on workload requirements.
- Upgrade: Manages seamless upgrades of MongoDB versions without downtime or data loss.
- Backups and Restores: Simplifies backing up data to external storage (e.g., AWS S3, Azure) and restoring it when needed.
- Self-Healing: Detects and resolves issues such as pod failures, keeping the cluster healthy.
- High Availability: Manages replica sets and failover mechanisms to ensure your database remains available.

What’s new in version 1.22.0

2 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 MongoDB. 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.

2.1 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.

2.2 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.

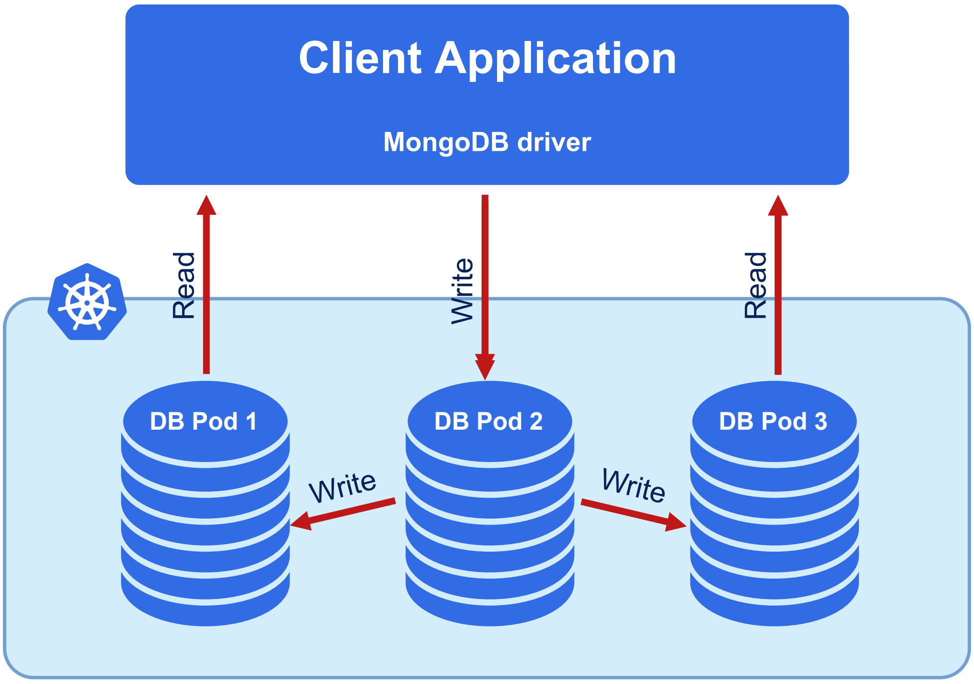
Service	Description
24/7 Expert Support	Our dedicated team of database experts is available 24/7 to assist you with any database issues. We provide flexible support plans tailored to your specific needs.
Hands-On Database Management	Our managed services team can take over the day-to-day management of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

3 Features

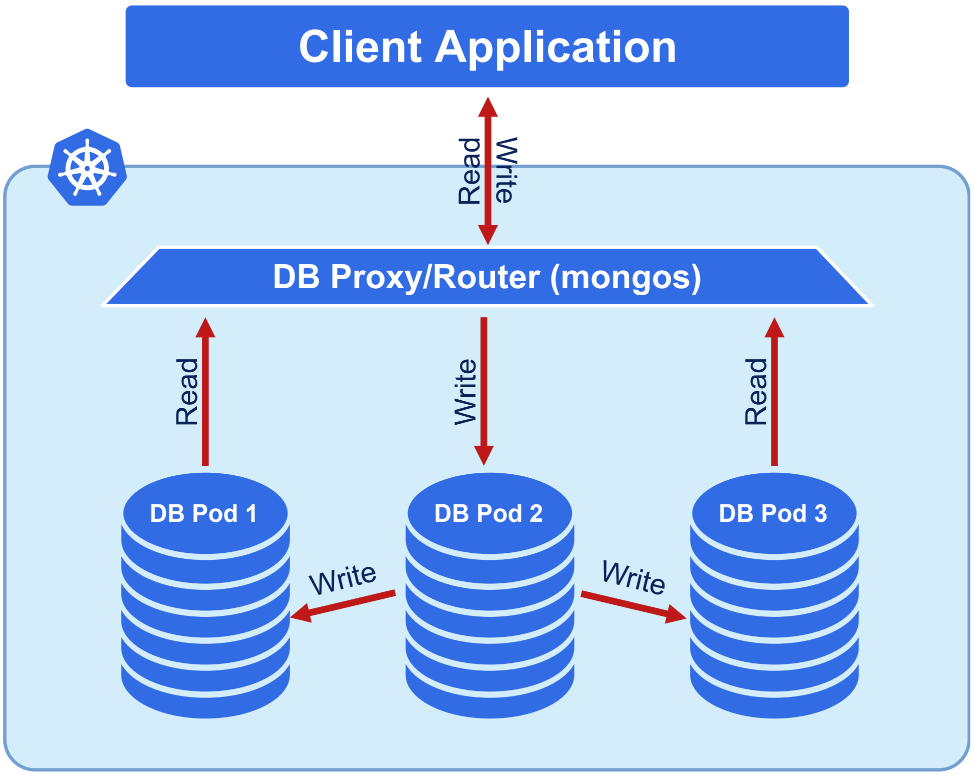
3.1 Design overview

The design of the Operator is tightly bound to the Percona Server for MongoDB replica set or sharded cluster. Replica set cluster is briefly described in the following diagram.



A replica set consists of one primary server and several secondary ones (two in the picture), and the client application accesses the servers via a driver.

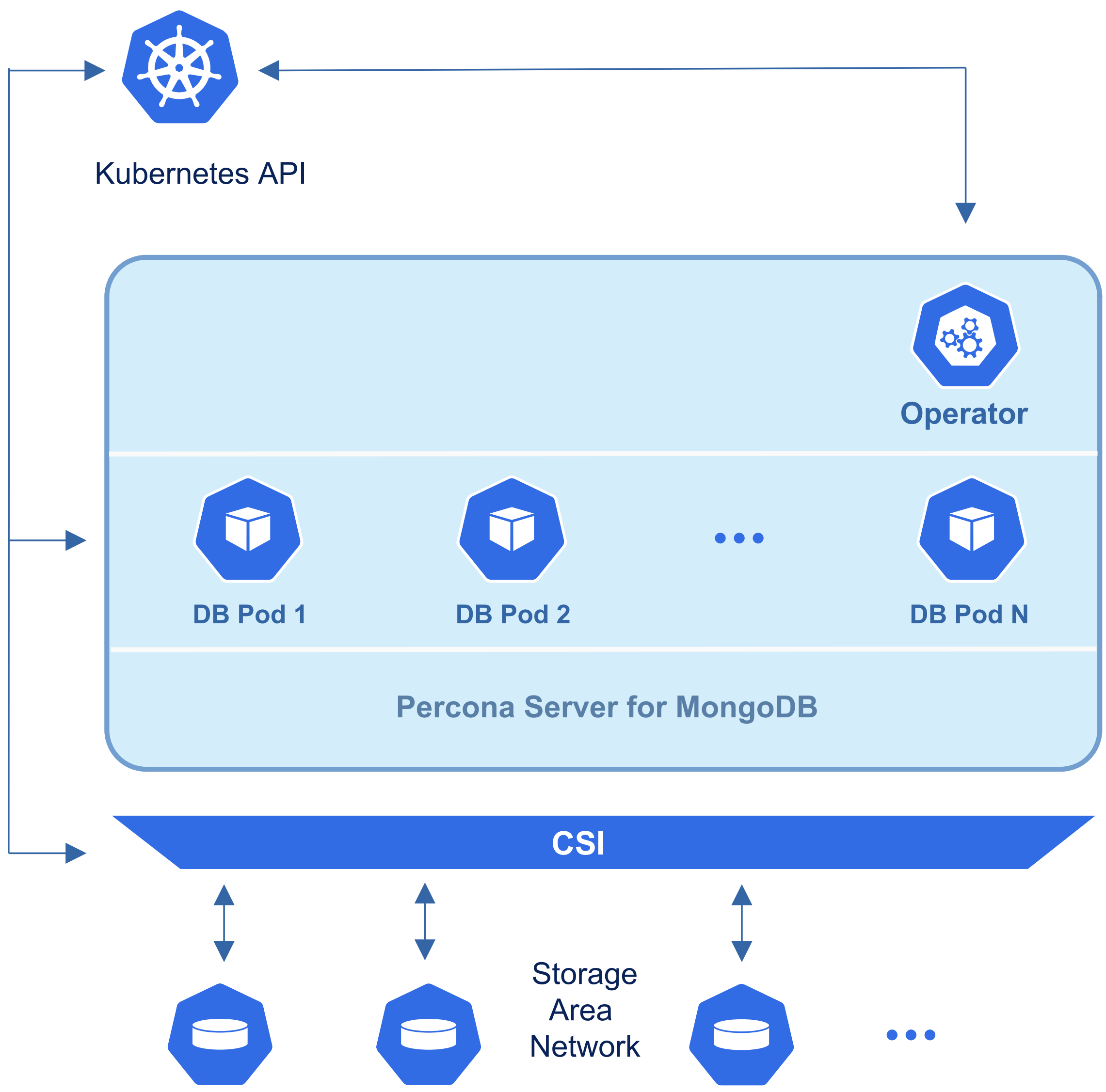
In the case of a sharded cluster, each shard is a replica set which contains a subset of data stored in the database, and the `mongos` query router acts as an entry point for client applications. You can find out more details about sharding [on a dedicated documentation page](), and a simplified diagram is as follows:



To provide high availability the Operator uses [node affinity](#) to run MongoDB instances on separate worker nodes if possible, and the database cluster is deployed as a single Replica Set with at least three nodes. If a node fails, the pod with the mongod process is automatically re-created on another node. If the failed node was hosting the primary server, the replica set initiates elections to select a new primary. If the failed node was running the Operator, Kubernetes will restart the Operator on another node, so normal operation will not be interrupted.

Client applications should use a mongo+srv URI for the connection. This allows the drivers (4.2 and up) to retrieve the list of replica set members from DNS SRV entries without having to list hostnames for the dynamically assigned nodes.

> **Note**
>
> The Operator uses security settings which are more secure than the default Percona Server for MongoDB setup. The initial configuration contains default passwords for all needed user accounts, which should be changed in the production environment, as stated in the [installation instructions](#).



To provide data storage for stateful applications, Kubernetes uses Persistent Volumes. A *PersistentVolumeClaim* (PVC) is used to implement the automatic storage provisioning to pods. If a failure occurs, the Container Storage Interface (CSI) should be able to re-mount storage on a different node. The PVC StorageClass must support this feature (Kubernetes and OpenShift support this in versions 1.9 and 3.9 respectively).

The Operator functionality extends the Kubernetes API with *PerconaServerMongoDB* object, and it is implemented as a golang application. Each *PerconaServerMongoDB* object maps to one separate Percona Server for MongoDB setup. The Operator listens to all events on the created objects. When a new PerconaServerMongoDB object is created, or an existing one undergoes some changes or deletion, the operator automatically creates/changes/deletes all needed Kubernetes objects with the appropriate settings to provide a properly operating replica set.

3.2 Compare various solutions to deploy MongoDB in Kubernetes

There are multiple ways to deploy and manage MongoDB in Kubernetes. Here we will focus on comparing the following open source solutions:

- Bitnami Helm chart
- KubeDB
- MongoDB Community Operator
- Percona Operator for MongoDB

Generic

Here is the review of generic features, such as supported MongoDB versions, open source models and more.

Feature/Product	Percona Operator for MongoDB	Bitnami Helm Chart	KubeDB for MongoDB	MongoDB Community Operator	MongoDB Enterprise Operator
Open source model	Apache 2.0	Apache 2.0	Open core	Open core	Open core
MongoDB versions	MongoDB 5.0, 6.0, 7.0 *	MongoDB 5.0	MongoDB 3.4, 3.6. 4.0, 4.1, 4.2	MongoDB 4.2, 4.4, 5.0, 6.0, 7.0	MongoDB 4.2, 4.4, 5.0, 6.0, 7.0
Kubernetes conformance	Various versions are tested	No guarantee	No guarantee	No guarantee	No guarantee
Cluster-wide mode	Yes	Not an operator	Enterprise only	Yes	Yes
Network exposure	Yes	Yes	No, only through manual config	No	Yes
Web-based GUI	Percona Everest	🚫	kubedb-ui	🚫	Ops Manager

* Percona Operator relies on Percona Server for MongoDB - a free, enhanced, fully compatible MongoDB software alternative for MongoDB Community Server with enterprise-grade features.

Maintenance

Upgrade and scaling are the two most common maintenance tasks that are executed by database administrators and developers.

Feature/Product	Percona Operator for MongoDB	Bitnami Helm Chart	KubeDB for MongoDB	MongoDB Community Operator	MongoDB Enterprise Operator
Operator upgrade	Yes	Helm upgrade	Image change	Yes	Yes
Database upgrade	Automated minor, manual major	No	Manual minor	Manual minor and major	Yes
Compute scaling	Horizontal and vertical	Horizontal and vertical	Enterprise only	Horizontal only	Yes
Storage scaling	Yes	Manual	Enterprise only	No	Yes

MongoDB topologies

The next comparison is focused on replica sets, arbiters, sharding and other node types.

Feature/Product	Percona Operator for MongoDB	Bitnami Helm Chart	KubeDB for MongoDB	MongoDB Community Operator	MongoDB Enterprise Operator
Multi-cluster	Yes	No	No	No	Yes

deployment					
Sharding	Yes	Yes, another chart	Yes	No	Yes
Arbiter	Yes	Yes	Yes	Yes	Yes
Non-voting nodes	Yes	No	No	No	Yes
Hidden nodes	Yes	Yes	Yes	Yes	Yes
Network exposure	Yes	Yes	Manual	No	Yes
Split Horizon	Yes	No	No	Yes	Yes

Backups

Here are the backup and restore capabilities of each solution.

Feature/Product	Percona Operator for MongoDB	Bitnami Helm Chart	KubeDB for MongoDB	MongoDB Community Operator	MongoDB Enterprise Operator
Scheduled backups	Yes	No	Enterprise only	No	Yes
Incremental backups	Yes	No	Enterprise only	No	No
Point-in-time recovery	Yes	No	No	No	Yes
Logical backups	Yes	No	No	No	Yes
Physical backups	Yes	No	No	No	Yes

Monitoring

Monitoring is crucial for any operations team.

Feature/Product	Percona Operator for MongoDB	Bitnami Helm Chart	KubeDB for MongoDB	MongoDB Community Operator	MongoDB Enterprise Operator
Custom exporters	Yes, through sidecars	mongodb-exporter as a sidecar	mongodb-exporter as a sidecar	Integrate with prometheus operator	Integrate with prometheus operator
Percona Monitoring and Management (PMM)	Yes	No	No	No	No

Miscellaneous

Finally, let’s compare various features that are not a good fit for other categories.

Feature/Product	Percona Operator for MongoDB	Bitnami Helm Chart	KubeDB for MongoDB	MongoDB Community Operator	MongoDB Enterprise Operator
Customize MongoDB configuration	Yes	Yes	Yes	No, only some params	No, only some params
Helm	Yes	Yes	Yes, for operator only	Yes, for operator only	Yes, for operator only
SSL/TLS	Yes	Yes	Enterprise only	Yes	Yes

Create users/roles	Yes	Yes	No	Yes	Yes

4 Quickstart guides

4.1 Overview

Ready to get started with the Percona Operator for MongoDB? These tutorials help you learn some basic operations, such as:

- Install and deploy an Operator
- Connect to Percona Server for MongoDB
- 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 →

4.2 1. Quick install

4.2.1 Install Percona Server for MongoDB 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 MongoDB is based on best practices for configuration and setup of a Percona Server for MongoDB and Percona Backup for MongoDB 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 Distribution for MongoDB, you need the following:

1. The **kubectl** tool to manage and deploy applications on Kubernetes, included in most Kubernetes distributions. If you don't have it installed, you can install it by following the 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
> - Create and configure the AKS cluster

What you will install

- The Operator - the custom controller that uses the custom resources to install and manage the lifecycle of your database cluster. It consists of the following components:
  - the Operator Deployment - the controller Pod
  - the CustomResourceDefinitions (CRDs) are a way to add new API types (custom resources) to Kubernetes so that it understands and handles them
  - Role-based access control (RBAC) is the system that controls who can perform which actions on which resources, using roles and bindings to enforce safe, predictable access.
- The database cluster - the actual Percona Server for MongoDB cluster that the Operator creates for you when you apply the Custom Resource or install the Helm chart. It includes StatefulSets for mongod/mongos/config servers, Services, Secrets, and optional components like backups and PMM.

The default Percona Server for MongoDB configuration includes three mongod, three mongos, and three config server instances with enabled sharding.

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 using the following command:

```
kubectl apply --server-side -f https://raw.githubusercontent.com/percona/percona-server-mongodb-operator/v1.22.0/deploy/bundle.yaml -n <namespace>
```

Expected output

```
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbs.psmdb.percona.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbbackups.psmdb.percona.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbrestores.psmdb.percona.com serverside-applied
role.rbac.authorization.k8s.io/percona-server-mongodb-operator serverside-applied
serviceaccount/percona-server-mongodb-operator serverside-applied
rolebinding.rbac.authorization.k8s.io/service-account-percona-server-mongodb-operator serverside-applied
deployment.apps/percona-server-mongodb-operator serverside-applied
```

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

3. Deploy Percona Server for MongoDB:

```
kubectl apply -f https://raw.githubusercontent.com/percona/percona-server-mongodb-operator/v1.22.0/deploy/cr.yaml -n <namespace>
```

Expected output

```
perconaservermongodb.psmdb.percona.com/my-cluster-name created
```

4. Check the Operator and the Percona Server for MongoDB Pods status.

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

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

Expected output

```
NAME              ENDPOINT                                           STATUS   AGE
my-cluster-name   my-cluster-name-mongos.default.svc.cluster.local   ready    5m26s
```

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

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

Next steps

Connect to Percona Server for MongoDB →

Useful links

Install Percona Server for MongoDB with customized parameters

4.2.2 Install Percona Server for MongoDB using Helm

Helm is the package manager for Kubernetes. A Helm chart is a package that contains all the necessary resources to deploy an application to a Kubernetes cluster.

You can find Percona Helm charts in percona/percona-helm-charts repository in GitHub.

Prerequisites

To install and deploy the Operator, you need the following:

1. Helm v3.
2. kubectl command line utility.
3. A Kubernetes environment. You can deploy it locally 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
> - Create and configure the AKS cluster

What you will install

- The Operator - the custom controller that uses the custom resources to install and manage the lifecycle of your database cluster. It consists of the following components:
  - the Operator Deployment - the controller Pod
  - the CustomResourceDefinitions (CRDs) are a way to add new API types (custom resources) to Kubernetes so that it understands and handles them
  - Role-based access control (RBAC) is the system that controls who can perform which actions on which resources, using roles and bindings to enforce safe, predictable access.
- The database cluster - the actual Percona Server for MongoDB cluster that the Operator creates for you when you apply the Custom Resource or install the Helm chart. It includes StatefulSets for mongod/mongos/config servers, Services, Secrets, and optional components like backups and PMM.

The default Percona Server for MongoDB configuration includes three mongod, three mongos, and three config server instances with enabled sharding.

Installation

Here's a sequence of steps to follow:

1. Add the Percona's 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. It is a good practice to isolate workloads in Kubernetes via namespaces. Create a namespace:

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

3. Install the Percona Server for MongoDB Operator Custom Resource Definitions (CRDs) first. You can later use this chart to manage and upgrade CRDs independently using Helm or GitOps tools.

```
helm install psmdb-operator-crds percona/psmdb-operator-crds --namespace <namespace>
```

Important

Although you install the CRD Helm chart (`psmdb-operator-crds`) in a specific namespace, the CRDs themselves are cluster-scoped resources and apply to *all* namespaces in your Kubernetes environment. If you uninstall or delete this CRD chart, it will remove the CRDs from the Kubernetes environment. This will irreversibly delete *all* Percona Server for MongoDB clusters managed by these CRDs across *all* namespaces where you deployed them.

4. Install Percona Operator for MongoDB Deployment. Replace the `namespace` with the name of your namespace:

```
helm install my-op percona/psmdb-operator --namespace <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 (use any name you like).

Expected output

```
NAME: my-op
LAST DEPLOYED: Mon Jan 19 11:45:12 2026
NAMESPACE: <namespace>
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
.....
```

5. Install Percona Server for MongoDB:

```
helm install cluster1 percona/psmdb-db --namespace <namespace>
```

The `cluster1` parameter is the name of [a new release object](#) which is created for the Percona Server for MongoDB when you install its Helm chart (use any name you like).

Expected output

```
NAME: cluster1
LAST DEPLOYED: Mon Jan 19 12:19:37 2026
NAMESPACE: <namespace>
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
....
```

6. Check the Operator and the Percona Server for MongoDB Pods status.

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

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

Expected output

```
NAME              ENDPOINT                                              STATUS   AGE
cluster1-psmdb-db   cluster1-psmdb-db-mongos.<namespace>.svc.cluster.local:27017   ready    5m26s
```

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

You can find in the documentation for the charts which [Operator](#) and [database](#) parameters can be customized during installation. Also, you can check the rest of the Operator's parameters in the [Custom Resource options reference](#).

Next steps

[Connect to Percona Server for MongoDB →](#)

Useful links

[Install Percona Server for MongoDB with customized parameters]

4.3 2. Connect to Percona Server for MongoDB

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

To connect to Percona Server for MongoDB you need to construct the MongoDB connection URI string. It includes the credentials of the admin user, which are stored in the Secrets object.

Here's how to do it:

1. List the Secrets objects

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

The Secrets object we target is named as `<cluster_name>-secrets`. The `<cluster_name>` value is the [name of your Percona Distribution for MongoDB](). The default variant is:

via kubectl

`my-cluster-name-secrets`

via Helm

`cluster1-psmdb-db-secrets`

2. Retrieve the admin user credentials. Replace the `secret-name` and `namespace` with your values in the following commands:

Retrieve the login

```
kubectl get secret <secret-name> -n <namespace> -o yaml -o jsonpath='{.data.MONGODB_DATABASE_ADMIN_USER}' | base64 --
decode | tr '\n' ' ' && echo " "
```

The default value is `databaseAdmin`

Retrieve the password

```
kubectl get secret <secret-name> -n <namespace> -o yaml -o jsonpath='{.data.MONGODB_DATABASE_ADMIN_PASSWORD}' | base64 --
decode | tr '\n' ' ' && echo " "
```

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

```
kubectl -n <namespace> run -i --rm --tty percona-client --image=percona/percona-server-mongodb:8.0.19-7 --restart=Never --
bash -il
```

4. Connect to Percona Server for MongoDB. The format of the MongoDB connection URI string is the following:

sharding is on

```
mongosh "mongodb://<databaseAdminUser>:<databaseAdminPassword>@<cluster-name>-mongos.<namespace>.svc.cluster.local/admin?
ssl=false"
```

sharding is off

```
mongosh  "mongodb://<databaseAdminUser>:<databaseAdminPassword>@<cluster-name>-rs0.<namespace>.svc.cluster.local/admin?
replicaSet=rs0&ssl=false"
```

If you run MongoDB 5.0 and earlier, use the old `mongo` client instead of `mongosh`.

Example

The following example connects to the `admin` database of Percona Server for MongoDB 6.0 sharded cluster with the name `my-cluster-name`. The cluster runs in the namespace `mongodb-operator`:

```
mongosh "mongodb://databaseAdmin:databaseAdminPassword@my-cluster-name-mongos.mongodb-operator.svc.cluster.local/admin?ssl=false"
```

Congratulations! You have connected to Percona Server for MongoDB.

Next steps

Insert sample data →

4.4 3. Insert sample data

In this tutorial you will learn to insert sample data to Percona Server for MongoDB.

MongoDB provides [multiple methods for data insert](). We will use a `For` loop to insert some sample documents.

1. Run the following command:

```
admin> for (var i = 1; i <= 50; i++) {
   db.test.insertOne( { x : i } )
}
```

If there is no `test` collection created, MongoDB creates when inserting documents.

Output

```
{
  acknowledged: true,
  insertedId: ObjectId("652567e5eedca48f97e1868f")
}
```

2. Query the collection to verify the data insertion

```
admin> db.test.find()
```

Output

```
[
  { _id: ObjectId("652567e4eedca48f97e1865e"), x: 1 },
  { _id: ObjectId("652567e4eedca48f97e1865f"), x: 2 },
  { _id: ObjectId("652567e4eedca48f97e18660"), x: 3 },
  { _id: ObjectId("652567e4eedca48f97e18661"), x: 4 },
  { _id: ObjectId("652567e4eedca48f97e18662"), x: 5 },
  { _id: ObjectId("652567e4eedca48f97e18663"), x: 6 },
  { _id: ObjectId("652567e4eedca48f97e18664"), x: 7 },
  { _id: ObjectId("652567e4eedca48f97e18665"), x: 8 },
  { _id: ObjectId("652567e4eedca48f97e18666"), x: 9 },
  { _id: ObjectId("652567e4eedca48f97e18667"), x: 10 },
  { _id: ObjectId("652567e4eedca48f97e18668"), x: 11 },
  { _id: ObjectId("652567e4eedca48f97e18669"), x: 12 },
  { _id: ObjectId("652567e4eedca48f97e1866a"), x: 13 },
  { _id: ObjectId("652567e4eedca48f97e1866b"), x: 14 },
  { _id: ObjectId("652567e4eedca48f97e1866c"), x: 15 },
  { _id: ObjectId("652567e4eedca48f97e1866d"), x: 16 },
  { _id: ObjectId("652567e4eedca48f97e1866e"), x: 17 },
  { _id: ObjectId("652567e4eedca48f97e1866f"), x: 18 },
  { _id: ObjectId("652567e4eedca48f97e18670"), x: 19 },
  { _id: ObjectId("652567e4eedca48f97e18671"), x: 20 }
]
```

You will have different `_id` values.

Now your cluster has some data in it.

Next steps

Make a backup →

4.5 4. 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-mongodb-operator repository:

```
git clone -b v1.22.0 https://github.com/percona/percona-server-mongodb-operator
cd percona-server-mongodb-operator
```

> **Note**
>
> It is crucial 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-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-s3.yaml

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: my-cluster-name-backup-s3
   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-s3.yaml -n <namespace>
   ```

4. Update your `deploy/cr.yaml` configuration. Specify the following parameters in the `backups` 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 (`my-cluster-name-backup-s3` 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:
  ...
  storages:
    s3-us-west:
      type: s3
      s3:
        bucket: "S3-BACKUP-BUCKET-NAME-HERE"
        region: "<AWS_S3_REGION>"
        credentialsSecret: my-cluster-name-backup-s3
  ...
```

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 logical backup

1. Before you start, verify the backup configuration in the `deploy/cr.yaml` file:
   - the `backup.enabled` key is set to `true`
   - the `backup.storages` subsection contains the [configured storage](#).
2. 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 MongoDB.
   - `spec.storageName` - specify the name of your already configured storage.

deploy/backup/backup.yaml

```
apiVersion: psmdb.percona.com/v1
kind: PerconaServerMongoDBBackup
metadata:
  finalizers:
  - percona.com/delete-backup
  name: backup1
spec:
  clusterName: my-cluster-name
  storageName: s3-us-west
  type: logical
```

3. 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>
```

4. Track the backup progress.

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

Sample output

``` {.text .no-copy}

NAME CLUSTER STORAGE DESTINATION TYPE SIZE STATUS COMPLETED AGE backup1 my-cluster-name s3-us-west s3://my-bucket/2025-09-23T10:34:59Z logical 105.44MB ready 43s 43s ```

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

## Troubleshooting

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

- View the information about the backup with the following command:

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

- View the backup-agent logs. Use the previous command to find the name of the pod where the backup was made:

```
kubectl logs pod/<pod-name> -c backup-agent -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 configure point-in-time recovery, enable server-side encryption and how to automatically make backups according to the schedule.

Next steps

Monitor the database →
```

4.6 5. Monitor database with Percona Monitoring and Management (PMM)

In this section you will learn how to monitor the health of Percona Server for MongoDB with Percona Monitoring and Management (PMM).

The Operator supports both PMM version 2 and PMM version 3.

It determines which PMM server version you are using based on the authentication method you provide. For PMM 2, the Operator uses API keys for authentication. For PMM 3, it uses service account tokens.

We recommend using the latest PMM 3.

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 needed metrics and sends gathered data to the PMM 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. If you are using PMM server version 2, use a PMM client image compatible with PMM 2. If you are using PMM server version 3, use a PMM client image compatible with PMM 3. Check Percona certified images for the right one.
2. If you specified both authentication methods for PMM server configuration and they have non-empty values, priority goes to PMM 3.
3. For migration from PMM2 to PMM3, see PMM upgrade documentation. Also check the Automatic migration of API keys page.

Install PMM Server

You must have PMM server up and running. You can run PMM Server as a *Docker image*, a *virtual appliance*, or in Kubernetes. Please refer to the official PMM documentation for the installation instructions.

Install PMM Client

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

Configure authentication

PMM3

PMM3 uses service accounts to control access to PMM server components and resources. To authenticate in PMM server, you need a service account token. [Generate a service account and token](). Specify the Admin role for the service account.

> **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.

PMM2

[Get the PMM API key from PMM Server](). The API key must have the role "Admin". You need this key to authorize PMM Client within PMM Server.

From PMM UI

[Generate the PMM API key]()

From command line

You can query your PMM Server installation for the API Key using `curl` and `jq` utilities. Replace `<login>:<password>@<server_host>` placeholders with your real PMM Server login, password, and hostname in the following command:

```
PMM_SERVER_API_KEY=$(curl --insecure -X POST -H "Content-Type: application/json" -d '{"name":"operator", "role": "Admin"}'
"https://<login>:<password>@<server_host>/graph/api/auth/keys" | jq .key)
```

> **Warning**
>
> The API key is not rotated automatically when it expires. You must manually recreate it and reconfigure the PMM Client.

Create a secret

Now you must pass the credentials to the Operator. To do so, create a Secret object.

1. Create a Secret configuration file. You can use the [deploy/secrets.yaml]() secrets file.

PMM 3

Specify the service account token as the `PMM_SERVER_TOKEN` value in the secrets file:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-cluster-name-secrets
type: Opaque
stringData:
  ....
  PMM_SERVER_TOKEN: ""
```

PMM 2

Specify the API key as the `PMM_SERVER_API_KEY` value in the secrets file:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-cluster-name-secrets
type: Opaque
stringData:
  ....
  PMM_SERVER_API_KEY: ""
```

2. Create the Secrets object using the `deploy/secrets.yaml` file.

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

Expected output

```
secret/my-cluster-name-secrets created
```

Deploy the PMM Client

1. Update the `pmm` section in the deploy/cr.yaml file:
   - Set `pmm.enabled`=`true`.
   - Specify your PMM Server hostname / an IP address for the `pmm.serverHost` option. The PMM Server IP address should be resolvable and reachable from within your cluster.
   - Check that the name of the Secret object that you created earlier is specified in the `secrets.users` field.

```
secrets:
  users: my-cluster-name-secrets
pmm:
  enabled: true
  image: percona/pmm-client:2.44.1-1
  serverHost: monitoring-service
```

2. Apply the changes:

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

3. 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>-rs0-0 -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 **MongoDB** from the left-hand navigation menu. You land on the **Instances Overview** page.
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 **MongoDB** → **Other dashboards** to see the list of available dashboards that allow you to drill down to the metrics you are interested in.

Next steps

What’s next →

4.7 What's next?

Congratulations! You have completed all the steps in the Get started guide.

You have the following options to move forward with the Operator:

- Deepen your monitoring insights by setting up Kubernetes monitoring with PMM
- Control Pods assignment on specific Kubernetes Nodes by setting up affinity / anti-affinity
- Ready to adopt the Operator for production use and need to delete the testing deployment? Use this guide to do it
- You can also try operating the Operator and database clusters via the web interface with Percona Everest - an open-source web-based database provisioning tool based on Percona Operators. See Get started with Percona Everest on how to start using it

5 Installation

5.1 System Requirements

The Operator was developed and tested with Percona Server for MongoDB 6.0.27-21, 7.0.30-16, and 8.0.19-7. Other options may also work but have not been tested. The Operator 1.22.0 also uses Percona Backup for MongoDB 2.12.0.

Officially supported platforms

The following platforms were tested and are officially supported by the Operator 1.22.0:

- Google Kubernetes Engine (GKE) 1.32 - 1.33
- Amazon Elastic Kubernetes Service (EKS) 1.32 - 1.35
- Azure Kubernetes Service (AKS) 1.32 - 1.34
- OpenShift Container Platform 4.17 - 4.21
- Minikube 1.38.0 based on Kubernetes v1.35.0

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

Resource Limits

A cluster running an officially supported platform contains at least 3 Nodes and the following resources (if sharding is turned off):

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

Consider using 4 CPU and 6 GB of RAM if sharding is turned on (the default behavior).

Also, the number of Replica Set Nodes should not be odd if Arbiter is not enabled.

Note

Use Storage Class with XFS as the default filesystem if possible to achieve better MongoDB performance .

Installation guidelines

Choose how you wish to install the Operator:

- with Helm
- with `kubectl`
- on Minikube
- on Google Kubernetes Engine (GKE)
- on Amazon Elastic Kubernetes Service (AWS EKS)
- on Microsoft Azure Kubernetes Service (AKS)
- on Openshift
- in a Kubernetes-based environment

5.2 Install Percona Server for MongoDB on Minikube

Installing the Percona Operator for MongoDB on Minikube is the easiest way to try it locally without a cloud provider. Minikube runs Kubernetes on GNU/Linux, Windows, or macOS system using a system-wide hypervisor, such as VirtualBox, KVM/QEMU, VMware Fusion or Hyper-V. Using it is a popular way to test Kubernetes application locally prior to deploying it on a cloud.

The following steps are needed to run Percona Operator for MongoDB on minikube:

1. Install minikube, using a way recommended for your system. This includes the installation of the following three components:

   a. kubectl tool,

   b. a hypervisor, if it is not already installed,

   c. actual minikube package

   After the installation, run `minikube start --memory=5120 --cpus=4 --disk-size=30g` (parameters increase the virtual machine limits for the CPU cores, memory, and disk, to ensure stable work of the Operator). Being executed, this command will download needed virtualized images, then initialize and run the cluster. After Minikube is successfully started, you can optionally run the Kubernetes dashboard, which visually represents the state of your cluster. Executing `minikube dashboard` will start the dashboard and open it in your default web browser.

2. Deploy the operator using the following command:

```
kubectl apply --server-side -f https://raw.githubusercontent.com/percona/percona-server-mongodb-operator/v1.22.0/deploy/bundle.yaml
```

3. Deploy MongoDB cluster with:

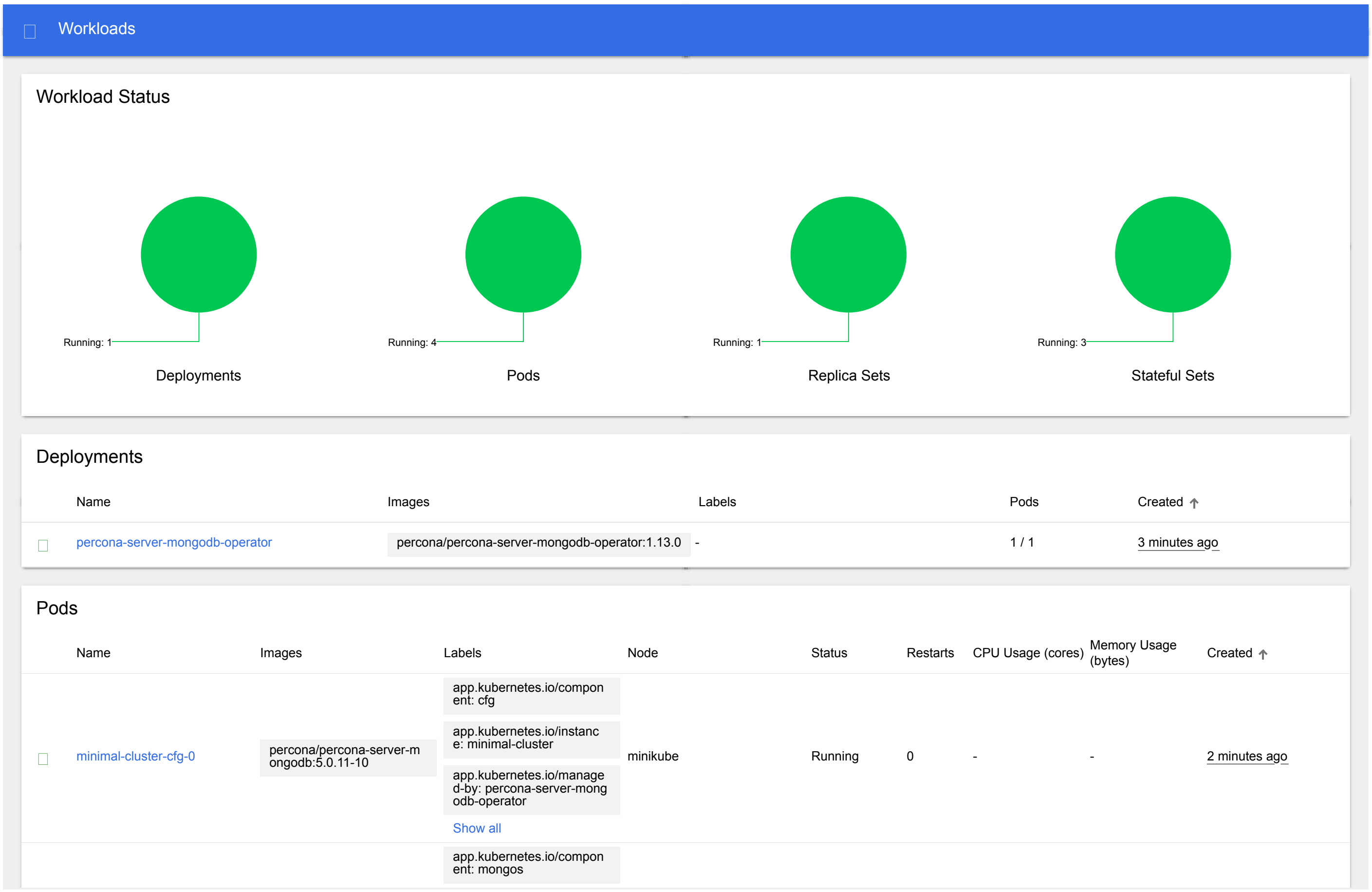
```
kubectl apply -f https://raw.githubusercontent.com/percona/percona-server-mongodb-operator/v1.22.0/deploy/cr-minimal.yaml
```

> **Note**
>
> This deploys a one-shard MongoDB cluster with one replica set with one node, one mongos node and one config server node. The deploy/cr-minimal.yaml is for minimal non-production deployment. For more configuration options please see deploy/cr.yaml and Custom Resource Options. You can clone the repository with all manifests and source code by executing the following command:
>
> ```
> git clone -b v1.22.0 https://github.com/percona/percona-server-mongodb-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.

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

   You can also track the progress via the Kubernetes dashboard:



Verifying the cluster operation

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

To connect to Percona Server for MongoDB you need to construct the MongoDB connection URI string. It includes the credentials of the admin user, which are stored in the Secrets object.

1. List the Secrets objects

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

   The Secrets object you are interested in has the `minimal-cluster-secrets` name by default.

2. View the Secret contents to retrieve the admin user credentials.

```
kubectl get secret minimal-cluster-secrets -o yaml
```

   The command returns the YAML file with generated Secrets, including the `MONGODB_DATABASE_ADMIN_USER` and `MONGODB_DATABASE_ADMIN_PASSWORD` strings, which should look as follows:

   Sample output

```
...
data:
  ...
  MONGODB_DATABASE_ADMIN_PASSWORD: aDAzQ0pCY3NSWEZ2ZUIzS1I=
  MONGODB_DATABASE_ADMIN_USER: ZGF0YWJhc2VBZG1pbg==
```

   The actual login name and password on the output are base64-encoded. To bring it back to a human-readable form, run:

```
echo 'MONGODB_DATABASE_ADMIN_USER' | base64 --decode
echo 'MONGODB_DATABASE_ADMIN_PASSWORD' | base64 --decode
```

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

```
kubectl run -i --rm --tty percona-client --image=percona/percona-server-mongodb:7.0.30-16 --restart=Never -- bash -il
```

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

4. Now run `mongosh` tool inside the `percona-client` command shell using the admin user credentialds you obtained from the Secret, and a proper namespace name instead of the `<namespace name>` placeholder. The command will look different depending on whether sharding is on (the default behavior) or off:

if sharding is on

```
mongosh "mongodb://databaseAdmin:databaseAdminPassword@minimal-cluster-mongos.<namespace name>.svc.cluster.local/admin?ssl=false"
```

if sharding is off

```
mongosh "mongodb+srv://databaseAdmin:databaseAdminPassword@minimal-cluster-rs0.<namespace name>.svc.cluster.local/admin?replicaSet=rs0&ssl=false"
```

5.3 Install Percona Server for MongoDB cluster using Everest

Percona Everest is an open source cloud-native database platform that helps developers deploy code faster, scale deployments rapidly, and reduce database administration overhead while regaining control over their data, database configuration, and DBaaS costs.

It automates day-one and day-two database operations for open source databases on Kubernetes clusters. Percona Everest provides API and Web GUI to launch databases with just a few clicks and scale them, do routine maintenance tasks, such as software updates, patch management, backups, and monitoring.

You can try it in action by Installing Percona Everest and managing your first cluster.

5.4 Install Percona Server for MongoDB on Google Kubernetes Engine (GKE)

This guide shows you how to deploy Percona Operator for MongoDB 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
```

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 `my-cluster-name`:

```
gcloud container clusters create my-cluster-name \
  --project <project ID> \
  --zone us-central1-a \
  --cluster-version 1.33 \
  --machine-type n1-standard-4 \
  --num-nodes=3
```

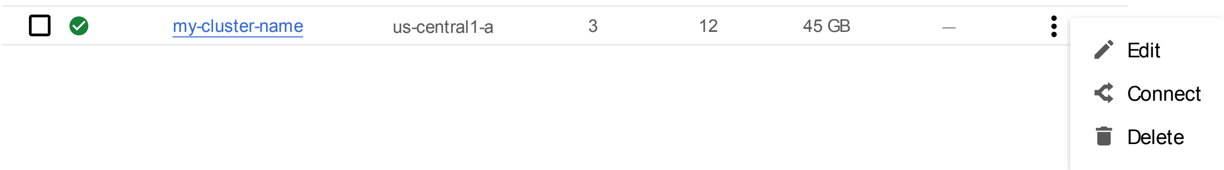
Note

You must edit the following 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` in the above example. Other parameters specify that we are creating a cluster with 3 nodes and with machine type of `n1-standard-4` with x86_64 vCPUs. If you need ARM64, use a different `--machine-type`, for example, `t2a-standard-4`.

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 my-cluster-name \
  --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 MongoDB 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 (replace the `<namespace name>` placeholder with some descriptive name):

   ```
   kubectl create namespace <namespace name>
   ```

   and/or set the context for the namespace as follows:

   ```
   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 by applying the `deploy/bundle.yaml` manifest from the Operator source tree.

For x86_64 architecture

You can apply it without downloading, using the following command:

```
kubectl apply --server-side -f https://raw.githubusercontent.com/percona/percona-server-mongodb-operator/v1.22.0/deploy/bundle.yaml
```

Expected output

```
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbs.psmdb.percona.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbbackups.psmdb.percona.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbrestores.psmdb.percona.com serverside-applied
role.rbac.authorization.k8s.io/percona-server-mongodb-operator serverside-applied
serviceaccount/percona-server-mongodb-operator serverside-applied
rolebinding.rbac.authorization.k8s.io/service-account-percona-server-mongodb-operator serverside-applied
deployment.apps/percona-server-mongodb-operator serverside-applied
```

For ARM64 architecture

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

```
git clone -b v1.22.0 https://github.com/percona/percona-server-mongodb-operator
```

Edit the `deploy/bundle.yaml` file: add the following affinity rules to the `spec` part of the `percona-server-mongodb-operator` Deployment:

```
    apiVersion: apps/v1
    kind: Deployment
    metadata:
      name: percona-server-mongodb-operator
    spec:
      replicas: 1
      selector:
        matchLabels:
          name: percona-server-mongodb-operator
      template:
        metadata:
          labels:
            name: percona-server-mongodb-operator
        spec:
          affinity:
            nodeAffinity:
              requiredDuringSchedulingIgnoredDuringExecution:
                nodeSelectorTerms:
                  - matchExpressions:
                    - key: kubernetes.io/arch
                      operator: In
                      values:
                        - arm64
```

After editing, apply your modified `deploy/bundle.yaml` file as follows:

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

Expected output

```
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbs.psmdb.percona.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbbackups.psmdb.percona.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbrestores.psmdb.percona.com serverside-applied
role.rbac.authorization.k8s.io/percona-server-mongodb-operator serverside-applied
serviceaccount/percona-server-mongodb-operator serverside-applied
rolebinding.rbac.authorization.k8s.io/service-account-percona-server-mongodb-operator serverside-applied
deployment.apps/percona-server-mongodb-operator serverside-applied
```

2. The Operator has been started, and you can deploy your MongoDB cluster:

For x86_64 architecture

```
kubectl apply -f https://raw.githubusercontent.com/percona/percona-server-mongodb-operator/v1.22.0/deploy/cr.yaml
```

Expected output

```
perconaservermongodb.psmdb.percona.com/my-cluster-name created
```

Note

This deploys default MongoDB cluster configuration, three mongod, three mongos, and three config server instances. 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 v1.22.0 https://github.com/percona/percona-server-mongodb-operator
```

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

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

For ARM64 architecture

Edit the `deploy/cr.yaml` file: set the following affinity rules in **all `affinity` subsections**:

```
....
affinity:
  advanced:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/arch
            operator: In
            values:
            - arm64
```

Also, set `image` and `backup.image` Custom Resource options to special multi-architecture image versions by adding a `-multi` suffix to their tags:

```
....
image: percona/percona-server-mongodb:7.0.30-16-multi
...
backup:
  ...
  image: percona/percona-backup-mongodb:2.12.0-multi
```

Please note, that currently monitoring with PMM is not supported on ARM64 configurations.

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

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

Expected output

```
perconaservermongodb.psmdb.percona.com/my-cluster-name created
```

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 psmdb
```

Expected output

```
NAME              ENDPOINT                                         STATUS    AGE
my-cluster-name   my-cluster-name-mongos.default.svc.cluster.local ready     5m26s
```

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		
my-cluster-name-cfg-0	Running	Pod	default	my-cluster-name
my-cluster-name-cfg-1	Running	Pod	default	my-cluster-name
my-cluster-name-cfg-2	Running	Pod	default	my-cluster-name
my-cluster-name-mongos-0	Running	Pod	default	my-cluster-name
my-cluster-name-mongos-1	Running	Pod	default	my-cluster-name
my-cluster-name-mongos-2	Running	Pod	default	my-cluster-name
my-cluster-name-rs0-0	Running	Pod	default	my-cluster-name
my-cluster-name-rs0-1	Running	Pod	default	my-cluster-name
my-cluster-name-rs0-2	Running	Pod	default	my-cluster-name
percona-server-mongodb-operator-665cd69f9b-xg5dl	Running	Pod	default	my-cluster-name

Verifying the cluster operation

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

To connect to Percona Server for MongoDB you need to construct the MongoDB connection URI string. It includes the credentials of the admin user, which are stored in the Secrets object.

1. List the Secrets objects

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

   The Secrets object you are interested in has the `my-cluster-name-secrets` name by default.

2. View the Secret contents to retrieve the admin user credentials.

```
kubectl get secret my-cluster-name-secrets -o yaml
```

   The command returns the YAML file with generated Secrets, including the `MONGODB_DATABASE_ADMIN_USER` and `MONGODB_DATABASE_ADMIN_PASSWORD` strings, which should look as follows:

   Sample output

```
...
data:
  ...
  MONGODB_DATABASE_ADMIN_PASSWORD: aDAzQ0pCY3NSWEZ2ZUIzS1I=
  MONGODB_DATABASE_ADMIN_USER: ZGF0YWJhc2VBZG1pbg==
```

   The actual login name and password on the output are base64-encoded. To bring it back to a human-readable form, run:

```
echo 'MONGODB_DATABASE_ADMIN_USER' | base64 --decode
echo 'MONGODB_DATABASE_ADMIN_PASSWORD' | base64 --decode
```

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

   ```
   kubectl run -i --rm --tty percona-client --image=percona/percona-server-mongodb:7.0.30-16 --restart=Never -- bash -il
   ```

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

4. Now run `mongosh` tool inside the `percona-client` command shell using the admin user credentialds you obtained from the Secret, and a proper namespace name instead of the `<namespace name>` placeholder. The command will look different depending on whether sharding is on (the default behavior) or off:

   if sharding is on

   ```
   mongosh "mongodb://databaseAdmin:databaseAdminPassword@my-cluster-name-mongos.<namespace name>.svc.cluster.local/admin?ssl=false"
   ```

   if sharding is off

   ```
   mongosh "mongodb+srv://databaseAdmin:databaseAdminPassword@my-cluster-name-rs0.<namespace name>.svc.cluster.local/admin?replicaSet=rs0&ssl=false"
   ```

Troubleshooting

If `kubectl get psmdb` 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
my-cluster-name-cfg-0                              2/2     Running   0          11m
my-cluster-name-cfg-1                              2/2     Running   1          10m
my-cluster-name-cfg-2                              2/2     Running   1          9m
my-cluster-name-mongos-0                           1/1     Running   0          11m
my-cluster-name-mongos-1                           1/1     Running   0          11m
my-cluster-name-mongos-2                           1/1     Running   0          11m
my-cluster-name-rs0-0                              2/2     Running   0          11m
my-cluster-name-rs0-1                              2/2     Running   0          10m
my-cluster-name-rs0-2                              2/2     Running   0          9m
percona-server-mongodb-operator-665cd69f9b-xg5dl   1/1     Running   0          37m
```

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

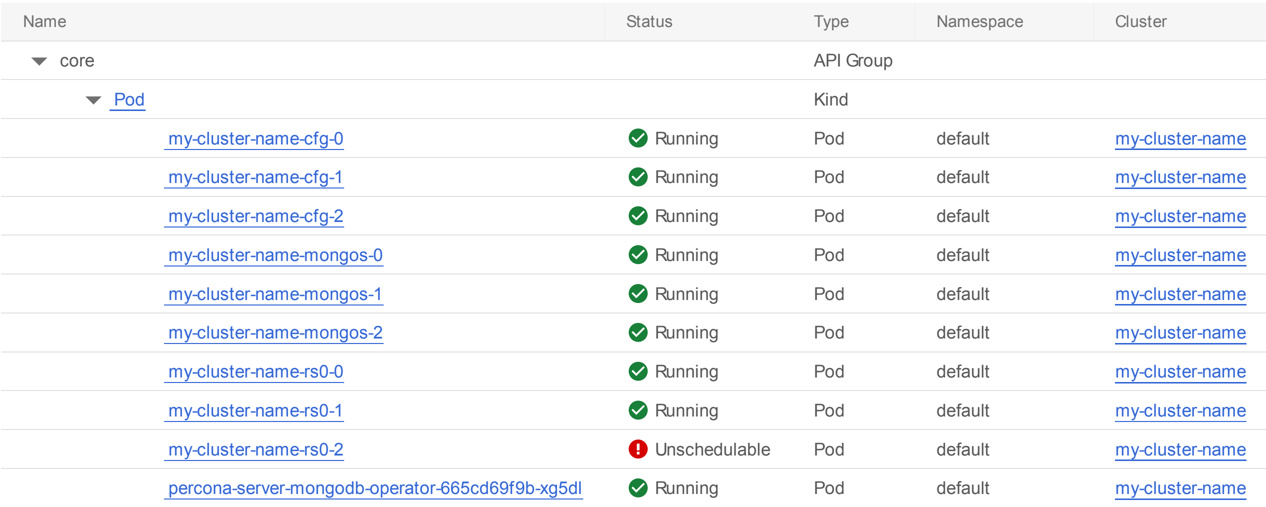
```
kubectl describe pod my-cluster-name-rs0-2
```

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.
```

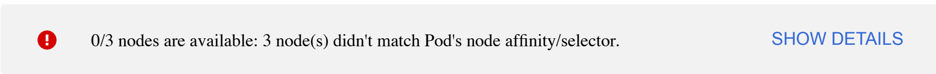
Alternatively, you can examine your Pods via the object browser

The errors will look as follows:



Name	Status	Type	Namespace	Cluster
core		API Group		
Pod		Kind		
my-cluster-name-cfg-0	Running	Pod	default	my-cluster-name
my-cluster-name-cfg-1	Running	Pod	default	my-cluster-name
my-cluster-name-cfg-2	Running	Pod	default	my-cluster-name
my-cluster-name-mongos-0	Running	Pod	default	my-cluster-name
my-cluster-name-mongos-1	Running	Pod	default	my-cluster-name
my-cluster-name-mongos-2	Running	Pod	default	my-cluster-name
my-cluster-name-rs0-0	Running	Pod	default	my-cluster-name
my-cluster-name-rs0-1	Running	Pod	default	my-cluster-name
my-cluster-name-rs0-2	Unschedulable	Pod	default	my-cluster-name
percona-server-mongodb-operator-665cd69f9b-xg5dl	Running	Pod	default	my-cluster-name

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.

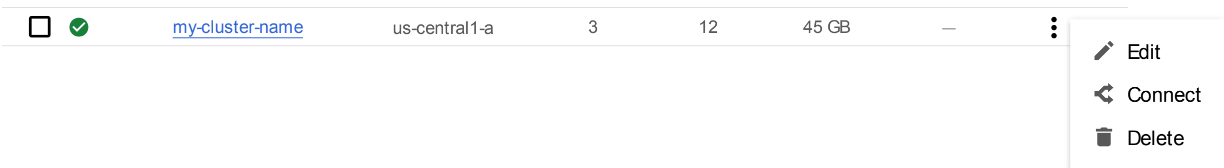
You can clean up the cluster with the `gcloud` command as follows:

```
gcloud container clusters delete <cluster name> \
  --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!

5.5 Install Percona Server for MongoDB on Amazon Elastic Kubernetes Service (EKS)

This guide shows you how to deploy Percona Operator for MongoDB on Amazon Elastic Kubernetes Service (EKS). The document assumes some experience with the platform. 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.

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 tho 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 MongoDB 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 was modified.

   Deploy the Operator by applying the `deploy/bundle.yaml` manifest from the Operator source tree.

For x86_64 architecture

You can apply it without downloading, using the following command:

```
kubectl apply --server-side -f https://raw.githubusercontent.com/percona/percona-server-mongodb-operator/v1.22.0/deploy/bundle.yaml
```

Expected output

```
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbs.psmdb.percona.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbbackups.psmdb.percona.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbrestores.psmdb.percona.com serverside-applied
role.rbac.authorization.k8s.io/percona-server-mongodb-operator serverside-applied
serviceaccount/percona-server-mongodb-operator serverside-applied
rolebinding.rbac.authorization.k8s.io/service-account-percona-server-mongodb-operator serverside-applied
deployment.apps/percona-server-mongodb-operator serverside-applied
```

For ARM64 architecture

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

```
git clone -b v1.22.0 https://github.com/percona/percona-server-mongodb-operator
```

Edit the `deploy/bundle.yaml` file: add the following affinity rules to the `spec` part of the `percona-server-mongodb-operator` Deployment:

```
    apiVersion: apps/v1
    kind: Deployment
    metadata:
      name: percona-server-mongodb-operator
    spec:
      replicas: 1
      selector:
        matchLabels:
          name: percona-server-mongodb-operator
      template:
        metadata:
          labels:
            name: percona-server-mongodb-operator
        spec:
          affinity:
            nodeAffinity:
              requiredDuringSchedulingIgnoredDuringExecution:
                nodeSelectorTerms:
                  - matchExpressions:
                    - key: kubernetes.io/arch
                      operator: In
                      values:
                        - arm64
```

After editing, apply your modified `deploy/bundle.yaml` file as follows:

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

Expected output

```
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbs.psmdb.percona.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbbackups.psmdb.percona.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbrestores.psmdb.percona.com serverside-applied
role.rbac.authorization.k8s.io/percona-server-mongodb-operator serverside-applied
serviceaccount/percona-server-mongodb-operator serverside-applied
rolebinding.rbac.authorization.k8s.io/service-account-percona-server-mongodb-operator serverside-applied
deployment.apps/percona-server-mongodb-operator serverside-applied
```

2. The Operator has been started, and you can deploy your MongoDB cluster:

For x86_64 architecture

```
kubectl apply -f https://raw.githubusercontent.com/percona/percona-server-mongodb-operator/v1.22.0/deploy/cr.yaml
```

Expected output

```
perconaservermongodb.psmdb.percona.com/my-cluster-name created
```

Note

This deploys default MongoDB cluster configuration, three mongod, three mongos, and three config server instances. 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 v1.22.0 https://github.com/percona/percona-server-mongodb-operator
```

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

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

For ARM64 architecture

Edit the `deploy/cr.yaml` file: set the following affinity rules in **all `affinity` subsections**:

```
....
affinity:
  advanced:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/arch
            operator: In
            values:
            - arm64
```

Also, set `image` and `backup.image` Custom Resource options to special multi-architecture image versions by adding a `-multi` suffix to their tags:

```
...
image: percona/percona-server-mongodb:7.0.30-16-multi
...
backup:
  ...
  image: percona/percona-backup-mongodb:2.12.0-multi
```

Please note, that currently monitoring with PMM is not supported on ARM64 configurations.

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

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

Expected output

```
perconaservermongodb.psmdb.percona.com/my-cluster-name created
```

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 psmdb
```

Expected output

```
NAME            ENDPOINT                                        STATUS   AGE
my-cluster-name my-cluster-name-mongos.default.svc.cluster.local ready    5m26s
```

Verifying the cluster operation

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

To connect to Percona Server for MongoDB you need to construct the MongoDB connection URI string. It includes the credentials of the admin user, which are stored in the [Secrets](#) object.

1. List the Secrets objects

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

   The Secrets object you are interested in has the `my-cluster-name-secrets` name by default.

2. View the Secret contents to retrieve the admin user credentials.

   ```
   kubectl get secret my-cluster-name-secrets -o yaml
   ```

   The command returns the YAML file with generated Secrets, including the `MONGODB_DATABASE_ADMIN_USER` and `MONGODB_DATABASE_ADMIN_PASSWORD` strings, which should look as follows:

   Sample output

   ```
   ...
   data:
     ...
     MONGODB_DATABASE_ADMIN_PASSWORD: aDAzQ0pCY3NSWEZ2ZUIzS1I=
     MONGODB_DATABASE_ADMIN_USER: ZGF0YWJhc2VBZG1pbg==
   ```

   The actual login name and password on the output are base64-encoded. To bring it back to a human-readable form, run:

   ```
   echo 'MONGODB_DATABASE_ADMIN_USER' | base64 --decode
   echo 'MONGODB_DATABASE_ADMIN_PASSWORD' | base64 --decode
   ```

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

   ```
   kubectl run -i --rm --tty percona-client --image=percona/percona-server-mongodb:7.0.30-16 --restart=Never -- bash -il
   ```

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

4. Now run `mongosh` tool inside the `percona-client` command shell using the admin user credentialds you obtained from the Secret, and a proper namespace name instead of the `<namespace name>` placeholder. The command will look different depending on whether sharding is on (the default behavior) or off:

   if sharding is on

   ```
   mongosh "mongodb://databaseAdmin:databaseAdminPassword@my-cluster-name-mongos.<namespace name>.svc.cluster.local/admin?ssl=false"
   ```

   if sharding is off

   ```
   mongosh "mongodb+srv://databaseAdmin:databaseAdminPassword@my-cluster-name-rs0.<namespace name>.svc.cluster.local/admin?replicaSet=rs0&ssl=false"
   ```

Troubleshooting

If `kubectl get psmdb` 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
my-cluster-name-cfg-0                               2/2     Running   0          11m
my-cluster-name-cfg-1                               2/2     Running   1          10m
my-cluster-name-cfg-2                               2/2     Running   1          9m
my-cluster-name-mongos-0                            1/1     Running   0          11m
my-cluster-name-mongos-1                            1/1     Running   0          11m
my-cluster-name-mongos-2                            1/1     Running   0          11m
my-cluster-name-rs0-0                               2/2     Running   0          11m
my-cluster-name-rs0-1                               2/2     Running   0          10m
my-cluster-name-rs0-2                               2/2     Running   0          9m
percona-server-mongodb-operator-665cd69f9b-xg5dl    1/1     Running   0          37m
```

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 my-cluster-name-rs0-2
```

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.`

Removing the EKS cluster

To delete your cluster, you will need the following data:

- name of your EKS cluster,
- AWS region in which you have deployed your cluster.

You can clean up the cluster with the `eksctl` command as follows (with real names instead of `<region>` and `<cluster name>` placeholders):

```
eksctl delete cluster --region=<region> --name="<cluster name>"
```

The cluster deletion may take time.

Warning

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

5.6 Install Percona Server for MongoDB on Azure Kubernetes Service (AKS)

This guide shows you how to deploy Percona Operator for MongoDB on Microsoft Azure Kubernetes Service (AKS). The document assumes some experience with the platform. For more information on the AKS, see the Microsoft AKS official documentation.

Prerequisites

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

1. **Azure Command Line Interface (Azure CLI)** for interacting with the different parts of AKS. You can install it following the official installation instructions for your system.
2. **kubectl** to manage and deploy applications on Kubernetes. Install it following the official installation instructions.

Also, you need to sign in with Azure CLI using your credentials according to the official guide.

Create and configure the AKS cluster

To create your cluster, you will need the following data:

- name of your AKS cluster,
- an Azure resource group, in which resources of your cluster will be deployed and managed.
- the amount of nodes you would like tho have.

You can create your cluster via command line using `az aks create` command. The following command will create a 3-node cluster named `my-cluster-name` within some already existing resource group named `my-resource-group`:

```
az aks create --resource-group my-resource-group --name my-cluster-name --enable-managed-identity --node-count 3 --node-vm-size Standard_B4ms --node-osdisk-size 30 --network-plugin kubenet  --generate-ssh-keys --outbound-type loadbalancer
```

Other parameters in the above example specify that we are creating a cluster with x86_64 machine type of Standard_B4ms and OS disk size reduced to 30 GiB. If you need ARM64, use different machine type, for example, Standard_D4ps_v5. You can see detailed information about cluster creation options in the AKS official documentation.

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

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

```
az aks get-credentials --resource-group my-resource-group --name my-cluster-name
```

Install the Operator and deploy your MongoDB 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 (`<cluster name>`) was modified.

   Deploy the Operator, using the following command:

   ```
   kubectl apply --server-side -f https://raw.githubusercontent.com/percona/percona-server-mongodb-operator/v1.22.0/deploy/bundle.yaml
   ```

Expected output

```
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbs.psmdb.percona.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbbackups.psmdb.percona.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbrestores.psmdb.percona.com serverside-applied
role.rbac.authorization.k8s.io/percona-server-mongodb-operator serverside-applied
serviceaccount/percona-server-mongodb-operator serverside-applied
rolebinding.rbac.authorization.k8s.io/service-account-percona-server-mongodb-operator serverside-applied
deployment.apps/percona-server-mongodb-operator serverside-applied
```

2. The Operator has been started, and you can deploy your MongoDB cluster:

For x86_64 architecture

```
kubectl apply -f https://raw.githubusercontent.com/percona/percona-server-mongodb-operator/v1.22.0/deploy/cr.yaml
```

Expected output

```
perconaservermongodb.psmdb.percona.com/my-cluster-name created
```

Note

This deploys default MongoDB cluster configuration, three mongod, three mongos, and three config server instances. 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 v1.22.0 https://github.com/percona/percona-server-mongodb-operator
```

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

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

For ARM64 architecture

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

```
git clone -b v1.22.0 https://github.com/percona/percona-server-mongodb-operator
```

Edit the `deploy/cr.yaml` configuration file: set `image` and `backup.image` Custom Resource options to special multi-architecture image versions by adding a `-multi` suffix to their tags:

```
....
image: percona/percona-server-mongodb:7.0.30-16-multi
...
backup:
  ...
  image: percona/percona-backup-mongodb:2.12.0-multi
```

Please note, that currently monitoring with PMM is not supported on ARM64 configurations.

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

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

Expected output

```
perconaservermongodb.psmdb.percona.com/my-cluster-name created
```

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 psmdb
```

Expected output

```
NAME              ENDPOINT                                           STATUS    AGE
my-cluster-name   my-cluster-name-mongos.default.svc.cluster.local   ready     5m26s
```

Verifying the cluster operation

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

To connect to Percona Server for MongoDB you need to construct the MongoDB connection URI string. It includes the credentials of the admin user, which are stored in the Secrets object.

1. List the Secrets objects

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

The Secrets object you are interested in has the `my-cluster-name-secrets` name by default.

2. View the Secret contents to retrieve the admin user credentials.

```
kubectl get secret my-cluster-name-secrets -o yaml
```

The command returns the YAML file with generated Secrets, including the `MONGODB_DATABASE_ADMIN_USER` and `MONGODB_DATABASE_ADMIN_PASSWORD` strings, which should look as follows:

Sample output


```
...
data:
  ...
  MONGODB_DATABASE_ADMIN_PASSWORD: aDAzQ0pCY3NSWEZ2ZUIzS1I=
  MONGODB_DATABASE_ADMIN_USER: ZGF0YWJhc2VBZG1pbg==
```


The actual login name and password on the output are base64-encoded. To bring it back to a human-readable form, run:

```
echo 'MONGODB_DATABASE_ADMIN_USER' | base64 --decode
echo 'MONGODB_DATABASE_ADMIN_PASSWORD' | base64 --decode
```

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

```
kubectl run -i --rm --tty percona-client --image=percona/percona-server-mongodb:7.0.30-16 --restart=Never -- bash -il
```

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

4. Now run `mongosh` tool inside the `percona-client` command shell using the admin user credentialds you obtained from the Secret, and a proper namespace name instead of the `<namespace name>` placeholder. The command will look different depending on whether sharding is on (the default behavior) or off:

if sharding is on

```
mongosh "mongodb://databaseAdmin:databaseAdminPassword@my-cluster-name-mongos.<namespace name>.svc.cluster.local/admin?ssl=false"
```

if sharding is off

```
mongosh "mongodb+srv://databaseAdmin:databaseAdminPassword@my-cluster-name-rs0.<namespace name>.svc.cluster.local/admin?replicaSet=rs0&ssl=false"
```

Troubleshooting

If `kubectl get psmdb` 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
my-cluster-name-cfg-0                              2/2     Running   0          11m
my-cluster-name-cfg-1                              2/2     Running   1          10m
my-cluster-name-cfg-2                              2/2     Running   1          9m
my-cluster-name-mongos-0                           1/1     Running   0          11m
my-cluster-name-mongos-1                           1/1     Running   0          11m
my-cluster-name-mongos-2                           1/1     Running   0          11m
my-cluster-name-rs0-0                              2/2     Running   0          11m
my-cluster-name-rs0-1                              2/2     Running   0          10m
my-cluster-name-rs0-2                              2/2     Running   0          9m
percona-server-mongodb-operator-665cd69f9b-xg5dl   1/1     Running   0          37m
```

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 my-cluster-name-rs0-2
```

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.`

Removing the AKS cluster

To delete your cluster, you will need the following data:

- name of your AKS cluster,
- AWS region in which you have deployed your cluster.

You can clean up the cluster with the `az aks delete` command as follows (with real names instead of `<resource group>` and `<cluster name>` placeholders):

```
az aks delete --name <cluster name> --resource-group <resource group> --yes --no-wait
```

It may take ten minutes to get the cluster actually deleted after executing this command.

Warning

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

5.7 Install Percona server for MongoDB on Kubernetes

1. Clone the percona-server-mongodb-operator repository:

    ```
    git clone -b v1.22.0 https://github.com/percona/percona-server-mongodb-operator
    cd percona-server-mongodb-operator
    ```

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

2. The Custom Resource Definition for Percona Server for MongoDB should be created from the `deploy/crd.yaml` file. The Custom Resource Definition extends the standard set of resources which Kubernetes "knows" about with the new items, in our case these items 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; the step does not need to be repeated with any other Operator deployments.

3. 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/<namespace name>` was created, and the context was modified.

4. The role-based access control (RBAC) for Percona Server for MongoDB is configured with the `deploy/rbac.yaml` file. Role-based access is based on defined roles and the available actions which correspond to each role. The role and actions are defined for Kubernetes resources in the yaml file. Further 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)
    > ```

5. Start the operator within Kubernetes:

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

6. Add the MongoDB Users secrets to Kubernetes. These secrets should be placed as plain text in the stringData section of the `deploy/secrets.yaml` file as login name and passwords for the user accounts (see Kubernetes documentation for details).

    After editing the yaml file, MongoDB Users secrets should be created using the following command:

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

    More details about secrets can be found in Users.

7. 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.

8. After the operator is started, Percona Server for MongoDB cluster can be created with the following command:

    ```
    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 psmdb
```

Expected output

```
NAME              ENDPOINT                                          STATUS   AGE
my-cluster-name   my-cluster-name-mongos.default.svc.cluster.local  ready    5m26s
```

Verifying the cluster operation

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

To connect to Percona Server for MongoDB you need to construct the MongoDB connection URI string. It includes the credentials of the admin user, which are stored in the Secrets object.

1. List the Secrets objects

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

    The Secrets object you are interested in has the `my-cluster-name-secrets` name by default.

2. View the Secret contents to retrieve the admin user credentials.

    ```
    kubectl get secret my-cluster-name-secrets -o yaml
    ```

    The command returns the YAML file with generated Secrets, including the `MONGODB_DATABASE_ADMIN_USER` and `MONGODB_DATABASE_ADMIN_PASSWORD` strings, which should look as follows:

    Sample output

    ```
    ...
    data:
      ...
      MONGODB_DATABASE_ADMIN_PASSWORD: aDAzQ0pCY3NSWEZ2ZUIzS1I=
      MONGODB_DATABASE_ADMIN_USER: ZGF0YWJhc2VBZG1pbg==
    ```

    The actual login name and password on the output are base64-encoded. To bring it back to a human-readable form, run:

    ```
    echo 'MONGODB_DATABASE_ADMIN_USER' | base64 --decode
    echo 'MONGODB_DATABASE_ADMIN_PASSWORD' | base64 --decode
    ```

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

    ```
    kubectl run -i --rm --tty percona-client --image=percona/percona-server-mongodb:7.0.30-16 --restart=Never -- bash -il
    ```

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

4. Now run `mongosh` tool inside the `percona-client` command shell using the admin user credentialds you obtained from the Secret, and a proper namespace name instead of the `<namespace name>` placeholder. The command will look different depending on whether sharding is on (the default behavior) or off:

if sharding is on

```
mongosh "mongodb://databaseAdmin:databaseAdminPassword@my-cluster-name-mongos.<namespace name>.svc.cluster.local/admin?ssl=false"
```

if sharding is off

```
mongosh "mongodb+srv://databaseAdmin:databaseAdminPassword@my-cluster-name-rs0.<namespace name>.svc.cluster.local/admin?replicaSet=rs0&ssl=false"
```

5.8 Install Percona Server for MongoDB on OpenShift

Percona Operator for Percona Server for MongoDB is a [Red Hat Certified Operator](). This means that Percona Operator is portable across hybrid clouds and fully supports the Red Hat OpenShift lifecycle.

Installing Percona Server for MongoDB on OpenShift includes two steps:

- Installing the Percona Operator for MongoDB,
- Install Percona Server for MongoDB using the Operator.

Install the Operator

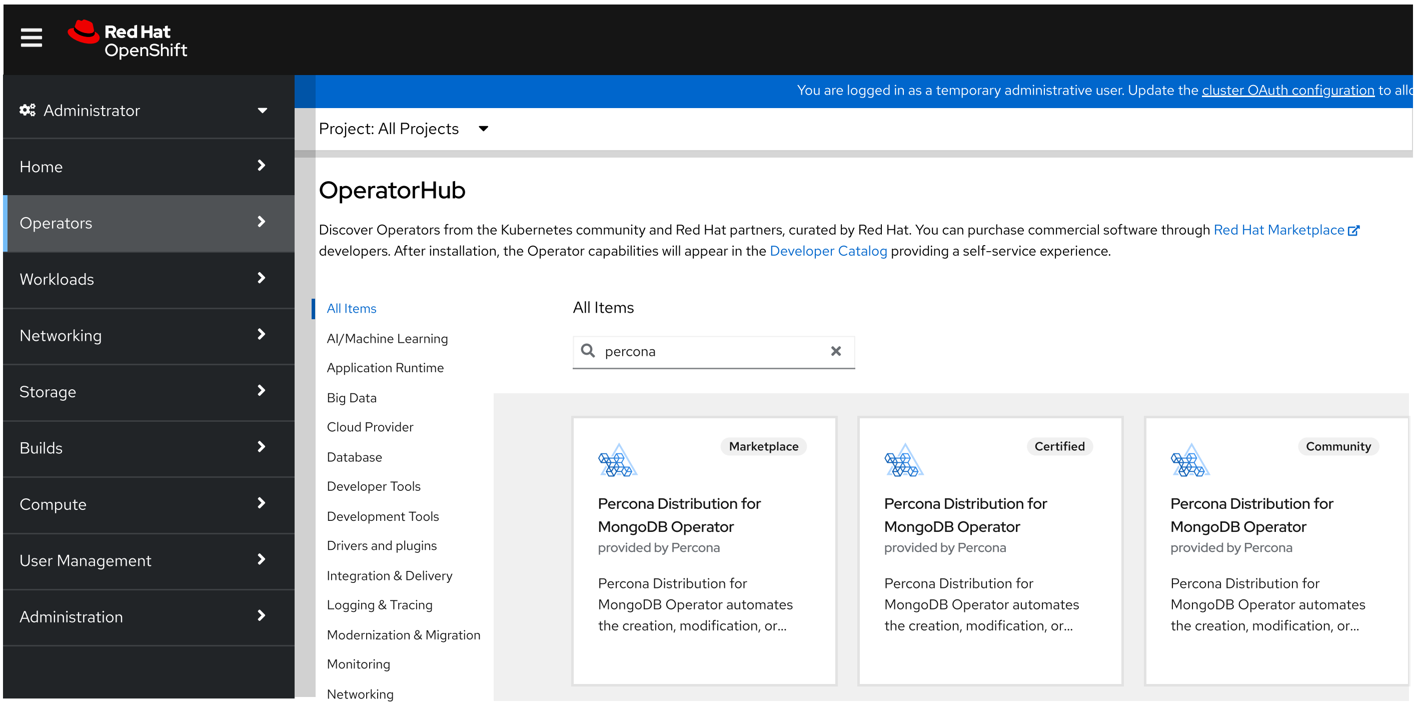
You can install Percona Operator for MongoDB on OpenShift using the web interface (the [Operator Lifecycle Manager]()), or using the command line interface.

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.

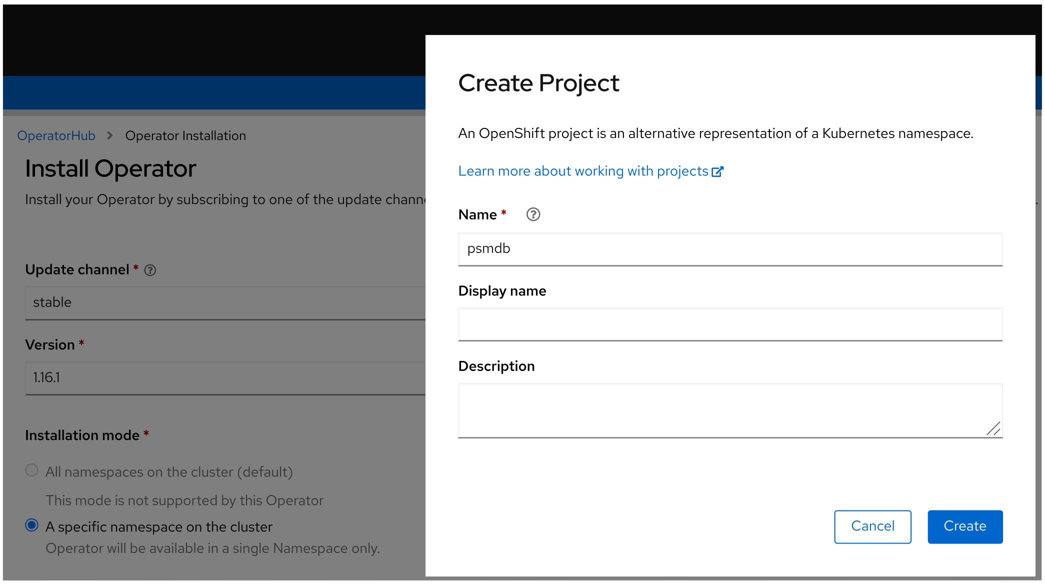
Following steps will allow you to deploy the Operator and Percona Server for MongoDB on your OLM installation:

1. Login to the OLM and click the needed Operator on the OperatorHub page:



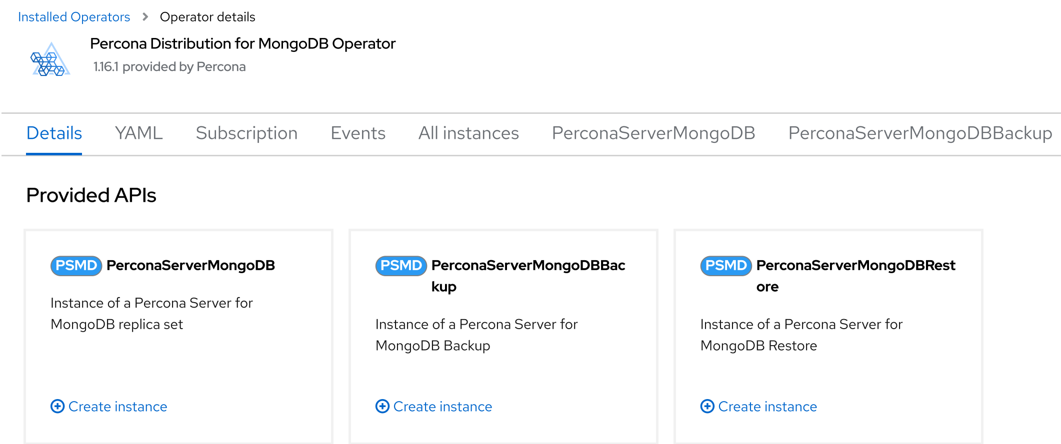
   Then click “Continue”, and “Install”.

2. A new page will allow you to choose the Operator version and the Namespace / OpenShift project you would like to install the Operator into.



Click “Install” button to actually install the Operator.

3. When the installation finishes, you can deploy your MongoDB cluster. In the “Operator Details” you will see Provided APIs (Custom Resources, available for installation). Click “Create instance” for the `PerconaServerMongoDB` Custom Resource.



You will be able to edit manifest to set needed Custom Resource options, and then click “Create” button to deploy your database cluster.

Install the Operator via the command-line interface

1. Clone the percona-server-mongodb-operator repository:

```
git clone -b v1.22.0 https://github.com/percona/percona-server-mongodb-operator
cd percona-server-mongodb-operator
```

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

2. The Custom Resource Definition for Percona Server for MongoDB should be created from the `deploy/crd.yaml` file. The Custom Resource Definition extends the standard set of resources which Kubernetes “knows” about with the new items, in our case these items are the core of the operator.

   This step should be done only once; it does not need to be repeated with other deployments.

   [Apply it](#) as follows:

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

> **Note**
>
> Setting Custom Resource Definition requires your user to have cluster-admin role privileges.

If you want to manage Percona Server for MongoDB cluster with a non-privileged user, the necessary permissions can be granted by applying the next clusterrole:

```
oc create clusterrole psmdb-admin --verb="*" --
resource=perconaservermongodbs.psmdb.percona.com,perconaservermongodbs.psmdb.percona.com/status,perconaservermongodbbackups
.psmdb.percona.com,perconaservermongodbbackups.psmdb.percona.com/status,perconaservermongodbrestores.psmdb.percona.com,perc
onaservermongodbrestores.psmdb.percona.com/status
oc adm policy add-cluster-role-to-user psmdb-admin <some-user>
```

If you have a cert-manager installed, then you have to execute two more commands to be able to manage certificates with a non-privileged user:

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

3. Create a new `psmdb` project:

```
oc new-project psmdb
```

4. Add role-based access control (RBAC) for Percona Server for MongoDB is configured with the `deploy/rbac.yaml` file. RBAC is based on clearly defined roles and corresponding allowed actions. These actions are allowed on specific Kubernetes resources. The details about users and roles can be found in OpenShift documentation.

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

5. Start the Operator within OpenShift:

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

Install Percona Server for MongoDB

1. Add the MongoDB Users secrets to OpenShift. These secrets should be placed as plain text in the stringData section of the `deploy/secrets.yaml` file as login name and passwords for the user accounts (see Kubernetes documentation for details).

   After editing the yaml file, the secrets should be created with the following command:

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

   More details about secrets can be found in Users.

2. 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.
3. Percona Server for MongoDB cluster can be created at any time with the following steps:

   a. Uncomment the `deploy/cr.yaml` field `#platform:` and edit the field to `platform: openshift`. The result should be like this:

```
apiVersion: psmdb.percona.com/v1alpha1
kind: PerconaServerMongoDB
metadata:
  name: my-cluster-name
spec:
  platform: openshift
...
```

   b. (optional) In you're using minishift, please adjust antiaffinity policy to `none`

```
   affinity:
     antiAffinityTopologyKey: "none"
...
```

c. Create/apply the Custom Resource file:

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

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

```
oc get psmdb
```

Expected output

```
NAME              ENDPOINT                                           STATUS   AGE
my-cluster-name   my-cluster-name-mongos.default.svc.cluster.local   ready    5m26s
```

Verifying the cluster operation

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

To connect to Percona Server for MongoDB you need to construct the MongoDB connection URI string. It includes the credentials of the admin user, which are stored in the [Secrets](#) object.

1. List the Secrets objects

   ```
   oc get secrets -n <namespace>
   ```

   The Secrets object you are interested in has the `my-cluster-name-secrets` name by default.

2. View the Secret contents to retrieve the admin user credentials.

   ```
   oc get secret my-cluster-name-secrets -o yaml
   ```

   The command returns the YAML file with generated Secrets, including the `MONGODB_DATABASE_ADMIN_USER` and `MONGODB_DATABASE_ADMIN_PASSWORD` strings, which should look as follows:

   Sample output

   ```
   ...
   data:
     ...
     MONGODB_DATABASE_ADMIN_PASSWORD: aDAzQ0pCY3NSWEZ2ZUIzS1I=
     MONGODB_DATABASE_ADMIN_USER: ZGF0YWJhc2VBZG1pbg==
   ```

   The actual login name and password on the output are base64-encoded. To bring it back to a human-readable form, run:

   ```
   echo 'MONGODB_DATABASE_ADMIN_USER' | base64 --decode
   echo 'MONGODB_DATABASE_ADMIN_PASSWORD' | base64 --decode
   ```

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

   ```
   oc run -i --rm --tty percona-client --image=percona/percona-server-mongodb:7.0.30-16 --restart=Never -- bash -il
   ```

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

4. Now run `mongosh` tool inside the `percona-client` command shell using the admin user credentialds you obtained from the Secret, and a proper namespace name instead of the `<namespace name>` placeholder. The command will look different depending on whether sharding is on (the default behavior) or off:

if sharding is on

```
mongosh "mongodb://databaseAdmin:databaseAdminPassword@my-cluster-name-mongos.<namespace name>.svc.cluster.local/admin?
ssl=false"
```

if sharding is off

```
mongosh "mongodb+srv://databaseAdmin:databaseAdminPassword@my-cluster-name-rs0.<namespace name>.svc.cluster.local/admin?
replicaSet=rs0&ssl=false"
```

6 Upgrade

6.1 Update Percona Operator for MongoDB

You can upgrade Percona Operator for MongoDB to newer versions. The upgrade process consists of these steps:

- Upgrade the Operator
- Upgrade the database (Percona Server for MongoDB).

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

You can chose how you want to update your database cluster when you run an upgrade:

- *Smart Update* 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. It is the default and recommended way to update.
- *Rolling Update* is initiated manually and controlled by Kubernetes. The StatefulSet controller in Kubernetes deletes a Pod, updates it, waits till it reports the Ready status and proceeds to the next Pod. The order for Pod update is the same as for Pod termination. However, this order may not be optimal from the Percona Server for MongoDB point of view.
- *On Delete* strategy requires a user to manually delete a Pod to make Kubernetes StatefulSet controller recreate it with the updated configuration.

To select an update strategy, set the `updateStrategy` key in the Custom Resource manifest to one of the following:

- `SmartUpdate`
- `RollingUpdate`
- `OnDelete`

For a manual update of your database cluster using the `RollingUpdate` or `OnDelete` strategies, refer to the low-level Kubernetes way of database upgrades guide.

Update on OpenShift

If you run the Operator on Red Hat Marketplace or you run Red Hat certified Operators on OpenShift, you need to do additional steps during the upgrade. See Upgrade Percona Server for MongoDB on OpenShift for details.

6.2 Upgrade CRD and the Operator

6.2.1 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.

The upgrade process is similar for all installation methods, including Helm and OLM.

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, and that point-in-time recovery is enabled to meet your Recovery Point Objective (RPO).
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.

For the 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 `1.18.0`:
   - `1` - major version
   - `18` - minor version
   - `0` - patch version

   You can only upgrade the Operator to the nearest `major.minor` version. For example, from `1.18.0` to `1.19.0`. To upgrade to a newer version, which differs from the current `minor.major` version by more than one, you need to make several incremental upgrades sequentially. For example, to upgrade the Operator from `1.17.0` to `1.19.1`, you need to first upgrade it to `1.18.0`, and then to `1.19.1`.

   Patch versions don't influence the upgrade, so you can safely move from `1.18.0` to `1.19.1`.

   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 previous minor versions. If the Operator is older than the CRD by no more than two versions, you should be able to continue using the old Operator version. But updating the CRD and Operator is the recommended path.
3. Starting from version 1.14.0, the Operator configures replica set members using local fully-qualified domain names (FQDN). Before this version, if you exposed a replica set, it used the exposed IP addresses in the replica set configuration. Therefore, if you upgrade to version 1.14.0 and your replica set is exposed, the replica set configuration will change to use FQDN. To prevent such reconfiguration, set the `clusterServiceDNSMode` Custom Resource option to `External` before the upgrade.
4. Starting with version 1.12.0, the Operator no longer has a separate API version for each release in CRD. Instead, the CRD has the API version `v1`. Therefore, if you installed the CRD when the Operator version was **older than 1.12.0**, you must update the API version in the CRD manually to run the upgrade. To check your CRD version, use the following command:

   ```
   kubectl get crd perconaservermongodbs.psmdb.percona.com -o yaml | yq .status.storedVersions
   ```

Sample output

```
- v1-11-0
- v1
```

If the CRD version is other than `v1` or has several entries, run the manual update.

5. Starting from the Operator version 1.15.0, the `spec.mongod` section (deprecated since 1.12.0) is finally removed from the Custom Resource configuration. If you have encryption disabled using the deprecated `mongod.security.enableEncryption` option, you need to set encryption as disabled via the custom configuration before the upgrade:

```
spec:
  ...
  replsets:
    - name: rs0
      ...
      configuration: |
        security:
          enableEncryption: false
        ...
```

6. Starting from the Operator version 1.16.0, MongoDB 4.4 support in the Operator has reached its end-of-life. Make sure that you have a supported MongoDB version before upgrading the Operator to 1.16.0 (you can use major version upgrade functionality) to fix it.
7. The Operator versions 1.19.0 and 1.19.1 have a recommended MongoDB version set to 7.0 because point-in-time recovery may fail on MongoDB 8.0 if sharding is enabled and the Operator version is 1.19.x. Therefore, upgrading to Operator 1.19.0/1.19.1 is not recommended for sharded MongoDB 8.0 clusters.

Update guides

Update manually | Update via Helm | Update via OLM

6.2.2 Upgrade the Operator and CRD manually

The upgrade includes the following steps.

1. **For Operators older than v1.12.0**: Update the API version in the Custom Resource Definition:

   Manually

   ```
   kubectl proxy &  \
   curl \
          --header "Content-Type: application/json-patch+json" \
          --request PATCH \
          --data '[{"op": "replace", "path": "/status/storedVersions", "value":["v1"]}]' --url
   "http://localhost:8001/apis/apiextensions.k8s.io/v1/customresourcedefinitions/perconaservermongodbs.psmdb.percona.com/status"
   ```

   Expected output

   ```
   {
    {...},
     "status": {
       "storedVersions": [
         "v1"
       ]
     }
   }
   ```

   Via `kubectl patch`

   ```
   kubectl patch customresourcedefinitions perconaservermongodbs.psmdb.percona.com --subresource='status' --type='merge' -p
   '{"status":{"storedVersions":["v1"]}}'
   ```

   Expected output

   ```
   customresourcedefinition.apiextensions.k8s.io/perconaservermongodbs.psmdb.percona.com patched
   ```

2. 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-mongodb-
   operator/v1.22.0/deploy/crd.yaml
   kubectl apply -f https://raw.githubusercontent.com/percona/percona-server-mongodb-operator/v1.22.0/deploy/rbac.yaml
   ```

3. Next, update the Percona Server for MongoDB 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. Then apply a patch to the Operator Deployment and specify the image name and version. Use the following command to update the Operator Deployment to the `1.22.0` version:

   ```
   kubectl patch deployment percona-server-mongodb-operator \
      -p'{"spec":{"template":{"spec":{"containers":[{"name":"percona-server-mongodb-operator","image":"percona/percona-server-
   mongodb-operator:1.22.0"}]}}}}'
   ```

4. 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-mongodb-operator
   ```

   Note

   Labels set on the Operator Pod will not be updated during upgrade.

5. Update the Custom Resource version, the database, the backup 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 done it 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 psmdb` command.

For example, to update the cluster with the name `my-cluster-name` to the `1.22.0` version, the command is as follows:

With PMM Client

```
kubectl patch psmdb my-cluster-name --type=merge --patch '{
   "spec": {
      "crVersion":"1.22.0",
      "image": "percona/percona-server-mongodb:8.0.19-7",
      "backup": { "image": "percona/percona-backup-mongodb:2.12.0" },
      "pmm": { "image": "percona/pmm-client:2.44.1-1" },
      "logcollector": { "image": "percona/fluentbit:4.0.1-2" }
   }}'
```

Without PMM Client

```
kubectl patch psmdb my-cluster-name --type=merge --patch '{
   "spec": {
      "crVersion":"1.22.0",
      "image": "percona/percona-server-mongodb:8.0.19-7",
      "backup": { "image": "percona/percona-backup-mongodb:2.12.0" },
      "logcollector": { "image": "percona/fluentbit:4.0.1-2" }
   }}'
```

6.2.3 Upgrade the Operator and CRD with 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`.

CRD management by Helm

If you installed the Operator deployment from the main `psmdb-operator` chart, Helm installs the CRDs from the `crds/` directory.

Helm v3 doesn't update the CRDs by default to prevent accidental CRD upgrades that could break existing resources. Starting with version 1.22.0, a dedicated CRD chart `psmdb-operator-crds` was added that enables Helm to handle automatic CRD updates. The use of the separate CRD chart also provides better compatibility with GitOPs tools such as ArgoCD and FluxCD and ensures version control and rollback capability for CRDs.

Learn more about the dedicated CRD chart.

To update the CRDs you have the following options:

- update them manually before you update the Operator deployment. Use this option if you run the Operator version *before* 1.22.0.
- install a dedicated CRD chart and update the CRDs using Helm

Upgrade steps

1. You must have the compatible version of the Custom Resource Definition (CRD) in all namespaces that the Operator manages. Starting with version 1.21.0, you can check it using the following command:

   ```
   kubectl get crd perconaservermongodbs.psmdb.percona.com --show-labels
   ```

2. Export the namespace where the Operator deployment is running as an environment variable:

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

3. Update the Custom Resource Definition for the Operator.

   Add the CRD chart (recommended)

   The steps differ slightly depending on the Helm version you run.

Helm v3.17.x and later

Install the `psmdb-operator-crds` Helm chart that contains the CRDs

```
helm upgrade --install psmdb-operator-crds percona/psmdb-operator-crds \
  --namespace $NAMESPACE \
  --version 1.22.0 \
  --take-ownership
```

The `--take-ownership` flag tells Helm to take ownership of existing CRDs that were previously installed from the `crds/` directory. This adds Helm metadata (labels and annotations) to the existing CRDs so that Helm can manage them going forward.

Helm earlier than v3.17.x

For Helm earlier than 3.17, the `--take-ownership` flag is not supported, and you need to add Helm ownership metadata first.

a. Export the CRDs as environment variable and then label and annotate them:

```
CRDS=(
  perconaservermongodbs.psmdb.percona.com
  perconaservermongodbbackups.psmdb.percona.com
  perconaservermongodbrestores.psmdb.percona.com
)

kubectl label crds "${CRDS[@]}" app.kubernetes.io/managed-by=Helm --overwrite
kubectl annotate crds "${CRDS[@]}" meta.helm.sh/release-name=psmdb-operator-crds --overwrite
kubectl annotate crds "${CRDS[@]}" meta.helm.sh/release-namespace=$NAMESPACE --overwrite
```

b. Install the CRD chart:

```
helm install psmdb-operator-crds percona/psmdb-operator-crds \
--namespace $NAMESPACE \
--version 1.22.0
```

> **Important**
>
> Although you install the CRD Helm chart (`psmdb-operator-crds`) in a specific namespace, the CRDs themselves are cluster-scoped resources and apply to *all* namespaces in your Kubernetes environment. If you uninstall or delete this CRD chart, it will remove the CRDs from the Kubernetes environment. This will irreversibly delete *all* Percona Server for MongoDB clusters managed by these CRDs across *all* namespaces where you deployed them.

Update manually

Update the Custom Resource Definition for the Operator, taking it from the official repository on Github.

Refer to the [compatibility between CRD and the Operator] and how you can update the CRD if it is too old. Use the following command and replace the version to the required one until you are safe to update to the latest CRD version.

```
kubectl apply --server-side -f https://raw.githubusercontent.com/percona/percona-server-mongodb-
operator/v1.22.0/deploy/crd.yaml -n $NAMESPACE
```

If you already have the latest CRD version in one of namespaces, don't re-run intermediate upgrades for it.

4. Upgrade the Operator deployment

With default parameters

To upgrade the Operator installed with default parameters, use the following command:

```
helm upgrade my-op percona/psmdb-operator --version 1.22.0
```

With customized parameters

If you installed the Operator with some [customized parameters](), list these options in the upgrade command.

a. Get the list of used options in YAML format :

```
helm get values my-op -a > my-values.yaml
```

b. Pass these options to the upgrade command as follows:

```
helm upgrade my-op percona/psmdb-operator --version 1.22.0 -f my-values.yaml
```

The `my-op` parameter in the above example is the name of a [release object]() which you have chosen for the Operator when installing its Helm chart.

During the upgrade, you may see a warning to manually apply the CRD if it has the outdated version. In this case, refer to step 3 to upgrade the CRD and then step 4 to upgrade the deployment.

Troubleshooting CRD upgrades with Helm

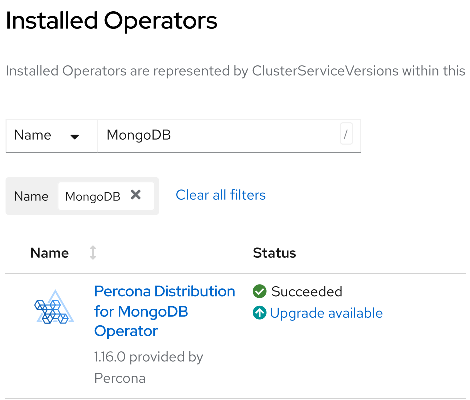
Error: "invalid ownership metadata" or "CRD already exists"

This happens when existing CRDs were installed from `crds/` and Helm does not own them. Pass the `--take-ownership` flag when you install the `psmdb-operator-crds` chart (Helm 3.17+). For Helm < 3.17+, add Helm ownership metadata before installing or upgrading `psmdb-operator-crds`.

6.2.4 Upgrade the Operator and CRD via Operator Lifecycle Manager (OLM)

If you have installed the Operator on the OpenShift platform using OLM, you can upgrade the Operator within it.

1. List installed Operators for your Namespace to see if there are upgradable items.



2. Click the “Upgrade available” link to see upgrade details, then click “Preview InstallPlan” button, and finally “Approve” to upgrade the Operator.

6.3 Upgrade Percona Server for MongoDB

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. The Operator updates the database to it automatically.

6.4 Minor upgrade

6.4.1 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.

Read more about the Smart Update strategy and other available ones in the Upgrade strategies section.

Procedure

To upgrade Percona Server for MongoDB to a specific version, do the following:

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 be the following:

   - `spec.updateStrategy` option is set to `SmartUpdate`
   - `spec.upgradeOptions.apply` option is set to `Disabled` or `Never`.

   ```
   ...
   spec:
     updateStrategy: SmartUpdate
     upgradeOptions:
       apply: Disabled
       ...
   ```

3. Check the current version of the Custom Resource and what versions of the database and cluster components are compatible with it. Use the following command:

   ```
   curl https://check.percona.com/versions/v1/psmdb-operator/1.22.0 |jq -r '.versions[].matrix'
   ```

   You can also find this information in the Versions compatibility matrix.

4. Update the database, the backup and PMM Client image names with a newer version tag. 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 done it yet, exclude PMM Client from the list of images to update.

   Starting with version 1.21.0, the Operator supports persistent logging. To update to version 1.21.0, exclude the `logcollector` 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 psmdb` command.

   This example command updates the cluster with the name `my-cluster-name` to the `1.22.0` version:

   With PMM Client

   ```
   kubectl patch psmdb my-cluster-name --type=merge --patch '{
      "spec": {
         "crVersion":"1.22.0",
         "image": "percona/percona-server-mongodb:8.0.19-7",
         "backup": { "image": "percona/percona-backup-mongodb:2.12.0" },
         "pmm": { "image": "percona/pmm-client:2.44.1-1" },
         "logcollector": { "image": "percona/fluentbit:4.0.1-2" }
      }}'
   ```

   Without PMM Client

   ```
   kubectl patch psmdb my-cluster-name --type=merge --patch '{
      "spec": {
         "crVersion":"1.22.0",
         "image": "percona/percona-server-mongodb:8.0.19-7",
         "backup": { "image": "percona/percona-backup-mongodb:2.12.0" },
         "logcollector": { "image": "percona/fluentbit:4.0.1-2" }
      }}'
   ```

5. After you applied the patch, the deployment rollout will be triggered automatically. You can track the rollout process in real time using the `kubectl rollout status` command with the name of your cluster:

```
kubectl rollout status sts my-cluster-name-rs0
```

The update process is successfully finished when all Pods have been restarted. If you turned on Percona Server for MongoDB Sharding, the mongos and Config Server nodes must be restarted too to complete the upgrade.

6.4.2 Automatic minor upgrade to the latest / recommended 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.

Read more about the Smart Update strategy and other available ones in the Upgrade strategies section.

Procedure

You can configure the Operator to automatically upgrade Percona Server for MongoDB to the latest available, the recommended or to a specific version of your choice. Learn more about automatic upgrades.

The steps are the following:

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. Make sure that `spec.updateStrategy` option in the Custom Resource manifest is set to `SmartUpdate`.

3. Set the `upgradeOptions.apply` option in the Custom Resource manifest from `Disabled` to one of the following values:

    - `Recommended` - automatic upgrade will choose the most recent version of software flagged as Recommended (for clusters created from scratch, the Percona Server for MongoDB 8.0 version will be selected instead of the Percona Server for MongoDB 7.0 or 6.0 version regardless of the image path; for already existing clusters, the 8.0 vs. 7.0 vs. 6.0 branch choice will be preserved),

    - `8.0-recommended`, `7.0-recommended`, `6.0-recommended` - same as above, but preserves specific major MongoDB version for newly provisioned clusters (ex. 8.0 will not be automatically used instead of 7.0),

    - `Latest` - automatic upgrade will choose the most recent version of the software available (for clusters created from scratch, the Percona Server for MongoDB 8.0 version will be selected instead of the Percona Server for MongoDB 7.0 or 6.0 version regardless of the image path; for already existing clusters, the 8.0 vs. 7.0 vs. 6.0 branch choice will be preserved),

    - `8.0-latest`, `7.0-latest`, `6.0-latest` - same as above, but preserves specific major MongoDB version for newly provisioned clusters (ex. 8.0 will not be automatically used instead of 7.0),

    - *version number* - specify the desired version explicitly (version numbers are specified as 6.0.27-21, 7.0.30-16, etc.). Actual versions can be found in the list of certified images.

    > **End of Life versions of MongoDB**
    >
    > - MongoDB 5.0 support has reached its end-of-life in the Operator version 1.19.0. Therefore, the `5.0-recommended` or `5.0-latest` values are no longer supported. Users of existing clusters based on Percona Server for MongoDB 5.0 should explicitly switch to a newer major database version before upgrading the Operator to 1.19.0.
    > - MongoDB 4.4 support has reached its end-of-life in the Operator version 1.16.0. Therefore, the `4.4-recommended` or `4.4-latest` values are no longer supported. Users of existing clusters based on Percona Server for MongoDB 4.4 should explicitly switch to a newer major database version before upgrading the Operator to 1.16.0.

4. Make sure to set the valid Version Server URL for the `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
    ```

5. 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.

6. Apply your changes to the Custom Resource:

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

6.4.3 Manual upgrades of Percona Server for MongoDB

The default and recommended way to upgrade the database cluster is using the Smart Update strategy. The Operator controls how objects are updated and restarts the Pods in a proper order during the database upgrade or for other events that require the cluster update. To these events belong ConfigMap updates, password rotation or changing resource values.

In some cases running an automatic upgrade of Percona Server for MongoDB is not an option. For example, if the database upgrade impacts your application, you may want to have a full control over the upgrade process.

Running a manual database upgrade allows you to do just that. You can use one of the following *upgrade strategies*:

- *Rolling Update*, initiated manually and controlled by Kubernetes. Note that the order of Pods restart may not be optimal from the Percona Server for MongoDB point of view.
- *On Delete*, done by Kubernetes on per-Pod basis when Pods are manually deleted.

Rolling Update strategy and semi-automatic updates

To run a semi-automatic update of Percona Server for MongoDB, do the following:

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. Edit the `deploy/cr.yaml` file and set the `updateStrategy` key to `RollingUpdate`.
3. Check the current version of the Custom Resource and what versions of the database and cluster components are compatible with it. Use the following command:

```
curl https://check.percona.com/versions/v1/psmdb-operator/1.22.0 |jq -r '.versions[].matrix'
```

You can also find this information in the Versions compatibility matrix.

4. Update the database, the backup and PMM Client image names with a newer version tag. 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 done it yet, exclude PMM Client from the list of images to update.

Starting with version 1.21.0, the Operator supports persistent logging. To update to version 1.21.0, exclude the `logcollector` 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 psmdb` command.

This example command updates the cluster with the name `my-cluster-name` to the `1.22.0` version:

With PMM Client

```
kubectl patch psmdb my-cluster-name --type=merge --patch '{
   "spec": {
      "crVersion":"1.22.0",
      "image": "percona/percona-server-mongodb:8.0.19-7",
      "backup": { "image": "percona/percona-backup-mongodb:2.12.0" },
      "pmm": { "image": "percona/pmm-client:2.44.1-1" },
      "logcollector": { "image": "percona/fluentbit:4.0.1-2" }
   }}'
```

Without PMM Client

```
kubectl patch psmdb my-cluster-name --type=merge --patch '{
   "spec": {
      "crVersion":"1.22.0",
      "image": "percona/percona-server-mongodb:8.0.19-7",
      "backup": { "image": "percona/percona-backup-mongodb:2.12.0" },
      "logcollector": { "image": "percona/fluentbit:4.0.1-2" }
   }}'
```

5. After you applied the patch, the deployment rollout will be triggered automatically. You can track the rollout process in real time using the `kubectl rollout status` command with the name of your cluster:

```
kubectl rollout status sts my-cluster-name-rs0
```

Manual upgrade (the On Delete strategy)

To upgrade Percona Server for MongoDB manually, do following:

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. Edit the `deploy/cr.yaml` file and set the `updateStrategy` key to `OnDelete`.

3. Check the current version of the Custom Resource and what versions of the database and cluster components are compatible with it. Use the following command:

   ```
   curl https://check.percona.com/versions/v1/psmdb-operator/1.22.0 |jq -r '.versions[].matrix'
   ```

   You can also find this information in the Versions compatibility matrix.

4. Update the database, the backup and PMM Client image names with a newer version tag. 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 done it yet, exclude PMM Client from the list of images to update.

   Starting with version 1.21.0, the Operator supports persistent logging. To update to version 1.21.0, exclude the `logcollector` 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 psmdb` command.

   This example command updates the cluster with the name `my-cluster-name` to the `1.22.0` version:

   With PMM Client

   ```
   kubectl patch psmdb my-cluster-name --type=merge --patch '{
      "spec": {
         "crVersion":"1.22.0",
         "image": "percona/percona-server-mongodb:8.0.19-7",
         "backup": { "image": "percona/percona-backup-mongodb:2.12.0" },
         "pmm": { "image": "percona/pmm-client:2.44.1-1" },
         "logcollector": { "image": "percona/fluentbit:4.0.1-2" }
      }}'
   ```

   Without PMM Client

   ```
   kubectl patch psmdb my-cluster-name --type=merge --patch '{
      "spec": {
         "crVersion":"1.22.0",
         "image": "percona/percona-server-mongodb:8.0.19-7",
         "backup": { "image": "percona/percona-backup-mongodb:2.12.0" },
         "logcollector": { "image": "percona/fluentbit:4.0.1-2" }
      }}'
   ```

5. The Pod with the newer Percona Server for MongoDB image will start after you delete it. Delete targeted Pods manually one by one to make them restart in the desired order:

   1. Delete the Pod using its name with the command like the following one:

      ```
      $ kubectl delete pod my-cluster-name-rs0-2
      ```

   2. Wait until Pod becomes ready:

      ```
      $ kubectl get pod my-cluster-name-rs0-2
      ```

      The output should be like this:

      ```
      NAME                    READY   STATUS    RESTARTS   AGE
      my-cluster-name-rs0-2   1/1     Running   0          3m33s
      ```

The update process is successfully finished when all Pods have been restarted. If you turned on Percona Server for MongoDB Sharding, the mongos and Config Server nodes must be restarted too to complete the upgrade.

6.5 Major version automated upgrades

Major version upgrade is moving from the current major version to the next one. For example, from 6.0.x to 7.0.x.

It is a more complicated task than a minor version upgrade because it might potentially affect how data is stored and how applications interact with the database (in case of some API changes). Therefore, we recommend to test the major version upgrade on a staging environment first.

A major upgrade is supported by the Operator within one major version at a time: for example, from 7.0 to 8.0. To upgrade Percona Server for MongoDB from 6.0 to 8.0, you should first upgrade it to 7.0, and then make a separate upgrade from 7.0 to 8.0. The same is true for major version downgrades.

> **Important**
>
> Before the upgrade, make a backup of your data.

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.

Read more about the Smart Update strategy and other available ones in the Upgrade strategies section.

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. Set the upgradeOptions.apply key in the `deploy/cr.yaml` Custom Resource manifest to `<version>-recommended`:

```
spec:
  upgradeOptions:
    apply: 8.0-recommended
```

1. Apply the `deploy/cr.yaml` Custom Resource manifest to start the major version upgrade.

Feature Compatibility Version

By default, the Operator doesn't set FeatureCompatibilityVersion (FCV) to match the new version. This ensures that backwards-incompatible features are not automatically enabled with the major version upgrade (which the is recommended and safe behavior).

You can turn this backward compatibility off at any moment (after the upgrade or even before it) by setting the upgradeOptions.setFCV flag in the `deploy/cr.yaml` configuration file to `true`.

Note that with this setting, the Operator doesn't yet support major version rollback if the `setFeatureCompatibilityVersion` is set. Therefore it is recommended to stay without enabling this flag for some time after the major upgrade to ensure that everything works as expected and you won't have to downgrade.

You can set the `setFCV` flag to `true` simultaneously with the `apply` flag but you must be absolutely sure that the whole procedure is tested on staging environment and works as expected.

Downgrades

When making downgrades (e.g. changing version from 7.0 to 6.0), make sure to remove incompatible features that are persisted and/or update incompatible configuration settings. Compatibility issues between major MongoDB versions can be found in upstream documentation.

6.6 Upgrade Database and Operator on OpenShift

Upgrading database and Operator on [Red Hat Marketplace](#) or to upgrade Red Hat certified Operators on [OpenShift](#) generally follows the [standard upgrade scenario](#), but includes a number of special steps specific for these platforms.

Upgrading the Operator and CRD

1. First of all you need to manually update `initImage` Custom Resource option with the value of an alternative initial Operator installation image. You need doing this for all database clusters managed by the Operator. Without this step the cluster will go into error state after the Operator upgrade.

    a. Find the initial Operator installation image with `kubectl get deploy` command:

    ```
    kubectl get deploy percona-server-mongodb-operator -o yaml
    ```

    Expected output

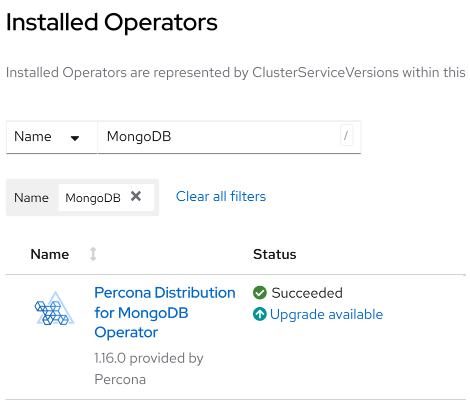
    ```
    ...
    "containerImage": "registry.connect.redhat.com/percona/percona-server-mongodb-
    operator@sha256:201092cf97c9ceaaaf3b60dd1b24c7c5228d35aab2674345893f4cd4d9bb0e2e",
    ...
    ```

    b. [Apply a patch](#) to update the `initImage` option of your cluster Custom Resource with this value taken from `containerImage`. Supposing that your cluster name is `my-cluster-name`, the command should look as follows:

    ```
    kubectl patch psmdb my-cluster-name --type=merge --patch '{
        "spec": {
           "initImage":"registry.connect.redhat.com/percona/percona-server-mongodb-
    operator@sha256:201092cf97c9ceaaaf3b60dd1b24c7c5228d35aab2674345893f4cd4d9bb0e2e"
        }}'
    ```

2. Now you can actually update the Operator via the [Operator Lifecycle Manager (OLM)](#) web interface.

    Login to your OLM installation and list installed Operators for your Namespace to see if there are upgradable items:



    Click the “Upgrade available” link to see upgrade details, then click “Preview InstallPlan” button, and finally “Approve” to upgrade the Operator.

Upgrading Percona Server for MongoDB

1. Make sure that `spec.updateStrategy` option in the [Custom Resource](#) 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
        ...
    ```

2. Find the **new** initial Operator installation image name (it had changed during the Operator upgrade) and other image names for the components of your cluster with the `kubectl get deploy` command:

```
kubectl get deploy percona-server-mongodb-operator -o yaml
```

Expected output

```
...
"image": "registry.connect.redhat.com/percona/percona-server-mongodb-operator-
containers@sha256:5d29132a60b89e660ab738d463bcc0707a17be73dc955aa8da9e50bed4d9ad3e",
...
"initImage": "registry.connect.redhat.com/percona/percona-server-mongodb-
operator@sha256:8adc57e9445cfcea1ae02798a8f9d6a4958ac89f0620b9c6fa6cf969545dd23f",
...
"pmm": {
  "enabled": true,
  "image": "registry.connect.redhat.com/percona/percona-server-mongodb-operator-
containers@sha256:165f97cdae2b6def546b0df7f50d88d83c150578bdb9c992953ed866615016f1",
...
"backup": {
  "enabled": true,
  "image": "registry.connect.redhat.com/percona/percona-server-mongodb-operator-
containers@sha256:a73889d61e996bc4fbc6b256a1284b60232565e128a64e4f94b2c424966772eb",
...
```

3. [Apply a patch](#) to set the necessary `crVersion` value (equal to the Operator version) and update images in your cluster Custom Resource. Supposing that your cluster name is `cluster1`, the command should look as follows:

```
kubectl patch psmdb my-cluster-name --type=merge --patch '{
    "spec": {
       "crVersion":"1.22.0",
       "image": "registry.connect.redhat.com/percona/percona-server-mongodb-operator-
containers@sha256:5d29132a60b89e660ab738d463bcc0707a17be73dc955aa8da9e50bed4d9ad3e",
        "initImage": "registry.connect.redhat.com/percona/percona-server-mongodb-
operator@sha256:8adc57e9445cfcea1ae02798a8f9d6a4958ac89f0620b9c6fa6cf969545dd23f",
       "pmm": {"image": "registry.connect.redhat.com/percona/percona-server-mongodb-operator-
containers@sha256:165f97cdae2b6def546b0df7f50d88d83c150578bdb9c992953ed866615016f1"},
       "backup": {"image": "registry.connect.redhat.com/percona/percona-server-mongodb-operator-
containers@sha256:a73889d61e996bc4fbc6b256a1284b60232565e128a64e4f94b2c424966772eb"}
    }}'
```

Warning

The above command upgrades various components of the cluster including PMM Client. If you didn't follow the [official recommendation](#) to upgrade PMM Server before upgrading PMM Client, you can avoid PMM Client upgrade by removing it from the list of images as follows:

```
kubectl patch psmdb my-cluster-name --type=merge --patch '{
    "spec": {
       "crVersion":"1.22.0",
       "image": "registry.connect.redhat.com/percona/percona-server-mongodb-operator-
containers@sha256:5d29132a60b89e660ab738d463bcc0707a17be73dc955aa8da9e50bed4d9ad3e",
       "initImage": "registry.connect.redhat.com/percona/percona-server-mongodb-
operator@sha256:8adc57e9445cfcea1ae02798a8f9d6a4958ac89f0620b9c6fa6cf969545dd23f",
       "backup": {"image": "registry.connect.redhat.com/percona/percona-server-mongodb-operator-
containers@sha256:a73889d61e996bc4fbc6b256a1284b60232565e128a64e4f94b2c424966772eb"}
    }}'
```

4. The deployment rollout will be automatically triggered by the applied patch.

7 Configuration

7.1 Manage users

7.1.1 Users

MongoDB user accounts within the cluster can be divided into two different groups:

- **Application-level users**: unprivileged user accounts for applications. You can create and manage these users.
- **System-level users**: privileged accounts needed to automate cluster deployment and management tasks, such as MongoDB health checks. The Operator creates these accounts during the cluster creation.

These two groups serve different purposes. Read the following sections to learn more:

- Application-level (unprivileged) users
- System users

Credentials management

The Operator stores user credentials in the Secret objects.

When the Operator creates application-level (unprivileged) users, it uses the username you specify in the Custom Resource (CR). The Operator either generates a password for this user and creates a corresponding Secret, or uses the credentials of a user you have manually created in MongoDB and creates a Secret with that data.

For system users, the Operator creates the required Secret automatically when it creates the cluster. If a Secret with the expected name already exists (as specified in the CR), the Operator will reuse it. Starting with version 1.22.0, you can also [store and manage system user credentials in HashiCorp Vault]. In this case, the Operator retrieves the passwords from Vault and creates or updates the relevant Secrets based on the Vault-stored credentials.

Authentication

Authentication is enabled in Percona Server for MongoDB clusters by default. If you need to disable authentication for development, testing, or migration purposes, see Disable authentication.

MongoDB internal authentication key (optional)

Default Secret name: `my-cluster-name-mongodb-keyfile`

Secret name field: `spec.secrets.key`

By default, the Operator creates a random, 1024-byte key for MongoDB Internal Authentication if it does not already exist. If you would like to deploy a different key, create the secret manually before starting the Operator. Example:

deploy/mongodb-keyfile.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: my-cluster-name-mongodb-keyfile
type: Opaque
data:
  mongodb-key: <replace-this-value-with-base-64-encoded-text>
```

7.1.2 Application-level (unprivileged) users

The Operator doesn't create application-level (unprivileged) user accounts by default.

You can create these unprivileged users in the following ways:

- Automatically via Custom Resource. This ability is available with the Operator versions 1.17.0 and newer
- Manually in Percona Server for MongoDB

Regardless of how you create users, their usernames must be unique.

Create users via Custom Resource

Starting from Operator version 1.17.0, you can create users in Percona Server for MongoDB via the `users` subsection in the Custom Resource. This is called declarative user management.

You can modify the `users` section in the `deploy/cr.yaml` configuration file either at cluster creation time or adjust it over time.

For every new user in the `deploy/cr.yaml` configuration file, specify the following:

- A username and the database where the user will be created. The username must be unique for every user
- Roles on databases in MongoDB that you want to grant to this user

Here's the example configuration:

```
...
users:
  - name: my-user
    db: admin
    passwordSecretRef:
      name: my-user-password
      key: password
    roles:
      - name: clusterAdmin
        db: admin
      - name: userAdminAnyDatabase
        db: admin
```

After you apply the configuration, the Operator creates a Secret named `<cluster-name>-custom-user-secret`, generates a password for the user, and sets it by the key named after the user name.

Generate user passwords manually

If you don't want the Operator to generate a user password automatically, you can create a Secret resource that contains the user password. Then specify a reference to this Secret resource in the `passwordSecretRef` key. You can find a detailed description of the corresponding options in the Custom Resource reference.

Here's how to do it:

1. Create a Secret configuration file:

   my-secret.yaml

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: my-user-password
   type: Opaque
   stringData:
     password: mypassword
   ```

2. Create a Secret object:

   ```
   kubectl apply -f my-secret.yaml
   ```

3. Reference this Secret in the Custom Resource

deploy/cr.yaml

```
...
users:
  - name: my-user
    db: admin
    passwordSecretRef:
      name: my-user-password
      key: password
    roles:
      - name: clusterAdmin
        db: admin
      - name: userAdminAnyDatabase
        db: admin
```

4. Apply the configuration to create users:

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

External database users

The Operator doesn't generate passwords for users created in the **`$external`** database. You can't set the `passwordSecretRef` for these users either.

Such users are used for authentication via an external authentication source, such as an LDAP server. The user credentials are stored in an external authentication source, and their usernames are mapped to those in the `$external` database during authentication.

The Operator tracks password changes in the Secret object and updates the user password in the database. This applies to manually created users as well: if a user was created manually in the database before creating the user via Custom Resource, the existing user is updated.

However, manual password updates in the database are not tracked: the Operator doesn't overwrite changed passwords with the old ones from the users Secret.

Custom MongoDB roles

Custom MongoDB roles allow providing fine-grained access control over your MongoDB deployment.

You can define custom MongoDB roles declaratively via the `roles` subsection in the Custom Resource.

This subsection contains an array of roles, each with:

- A defined custom name (`roles.name`)
- The database in which you want to store the user-defined role (`roles.db`)
- The `roles.privileges.actions` list of custom role actions that users granted this role can perform. For a list of accepted values, see Privilege Actions in the manual of the corresponding MongoDB version.

Here's what you can do with actions:

- You can grant actions either to the whole cluster (if `roles.privileges.resource.cluster` is set to true) or to a specific database or collection.
- You can inherit privileges from existing roles by adding existing role and database names to the `roles.roles` subsection,
- You can apply authentication restrictions for your custom role based on IP address ranges for the client and server.

The following example shows how the `roles` subsection may look:

```yaml
roles:
    - role: my-role
      db: admin
      privileges:
        - resource:
            db: ''
            collection: ''
          actions:
            - find
      authenticationRestrictions:
        - clientSource:
            - 127.0.0.1
          serverAddress:
            - 127.0.0.1
      roles:
        - role: read
          db: admin
        - role: readWrite
          db: admin
```

Find more information about available options and their accepted values in the [roles subsection of the Custom Resource reference](#).

Create users manually

You can create application-level users manually. Run the commands below, substituting the `<namespace name>` placeholder with the real namespace of your database cluster:

If sharding is enabled

1. Connect to Percona Server for MongoDB:

```
kubectl run -i --rm --tty percona-client --image=percona/percona-server-mongodb:8.0.19-7 --restart=Never -- bash -il
```

Sample output

```
mongodb@percona-client:/$
```

2. Start the `mongosh` session and create a user:

```
mongosh "mongodb://userAdmin:userAdmin123456@my-cluster-name--mongos.<namespace name>.svc.cluster.local/admin?ssl=false"
rs0:PRIMARY> db.createUser({
    user: "myApp",
    pwd: "myAppPassword",
    roles: [
      { db: "myApp", role: "readWrite" }
    ],
    mechanisms: [
       "SCRAM-SHA-1"
    ]
})
```

3. Test the newly created user:

```
rs0:PRIMARY> use myApp
rs0:PRIMARY> db.test.insert({ x: 1 })
rs0:PRIMARY> db.test.findOne()
```

If sharding is disabled

1. Connect to Percona Server for MongoDB:

```
kubectl run -i --rm --tty percona-client --image=percona/percona-server-mongodb:8.0.19-7 --restart=Never -- bash -il
```

Sample output

```
mongodb@percona-client:/$
```

2. Start the `mongosh` session and create a user:

```
mongosh "mongodb+srv://userAdmin:userAdmin123456@my-cluster-name-rs0.<namespace name>.svc.cluster.local/admin?
replicaSet=rs0&ssl=false"
rs0:PRIMARY> db.createUser({
    user: "myApp",
    pwd: "myAppPassword",
    roles: [
      { db: "myApp", role: "readWrite" }
    ],
    mechanisms: [
       "SCRAM-SHA-1"
    ]
})
```

3. Test the newly created user:

```
rs0:PRIMARY> use myApp
rs0:PRIMARY> db.test.insert({ x: 1 })
rs0:PRIMARY> db.test.findOne()
```

7.1.3 System users

To automate the deployment and management of cluster components, the Operator requires system-level MongoDB users.

Credentials for these users are stored in a Kubernetes Secrets object. The Operator first checks if a Secret already exists with the name specified in the `spec.secrets.users` field of your `deploy/cr.yaml` configuration file and reuses it if found. If no matching Secret exists, the Operator automatically creates a new Secret with randomly generated passwords during cluster creation.

System users Secrets

The default name of the Secrets object for the system users is `my-cluster-name-secrets`. It is referenced in the `spec.secrets.users` field of the Custom Resource.

Aside from this Secret, the Operator creates an internal secret for its own internal purposes. See Internal Secret and its usage to learn more.

> **Warning**
>
> Don't use system users to run applications.

User Purpose	Username Secret Key	Password Secret Key
Backup/Restore	MONGODB_BACKUP_USER	MONGODB_BACKUP_PASSWORD
Cluster Admin	MONGODB_CLUSTER_ADMIN_USER	MONGODB_CLUSTER_ADMIN_PASSWORD
Cluster Monitor	MONGODB_CLUSTER_MONITOR_USER	MONGODB_CLUSTER_MONITOR_PASSWORD
Database Admin	MONGODB_DATABASE_ADMIN_USER	MONGODB_DATABASE_ADMIN_PASSWORD
User Admin	MONGODB_USER_ADMIN_USER	MONGODB_USER_ADMIN_PASSWORD
PMM Server	PMM_SERVER_USER	PMM_SERVER_TOKEN

System users and MongoDB roles

The following table maps MongoDB roles to system users:

User Purpose	MongoDB Roles
Backup/Restore	backup, restore, clusterMonitor, readWrite, pbmAnyAction
Cluster Admin	clusterAdmin
Cluster Monitor	clusterMonitor, read (on the `local` database), explainRole
Database Admin	readWriteAnyDatabase, readAnyDatabase, dbAdminAnyDatabase, backup, restore, clusterMonitor
User Admin	userAdminAnyDatabase
PMM Server	See PMM documentation

If you change credentials for the `MONGODB_CLUSTER_MONITOR` user, the cluster Pods will go into a restart cycle, and the cluster may not be accessible through the `mongos` service until this cycle finishes.

YAML object format

You can create your own Secret and reference it in the CR for your cluster in `spec.secrets.users` key.

When you create the Secret object yourself, your YAML file should match the following simple format:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-cluster-name-secrets
type: Opaque
stringData:
  MONGODB_BACKUP_USER: backup
  MONGODB_BACKUP_PASSWORD: backup123456
  MONGODB_DATABASE_ADMIN_USER: databaseAdmin
  MONGODB_DATABASE_ADMIN_PASSWORD: databaseAdmin123456
  MONGODB_CLUSTER_ADMIN_USER: clusterAdmin
  MONGODB_CLUSTER_ADMIN_PASSWORD: clusterAdmin123456
  MONGODB_CLUSTER_MONITOR_USER: clusterMonitor
  MONGODB_CLUSTER_MONITOR_PASSWORD: clusterMonitor123456
  MONGODB_USER_ADMIN_USER: userAdmin
  MONGODB_USER_ADMIN_PASSWORD: userAdmin123456
  PMM_SERVER_USER: admin
  #PMM_SERVER_PASSWORD: admin
  #PMM_SERVER_API_KEY: apikey
  PMM_SERVER_TOKEN: token
```

The example above matches the default `deploy/secrets.yaml` file, which includes sample passwords and PMM Server credentials. These are intended only for development or automated testing. **Don't use them in production**.

Update the Secret

When you create the Secrets object, you use the `stringData` type and specify all values for each key/value in plain text format. However, the resulting Secrets object contains passwords stored as base64-encoded strings in the `data` type.

To update any field, you'll need to encode the value into base64 format.

Here's how to do it:

1. Run the following command in your local shell to encode the new value. Replace the `new_password` with your value:

   Linux

   ```
   echo -n "new_password" | base64 --wrap=0
   ```

   in macOS

   ```
   echo -n "new_password" | base64
   ```

2. Update the Secrets object. For example, the following command updates the Database Admin user's password to `new_password` in the `my-cluster-name-secrets` object can be done with the following command:

   in Linux

   ```
   kubectl patch secret/my-cluster-name-secrets -p '{"data":{"MONGODB_DATABASE_ADMIN_PASSWORD": "'$(echo -n new_password |
   base64 --wrap=0)'"}}'
   ```

   in macOS

   ```
   kubectl patch secret/my-cluster-name-secrets -p '{"data":{"MONGODB_DATABASE_ADMIN_PASSWORD": "'$(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 you don't need to take any action beyond changing the password.

> **Note**
>
> Please don't change the `secrets.users` option in the CR. Make changes inside the secrets object itself.

Reproduce system users for migration

In some situations, you may need to reproduce system users in a bare-bones MongoDB. For example, this is a required step in migration scenarios to move existing on-premises MongoDB databases to Kubernetes-based MongoDB clusters managed by the Operator. You can use the following example script that produces a text file with mongo shell commands to create the needed system users with appropriate roles:

gen_users.sh

```
clusterAdminPass="clusterAdmin"
userAdminPass="userAdmin"
clusterMonitorPass="clusterMonitor"
backupPass="backup"

# mongo shell
cat <<EOF > user-mongo-shell.txt
use admin
db.createRole({
    "roles" : [],
    "role" : "pbmAnyAction",
    "privileges" : [
        {
            "resource" : { "anyResource" : true },
            "actions" : [ "anyAction" ]
        }
    ]
})

db.createUser( { user: "clusterMonitor", pwd: "$clusterMonitorPass", roles: [ "clusterMonitor" ] } )
db.createUser( { user: "userAdmin", pwd: "$userAdminPass", roles: [ "userAdminAnyDatabase" ] } )
db.createUser( { user: "clusterAdmin", pwd: "$clusterAdminPass", roles: [ "clusterAdmin" ] } )
db.createUser( { user: "backup", pwd: "$backupPass", roles: [ "readWrite", "backup", "clusterMonitor", "restore", "pbmAnyAction" ] } )
EOF
```

Development mode

To make development and testing easier, the `deploy/secrets.yaml` secrets file contains default passwords for MongoDB system users.

These development-mode credentials from `deploy/secrets.yaml` are:

Secret Key	Secret Value
MONGODB_BACKUP_USER	backup
MONGODB_BACKUP_PASSWORD	backup123456
MONGODB_DATABASE_ADMIN_USER	databaseAdmin
MONGODB_DATABASE_ADMIN_PASSWORD	databaseAdmin123456
MONGODB_CLUSTER_ADMIN_USER	clusterAdmin
MONGODB_CLUSTER_ADMIN_PASSWORD	clusterAdmin123456
MONGODB_CLUSTER_MONITOR_USER	clusterMonitor
MONGODB_CLUSTER_MONITOR_PASSWORD	clusterMonitor123456
MONGODB_USER_ADMIN_USER	userAdmin
MONGODB_USER_ADMIN_PASSWORD	userAdmin123456
PMM_SERVER_USER	admin
PMM_SERVER_PASSWORD	admin
PMM_SERVER_API_KEY	apikey

> **Warning**
>
> Do not use the default MongoDB users and/or default PMM API key in production!

Internal Secret and its usage

The Operator creates and updates an additional Secrets object which is named based on the cluster name, like `internal-my-cluster-name-users`. This Secrets object is used only by the Operator. Users must not change it.

This object contains secrets with the same passwords as the one specified in `spec.secrets.users` (e.g., `my-cluster-name-secrets`). When the user updates the `my-cluster-name-secrets` Secret, the Operator propagates these changes to the internal `internal-my-cluster-name-users` Secrets object.

7.1.4 Manage system users with Vault

7.1.4.1 Manage system users with Vault

You can integrate the Percona Server for MongoDB Operator with HashiCorp Vault to fetch and manage system user credentials.

This allows you to centralize password management, enforce rotation policies, and maintain a single source of truth for sensitive credentials.

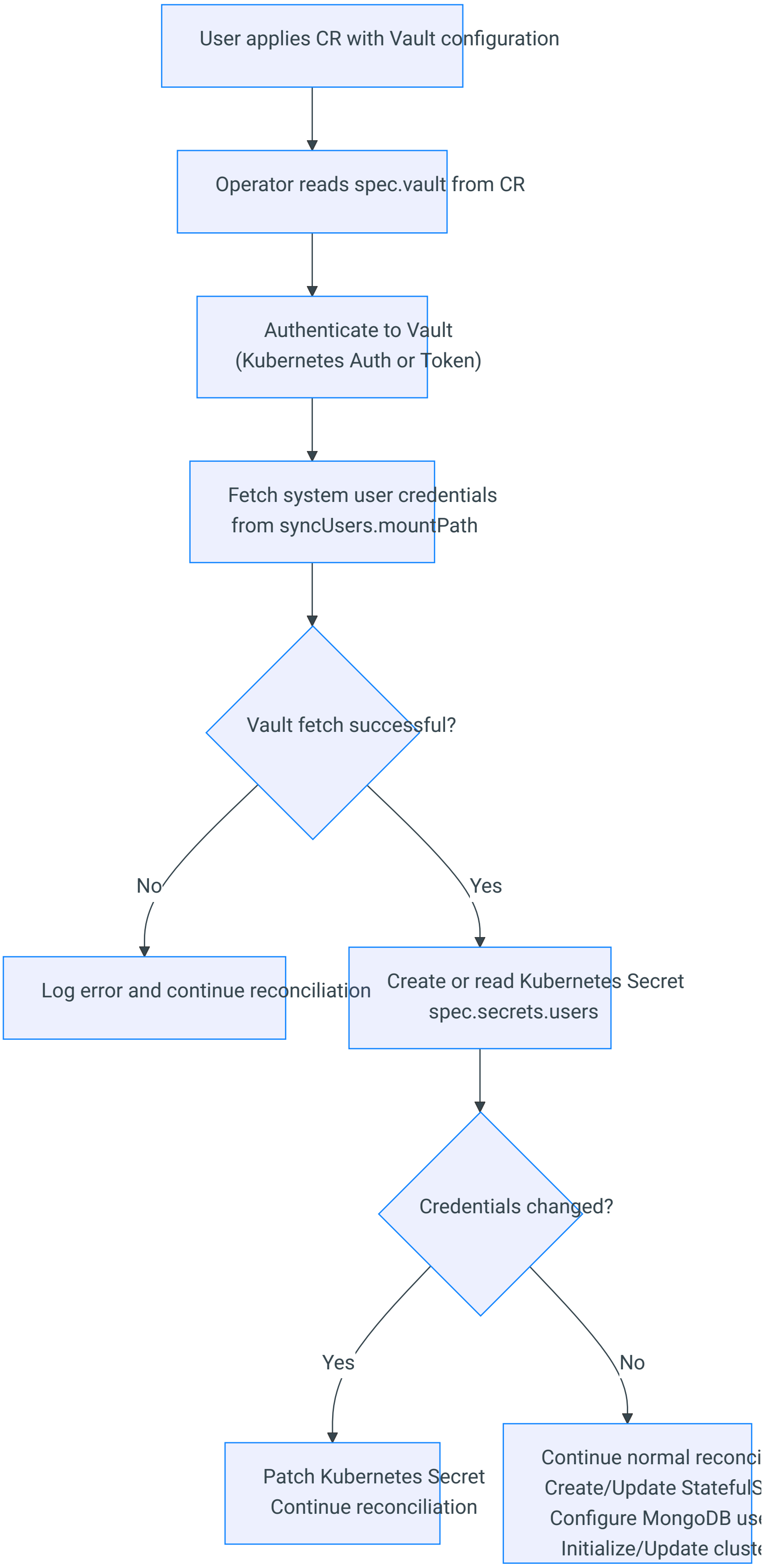
How it works

When Vault is enabled, the Operator:

- Retrieves system user passwords from Vault during cluster creation
- Creates Kubernetes Secrets using those passwords
- Sends periodic requests to Vault for updates
- Compares Vault credentials with the Secret during each reconciliation loop and patches the Secret when values differ.
- Applies its standard password rotation routine to update credentials inside the cluster
- Ensures Vault remains the authoritative source
- Continues reconciling even if Vault is temporarily unavailable

If the Operator fails to retrieve credentials from Vault and a user Secret exists in the cluster, it logs an error and continues reconciling so the cluster stays available. If no user Secret exists, the Operator blocks the cluster reconciliation.

The following diagram outlines the workflow:



Default path for user credentials

By default, the user credentials are stored at the path `secret/data/psmdb/{role}/{namespace}/{name}/users`, where:

- `{role}` is the role for the Operator that you create in Vault
- `{namespace}` is the namespace where Percona Server for MongoDB is deployed
- `{name}` is the Percona Server for MongoDB cluster name as defined in the Custom Resource

You can override this path using the `vault.syncUsers.mountPath` and `vault.syncUsers.keyPath` options in the Custom Resource.

Authentication options for Vault

The Operator can authenticate in Vault in two ways:

- **Using Kubernetes service accounts** - The Operator authenticates to Vault using its Service Account JWT. This method is recommended when Vault runs in the same Kubernetes cluster
- **Using Vault tokens** - The Operator authenticates to Vault using a pre-created Vault token. This method is used when Vault is deployed outside Kubernetes or in another cluster.

Configuration

See [Configure Vault for system user management]

7.1.4.2 Configure Vault for system user management

This document guides you through the configuration of the Operator and Vault for system user management. By default, the Operator and Vault communicate over an unencrypted HTTP protocol. You can enable encrypted HTTPS protocol with TLS as an additional security layer to protect the data transmitted between Vault and your Percona Server for MongoDB nodes. HTTPS ensures that sensitive information, such as encryption keys and secrets, cannot be intercepted or tampered with on the network.

Assumptions

1. This guide is provided as a best effort and builds upon procedures described in the official Vault documentation. Since Vault's setup steps may change in future releases, this document may become outdated; we cannot guarantee ongoing accuracy or responsibility for such changes. For the most up-to-date and reliable information, please always refer to the official Vault documentation.
2. In the following sections we deploy the Vault server in High Availability (HA) mode on Kubernetes via Helm with TLS enabled. The HA setup uses Raft storage backend and consists of 3 replicas for redundancy. Using Helm is not mandatory. Any supported Vault deployment (on-premises, in the cloud, or a managed Vault service) works as long as the Operator can reach it.
3. This guide uses Vault Helm chart version 0.30.0. You may want to change it to the required version by setting the `VAULT_HELM_VERSION` variable.

Prerequisites

Before you begin, ensure you have the following tools installed:

- `kubectl` - Kubernetes command-line interface
- `helm` - Helm package manager
- `jq` - JSON processor

Prepare your environment

1. Export the namespaces and other variables as environment variables to simplify further configuration:

```
export NAMESPACE="vault"
export CLUSTER_NAMESPACE="psmdb"
export VAULT_HELM_VERSION="0.30.0"
export SERVICE="vault"
export CSR_NAME="vault-csr"
export SECRET_NAME_VAULT="vault-secret"
export POLICY_NAME="operator"
export WORKDIR="/tmp/vault"
```

2. Create a working directory for certificate and configuration files:

```
mkdir -p /tmp/vault
```

3. It is a good practice to isolate workloads in Kubernetes using namespaces. Create namespaces with the following command:
   - For Vault server:

```
kubectl create namespace vault
```

   - For Percona Server for MongoDB cluster:

```
kubectl create namespace psmdb
```

Generate TLS certificates

To use TLS, you'll need the following certificates:

- A private key for the Vault server
- A certificate for the Vault server signed by the Kubernetes CA

- The Kubernetes CA certificate

These files store sensitive information. Make sure to keep them in a secure location.

Generate the private key

Generate a private key for the Vault server:

```
openssl genrsa -out ${WORKDIR}/vault.key 2048
```

Create the Certificate Signing Request (CSR)

A Certificate Signing Request (CSR) is a file that contains information about your server and the certificate you need. You create it using your private key, and then submit it to Kubernetes to get a certificate signed by the Kubernetes Certificate Authority (CA). The signed certificate proves your server's identity and enables secure TLS connections.

1. Create the Certificate Signing Request configuration file:

   Specify the certificate details that Kubernetes needs to sign your certificate:

   - **Request settings** (`[req]`): References the sections for certificate extensions and distinguished name. The distinguished name section is left empty as Kubernetes will populate it automatically.
   - **Certificate extensions** (`[v3_req]`): Defines how the certificate can be used. `serverAuth` allows the certificate for server authentication, while `keyUsage` specifies the cryptographic operations the certificate supports (non-repudiation, digital signature, and key encipherment).
   - **Subject Alternative Names** (`[alt_names]`): Lists all DNS names and IP addresses where your Vault service can be accessed. This includes the service name, fully qualified domain names (FQDNs) for different Kubernetes DNS contexts (namespace-scoped, cluster-scoped with `.svc`, and fully qualified with `.svc.cluster.local`), and the localhost IP address.

   ```
   cat > "${WORKDIR}/csr.conf" <<'EOF'
   [req]
   default_bits = 2048
   prompt = no
   encrypt_key = yes
   default_md = sha256
   distinguished_name = kubelet_serving
   req_extensions = v3_req
   [ kubelet_serving ]
   O = system:nodes
   CN = system:node:*.vault.svc.cluster.local
   [ v3_req ]
   basicConstraints = CA:FALSE
   keyUsage = nonRepudiation, digitalSignature, keyEncipherment, dataEncipherment
   extendedKeyUsage = serverAuth, clientAuth
   subjectAltName = @alt_names
   [alt_names]
   DNS.1 =*.vault-internal
   DNS.2 = *.vault-standby
   DNS.3 =*.vault-internal.vault.svc.cluster.local
   DNS.4 = *.vault-standby.vault.svc.cluster.local
   DNS.5 =*.vault
   DNS.6 = vault.vault.svc.cluster.local
   IP.1 = 127.0.0.1
   EOF
   ```

2. Generate the CSR. The following command creates the Certificate Signing Request file using your private key and the configuration file.

   The `-subj` parameter specifies the distinguished name directly: the Common Name (CN) identifies your Vault service using the Kubernetes node naming convention (`system:node:${SERVICE}.${NAMESPACE}.svc`), and the Organization (O) field is set to `system:nodes`, which Kubernetes requires to recognize and sign the certificate. The command combines these subject fields with the certificate extensions defined in the configuration file to produce the complete CSR.

   ```
   openssl req -new -key $WORKDIR/vault.key \
     -subj "/CN=system:node:${SERVICE}.${NAMESPACE}.svc;/O=system:nodes" \
     -out $WORKDIR/server.csr -config $WORKDIR/csr.conf
   ```

Issue the certificate

To get your certificate signed by Kubernetes, you need to submit the CSR through the Kubernetes API. The CSR file you generated with OpenSSL must be wrapped in a Kubernetes CertificateSigningRequest resource.

1. Create the CSR YAML file to send it to Kubernetes:

   This YAML file creates a Kubernetes CertificateSigningRequest object that contains your CSR. The file embeds the base64-encoded CSR content and specifies:

   - The signer name (`kubernetes.io/kubelet-serving`) that tells Kubernetes which CA should sign the certificate
   - The groups field (`system:authenticated`) that identifies who can approve this CSR
   - The certificate usages that define how the certificate can be used (digital signature, key encipherment, and server authentication)

   ```
   cat > $WORKDIR/csr.yaml <<EOF
   apiVersion: certificates.k8s.io/v1
   kind: CertificateSigningRequest
   metadata:
     name: ${CSR_NAME}
   spec:
     groups:
     - system:authenticated
     request: $(cat $WORKDIR/server.csr | base64 | tr -d '\n')
     signerName: kubernetes.io/kubelet-serving
     usages:
     - digital signature
     - key encipherment
     - server auth
   EOF
   ```

2. Create the CertificateSigningRequest (CSR) object:

   ```
   kubectl create -f ${WORKDIR}/csr.yaml
   ```

3. Approve the CSR in Kubernetes:

   ```
   kubectl certificate approve ${CSR_NAME}
   ```

4. Confirm the certificate was issued:

   ```
   kubectl get csr ${CSR_NAME}
   ```

   Sample output

   ```
   NAME        AGE   SIGNERNAME                      REQUESTOR       REQUESTEDDURATION   CONDITION
   vault-csr   16s   kubernetes.io/kubelet-serving   minikube-user   <none>              Approved,Issued
   ```

Retrieve the certificates

After Kubernetes approves and signs your CSR, you need to retrieve the signed certificate and the Kubernetes CA certificate. These certificates are required to configure TLS for your Vault server.

1. Retrieve the signed certificate from the CertificateSigningRequest object. The certificate is base64-encoded in Kubernetes, so you decode it and save it to a file.

   ```
   kubectl get csr ${CSR_NAME} -o jsonpath='{.status.certificate}' | base64 -d > $WORKDIR/vault.crt
   ```

2. Retrieve Kubernetes CA certificate:

   This command retrieves the Kubernetes cluster's Certificate Authority (CA) certificate from your `kubeconfig` file. The CA certificate is needed to verify that the signed certificate is valid and was issued by the Kubernetes CA. The command uses `kubectl config view` with flags to get the raw, flattened configuration and extract the CA certificate data, which is also base64-encoded.

```
kubectl config view \
  --raw \
  --minify \
  --flatten \
  -o jsonpath='{.clusters[].cluster.certificate-authority-data}' \
  | base64 -d > ${WORKDIR}/vault.ca
```

Store certificates in Kubernetes secrets

Create a TLS secret in Kubernetes to store the certificates and key:

```
kubectl create secret generic ${SECRET_NAME_VAULT} \
  --namespace ${NAMESPACE} \
  --from-file=vault.key=$WORKDIR/vault.key \
  --from-file=vault.crt=$WORKDIR/vault.crt \
  --from-file=vault.ca=$WORKDIR/vault.ca
```

Install Vault with TLS

For this setup, we install Vault in Kubernetes using the [Helm 3 package manager]() in High Availability (HA) mode with Raft storage backend and with TLS enabled.

1. Add and update the Vault Helm repository:

```
helm repo add hashicorp https://helm.releases.hashicorp.com
helm repo update
```

2. Install Vault with TLS enabled:

```
helm upgrade --install ${SERVICE} hashicorp/vault \
  --disable-openapi-validation \
  --version ${VAULT_HELM_VERSION} \
  --namespace ${NAMESPACE} \
  --set "global.enabled=true" \
  --set "global.tlsDisable=false" \
  --set "global.platform=kubernetes" \
  --set server.extraEnvironmentVars.VAULT_CACERT=/vault/userconfig/${SECRET_NAME_VAULT}/vault.ca \
  --set "server.extraEnvironmentVars.VAULT_TLSCERT=/vault/userconfig/${SECRET_NAME_VAULT}/vault.crt" \
  --set "server.extraEnvironmentVars.VAULT_TLSKEY=/vault/userconfig/${SECRET_NAME_VAULT}/vault.key" \
  --set "server.volumes[0].name=userconfig-${SECRET_NAME_VAULT}" \
  --set "server.volumes[0].secret.secretName=${SECRET_NAME_VAULT}" \
  --set "server.volumes[0].secret.defaultMode=420" \
  --set "server.volumeMounts[0].mountPath=/vault/userconfig/${SECRET_NAME_VAULT}" \
  --set "server.volumeMounts[0].name=userconfig-${SECRET_NAME_VAULT}" \
  --set "server.volumeMounts[0].readOnly=true" \
  --set "server.ha.enabled=true" \
  --set "server.ha.replicas=3" \
  --set "server.ha.raft.enabled=true" \
  --set "server.ha.raft.setNodeId=true" \
  --set-string "server.ha.raft.config=cluster_name = \"vault-integrated-storage\"
ui = true
listener \"tcp\" {
  tls_disable = 0
  address = \"[::]:8200\"
  cluster_address = \"[::]:8201\"
  tls_cert_file = \"/vault/userconfig/${SECRET_NAME_VAULT}/vault.crt\"
  tls_key_file  = \"/vault/userconfig/${SECRET_NAME_VAULT}/vault.key\"
  tls_client_ca_file = \"/vault/userconfig/${SECRET_NAME_VAULT}/vault.ca\"
}
storage \"raft\" {
  path = \"/vault/data\"
}
disable_mlock = true
service_registration \"kubernetes\" {}"
```

This command does the following:

- Installs HashiCorp Vault in High Availability (HA) mode with secure TLS enabled in your Kubernetes cluster.

- Configures Vault pods to use certificates from a Kubernetes Secret via volume mounts for secure HTTPS communication between Vault and clients.
- Sets up Raft as the backend storage with three replicas for fault tolerance, and configures the Vault TCP listener to enforce TLS with your specified certificate files.

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 $NAMESPACE get pod -l app.kubernetes.io/name=${SERVICE} -o jsonpath='{range .items[*]}{.metadata.name}{"\n"}
{end}'
```

Sample output

```
vault-0
vault-1
vault-2
```

Initialize and unseal Vault

1. After Vault is installed, you need to initialize it. Run the following command to initialize the first pod:

```
kubectl exec -it pod/vault-0 -n $NAMESPACE -- vault operator init -key-shares=1 -key-threshold=1 -format=json >
${WORKDIR}/vault-init
```

The command does the following:

- Connects to the Vault Pod
- Initializes Vault server with TLS enabled
- Creates 1 unseal key share which is required to unseal the server
- Outputs the init response to a local file. The file includes unseal keys and root token.

2. 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[]" < ${WORKDIR}/vault-init)
```

Now, unseal Vault. Run the following command:

```
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                  1.19.0
Build Date               2025-03-04T12:36:40Z
Storage Type             raft
Cluster Name             vault-integrated-storage
Cluster ID               ed275c91-e227-681b-5aaa-f7a9fc19e37e
Removed From Cluster     false
HA Enabled               true
HA Cluster               <https://vault-0.vault-internal:8201>
HA Mode                  active
Active Since             2025-12-15T13:36:42.542059059Z
Raft Committed Index     37
Raft Applied Index       37
```

3. Add the remaining Pods to the Vault cluster. If you have another secret name, replace the `vault-secret` with your value in the following for loop:

```
for POD in vault-1 vault-2; do
  kubectl -n "$NAMESPACE" exec $POD -- sh -c '
    vault operator raft join -address=https://${HOSTNAME}.vault-internal:8200 \
      -leader-ca-cert="$(cat /vault/userconfig/vault-secret/vault.ca)" \
      -leader-client-cert="$(cat /vault/userconfig/vault-secret/vault.crt)" \
      -leader-client-key="$(cat /vault/userconfig/vault-secret/vault.key)" \
      https://vault-0.vault-internal:8200;
  '
done
```

The command connects to each Vault Pod (`vault-1` and `vault-2`) and issues the `vault operator raft join` command, which: * Joins the Pods to the Vault Raft cluster, enabling HA mode. * Uses the necessary TLS certificates to securely connect to the cluster leader (`vault-0`). * Ensures all nodes participate in the Raft consensus and share storage responsibilities.

Sample output

```
Key       Value
---       -----
Joined    true
```

4. Unseal the remaining Pods. Use this for loop:

```
for POD in vault-1 vault-2; do
    kubectl -n "$NAMESPACE" exec $POD -- sh -c "
        vault operator unseal \"$unsealKey\"
    "
done
```

Expected output

```
Key                      Value
---                      -----
Seal Type                shamir
Initialized              true
Sealed                   true
Total Shares             1
Threshold                1
Unseal Progress          0/1
Unseal Nonce             n/a
Version                  1.19.0
Build Date               2025-03-04T12:36:40Z
Storage Type             raft
Removed From Cluster     false
HA Enabled               true
```

Configure Vault

Enable secrets engine

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.

Run the following command on a leader node. The remaining ones will synchronize from the leader.

1. Extract the Vault root token from the file where you saved the init response output:

   ```
   cat ${WORKDIR}/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 `secret` mount-path:

   ```
   vault secrets enable --version=2 -path=secret kv
   ```

   Sample output

   ```
   Success! Enabled the kv secrets engine at: secret/
   ```

5. Exit the Vault pod:

   ```
   exit
   ```

Create the access policy for the Operator

Create a policy for accessing the kv engine path and define the required permissions in the `capabilities` parameter:

```
kubectl -n "$NAMESPACE" exec vault-0 -- sh -c "vault policy write $POLICY_NAME - << 'EOF'
path \"secret/psmdb/operator/*\" {
  capabilities = [\"read\"]
}

path \"secret/data/psmdb/operator/*\" {
  capabilities = [\"read\"]
}
EOF"
```

Configure authentication in Vault

Depending on your Vault setup, configure one of the authentication methods:

- [Kubernetes Service Account authentication](#)
- [Authentication with Vault token](#)

Kubernetes Service Account authentication

When the Operator authenticates to Vault using the Kubernetes authentication method, it presents a JWT token issued by the Kubernetes API server.

Vault must verify that JWT. To do that, Vault needs two things:

1. The Kubernetes API server URL, so it can call the TokenReview API.
2. The JWT issuer URL, so it can validate the token signature and claims.

The following steps guide you through the setup.

1. Enable the Kubernetes authentication method in Vault:

   ```
   kubectl -n "$NAMESPACE" exec vault-0 -- sh -c "vault auth enable kubernetes"
   ```

   Expected output

   ```
   Success! Enabled kubernetes auth method at: kubernetes/
   ```

2. Configure the Kubernetes endpoint in Vault:

   ```
   kubectl -n "$NAMESPACE" exec vault-0 -- sh -c "vault write auth/kubernetes/config
   kubernetes_host=https://$KUBERNETES_SERVICE_HOST:$KUBERNETES_SERVICE_PORT"
   ```

   Expected output

   ```
   Success! Data written to: auth/kubernetes/config
   ```

3. Discover the service account issuer and export it as an environment variable:

   ```
   export ISSUER="$(kubectl get --raw /.well-known/openid-configuration | jq -r '.issuer')"
   ```

   Sample output for GKE

   ```
   echo $ISSUER
   https://container.googleapis.com/v1/projects/<project-ID>/locations/<region>/clusters/<kubernetes-cluster-name>
   ```

4. Create a Vault role bound to the Operator Service Account:

   ```
   kubectl -n "$NAMESPACE" exec vault-0 -- sh -c "vault write auth/kubernetes/role/operator \
    bound_service_account_names=percona-server-mongodb-operator \
    bound_service_account_namespaces=$CLUSTER_NAMESPACE \
    policies=operator \
    audience="$ISSUER" \
    ttl=1h"
   ```

   Expected output

   ```
   Success! Data written to: auth/kubernetes/role/operator
   ```

5. Create a RoleBinding resource for your cluster. This grants Vault permission to call the Kubernetes TokenReview API (`system:auth-delegator`), which is required to validate the Operator service account JWT during login. The default service account name is `percona-server-mongodb-operator`.

```
kubectl apply -f - <<EOF
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
    name: role-tokenreview-binding
    namespace: $CLUSTER_NAMESPACE
roleRef:
    apiGroup: rbac.authorization.k8s.io
    kind: ClusterRole
    name: system:auth-delegator
subjects:
    - kind: ServiceAccount
      name: percona-server-mongodb-operator
      namespace: $CLUSTER_NAMESPACE
EOF
```

6. Check that a RoleBinding is created:

```
kubectl get rolebindings -n $CLUSTER_NAMESPACE
```

Sample output

```
NAME                                          ROLE                                  AGE
role-tokenreview-binding                      ClusterRole/system:auth-delegator     16s
service-account-percona-server-mongodb-operator   Role/percona-server-mongodb-operator   4m39s
```

Authentication with Vault token

1. Create a non-root token in Vault using the [access policy](#) you created before:

```
kubectl -n "${NAMESPACE}" exec pod/vault-0 -- sh -c "vault token create -policy="operator" -format=json" >
"${WORKDIR}/vault-token.json"
```

2. Export a token as an environment variable:

```
export NEW_TOKEN=$(jq -r '.auth.client_token' "${WORKDIR}/vault-token.json")
```

3. Create a Kubernetes Secret with the token for the namespace where Percona Server for MongoDB is deployed

```
kubectl create secret generic vault-sync-secret \
 --from-literal=token="$NEW_TOKEN" -n $CLUSTER_NAMESPACE
```

Sample output

```
secret/vault-sync-secret created
```

Upload data to Vault

The default path in Vault to store user passwords is `secret/data/psmdb/{role}/{namespace}/{name}/users`, where:

- `{role}` is the role for the Operator that you create in Vault
- `{namespace}` is the namespace where Percona Server for MongoDB is deployed
- `{name}` is the Percona Server for MongoDB cluster name as defined in the Custom Resource

For testing purposes, let's insert the credentials from the sample `deploy/secrets.yaml` file in the JSON format.

1. Create a file in JSON format:

```
cat <<EOF | tee $WORKDIR/passwords.json
{
    "MONGODB_USER_ADMIN_USER": "userAdmin",
    "MONGODB_USER_ADMIN_PASSWORD": "strongPass",
    "MONGODB_DATABASE_ADMIN_USER": "dbAdmin",
    "MONGODB_DATABASE_ADMIN_PASSWORD": "strongPass",
    "MONGODB_BACKUP_USER": "backup",
    "MONGODB_BACKUP_PASSWORD": "strongPass",
    "MONGODB_CLUSTER_ADMIN_USER": "clusterAdmin",
    "MONGODB_CLUSTER_ADMIN_PASSWORD": "strongPass",
    "MONGODB_CLUSTER_MONITOR_USER": "clusterMonitor",
    "MONGODB_CLUSTER_MONITOR_PASSWORD": "strongPass"
}
EOF
```

2. Copy the file to a folder inside the Vault Pod:

```
kubectl cp $WORKDIR/passwords.json $NAMESPACE/vault-0:/tmp/passwords.json
```

3. Upload the file in Vault:

```
kubectl exec -it vault-0 -n $NAMESPACE -- /bin/sh -c "vault kv put -mount=secret psmdb/operator/$CLUSTER_NAMESPACE/my-cluster-name/users/ @/tmp/passwords.json"
```

Sample output

```
========================== Secret Path ==========================
secret/data/psmdb/operator/psmdb/my-cluster-name/users/databaseAdmin
======= Metadata =======
Key                Value
---                -----
created_time       2026-02-05T14:32:48.376356584Z
custom_metadata    <nil>
deletion_time      n/a
destroyed          false
version            1
```

4. Verify the insertion:

```
kubectl exec -it vault-0 -n $NAMESPACE -- /bin/sh -c "vault kv get -mount=secret psmdb/operator/$CLUSTER_NAMESPACE/my-cluster-name/users/databaseAdmin"
```

Sample output

```
========================== Secret Path ==========================
secret/data/psmdb/operator/psmdb/my-cluster-name/users/databaseAdmin
======= Metadata =======
Key                Value
---                -----
created_time       2026-02-06T11:22:18.96780783Z
custom_metadata    <nil>
deletion_time      n/a
destroyed          false
version            2

================== Data ==================
Key                                 Value
---                                 -----
MONGODB_BACKUP_PASSWORD             strongPass
MONGODB_BACKUP_USER                 backup
MONGODB_CLUSTER_ADMIN_PASSWORD      strongPass
MONGODB_CLUSTER_ADMIN_USER          clusterAdmin
MONGODB_CLUSTER_MONITOR_PASSWORD    strongPass
MONGODB_CLUSTER_MONITOR_USER        clusterMonitor
MONGODB_DATABASE_ADMIN_PASSWORD     newStrongPass
MONGODB_DATABASE_ADMIN_USER         dbAdmin
MONGODB_USER_ADMIN_PASSWORD         strongPass
MONGODB_USER_ADMIN_USER             userAdmin
```

See the Vault documentation for more usage examples of manipulating the data.

Configure the Operator

Now you must configure the Operator to communicate with Vault.

Create a Secret for Vault (optional)

You can specify what TLS certificate the Operator should use when it connects to Vault over TLS. To do that, you must create a Secret object with this data and reference it in the `tlsSecret` option in the Custom Resource.

If you don't set the `tlsSecret`, the Operator authenticates to Vault over an unencrypted HTTP protocol.

Create the Secret with the following command. Specify the path to your TLS certificate:

```
kubectl create secret generic my-tls-vault-secret --from-file=ca.crt=$WORKDIR/vault.ca -n $CLUSTER_NAMESPACE
```

Reference Vault configuration in the Custom Resource

Now, configure Vault in the Custom Resource to make the Operator aware of it.

Specify the following information:

- `endpointURL`: Where your Vault server is running.
- `tlsSecret` (optional): The Secret name that contains TLS certificates for accessing Vault via TLS. This is the Secret object that you created at the previous step
- Specify the Vault-related information for the `syncUsers` subsection:
- `role`: The role you have created for the Operator in Vault
- `mountPath`: The mount path at which you have enabled the secrets engine
- `keyPath`: The path to store user passwords
- `tokenSecret`: The Secret object that contains the authentication token. Specify it if you configured [authentication with Vault tokens]
- Add the Vault configuration to your Custom Resource.

Kubernetes authentication

```
spec:
  vault:
    endpointUrl: https://vault.vault.svc.cluster.local:8200
    tlsSecret: my-tls-vault-secret
    syncUsers:
      role: operator
      mountPath: /secret/operator/psmdb/system
      keyPath: psmdb/operator/namespace/my-cluster-name/users
```

Authentication with Vault token

```
spec:
  vault:
    endpointUrl: https://vault.vault.svc.cluster.local:8200
    tlsSecret: my-tls-vault-secret
    syncUsers:
      role: operator
      mountPath: /secret/operator/psmdb/system
      keyPath: psmdb/operator/namespace/my-cluster-name/users
      tokenSecret: vault-token-operator
```

Authenticate in Percona Server for MongoDB to verify password management

To verify that the Operator retrieves passwords from Vault, we'll do the following:

- deploy a separate `mongo` client Pod
- authenticate in Percona Server for MongoDB with the current password
- change the password in Vault

- authenticate with the new password

Here's how to do it:

1. Get the user credentials from the Secret `<cluster-name>-secrets`. Run the following commands to retrieve the username and password for the database admin user:

```
kubectl get secret my-cluster-name-secrets -n $CLUSTER_NAMESPACE -o yaml -o jsonpath='{.data.MONGODB_DATABASE_ADMIN_USER}'
| base64 --decode | tr '\n' ' ' && echo " "
kubectl get secret my-cluster-name-secrets -n $CLUSTER_NAMESPACE -o yaml -o
jsonpath='{.data.MONGODB_DATABASE_ADMIN_PASSWORD}' | base64 --decode | tr '\n' ' ' && echo " "
```

2. Spin up a `mongo` client Pod:

```
kubectl -n $CLUSTER_NAMESPACE run -i --rm --tty percona-client --image=percona/percona-server-mongodb:8.0.19-7 --
restart=Never -- bash -il
```

3. Inside the Pod, run the following command:

sharding is on

```
mongosh "mongodb://<databaseAdminUser>:<databaseAdminPassword>@my-cluster-name-mongos.<namespace>.svc.cluster.local/admin?
ssl=false"
```

sharding is off

```
mongosh  "mongodb://<databaseAdminUser>:<databaseAdminPassword>@my-cluster-name-rs0.<namespace>.svc.cluster.local/admin?
replicaSet=rs0&ssl=false"
```

Expected output

```
.....
[direct: mongos] admin>
```

4. Update the password for the `MONGODB_DATABASE_ADMIN_PASSWORD` user:

```
kubectl exec -it vault-0 -n $NAMESPACE -- /bin/sh -c "vault kv patch -mount=secret psmdb/operator/$CLUSTER_NAMESPACE/my-
cluster-name/users MONGODB_DATABASE_ADMIN_PASSWORD="newStrongPass""
```

5. Repeat steps 2-3 and connect to Percona Server for MongoDB with this new password. As a result you should be successfully authenticated.

7.2 Disable authentication in Percona Server for MongoDB

You can disable authentication in Percona Server for MongoDB clusters. This is useful for development, testing, or migration scenarios where you need access to the database without authentication.

> **Warning**
>
> Disabling authentication removes an important security layer. Only disable authentication in trusted environments, such as isolated development or testing clusters. Never disable authentication in production environments without proper security measures.

When to disable authentication

You might want to disable authentication in the following scenarios:

- **Development and testing**: When you need quick access to the database during development without managing user credentials
- **Migration and data import**: When migrating data from external MongoDB instances that don't use authentication
- **Troubleshooting**: When debugging authentication-related issues
- **Legacy application compatibility**: When working with applications that don't support MongoDB authentication

Considerations

Consider the following for disabling authentication

- You must disable TLS encryption first. You cannot disable authentication while TLS is enabled.
- You must disable authentication for each component of your Percona Server for MongoDB cluster separately. This mean you must modify the configuration for each shard and the config server replica set.

Disable authentication for a new cluster

To disable authentication for a new cluster, follow these steps:

1. Export the namespace where you will deploy your cluster as an environment variable

   ```
   export NAMEPSPACE=my-namespace
   ```

2. Edit the `deploy/cr.yaml` Custom Resource manifest and set the following configuration:
   - Set `tls.mode` to `disabled`
   - Set `unsafeFlags.tls` to `true`
   - Add the `security.authorization: disabled` option to the `configuration` section for each component

Non-sharded cluster

```
...
spec:
  ...
  unsafeFlags:
    tls: true
  tls:
    mode: disabled
  replsets:
    - name: rs0
      size: 3
      configuration: |
        security:
          authorization: disabled
  ...
```

Sharded cluster

For sharded clusters, you must disable authentication for each component separately: replica sets, config server replica set, and mongos.

```
...
spec:
  ...
  unsafeFlags:
    tls: true
  tls:
    mode: disabled
  replsets:
    - name: rs0
      size: 3
      configuration: |
        security:
          authorization: disabled
  sharding:
    configsvrReplSet:
      configuration: |
        security:
          authorization: disabled
    mongos:
      configuration: |
        security:
          authorization: disabled
  ...
```

Warning

Always disable authentication for all components. If you disable authentication for replica sets but not for the config server replica set in a sharded cluster, the cluster enters an error state.

3. Apply the manifest.

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

Disable authentication for a running cluster

To disable authentication for a running cluster, you must pause the cluster, update the configuration, and then unpause it.

1. Export the namespace where your cluster is running as an environment variable:

```
export NAMEPSPACE=my-namespace
```

2. Pause the cluster. Edit the `deploy/cr.yaml` Custom Resource manifest:

```
spec:
  .......
  pause: true
```

3. Apply the configuration:

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

4. Wait for the cluster to be paused. Check the status with the `kubectl get psmdb` command:

```
kubectl get psmdb -n $NAMESPACE
```

Expected output

```
NAME              ENDPOINT                                              STATUS   AGE
my-cluster-name   my-cluster-name-mongos.default.svc.cluster.local:27017   paused   3m
```

5. Disable TLS and authentication by setting the following configuration in the `deploy/cr.yaml` Custom Resource manifest:

Non-sharded cluster

```
...
spec:
  ...
  unsafeFlags:
    tls: true
  tls:
    mode: disabled
  replsets:
    - name: rs0
      size: 3
      configuration: |
        security:
          authorization: disabled
  ...
```

Sharded cluster

```
...
spec:
  ...
  unsafeFlags:
    tls: true
  tls:
    mode: disabled
  replsets:
    - name: rs0
      size: 3
      configuration: |
        security:
          authorization: disabled
  sharding:
    configsvrReplSet:
      configuration: |
        security:
          authorization: disabled
    mongos:
      configuration: |
        security:
          authorization: disabled
  ...
```

6. Apply the changes:

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

7. Resume the cluster. Set the `pause` key to `false` in the Custom Resource manifest:

```
spec:
......
   pause: false
```

8. Apply the changes:

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

9. Wait for the cluster to be resumed. Check the status with the `kubectl get psmdb` command.

Re-enable authentication

To re-enable authentication, repeat the same steps as for disabling it (pause the cluster, apply config changes, resume), but **remove** any `security.authorization: disabled` lines from your `deploy/cr.yaml` manifest. Optionally, set `tls.mode` back to your preferred state if you also want to re-enable TLS. Apply the manifest and unpause the cluster.

Verify authentication is disabled

After disabling authentication, you can connect to the cluster without providing credentials:

Non-sharded cluster

```
mongosh "mongodb://my-cluster-name-rs0.<namespace>.svc.cluster.local:27017/?replicaSet=rs0"
```

Sharded cluster

```
mongosh "mongodb://my-cluster-name-mongos.<namespace>.svc.cluster.local:27017"
```

If authentication is disabled, you should be able to connect without providing a username or password.

7.3 Changing MongoDB options

When you deploy a new Percona Server for MongoDB cluster, the Operator spins it up with the set of defaults that ensure its correct operation. However, your application may need additional configuration of MongoDB. You can define this configuration using the `mongod.conf` configuration file options. You can configure mongod Pods, mongos Pods and config server replica set Pods separately, based on your requirements.
Then you pass these options to MongoDB instances in the cluster in one of the following ways:

- Edit the `deploy/cr.yaml` file
- Use a ConfigMap
- Use a Secret object

Note that you can't change options that may break the behavior of the Operator. For example, TLS/SSL options. If you try changing such options, your changes will be ignored.

Edit the `deploy/cr.yaml` file

You can add MongoDB configuration options to the following keys of the `deploy/cr.yaml`:

- replsets.configuration
- sharding.mongos.configuration
- sharding.configsvrReplSet.configuration

Example

This example shows how to enable rate limit for database profiler and define the default verbosity level for system log:

```
spec:
  ...
  replsets:
    - name: rs0
      size: 3
      configuration: |
        operationProfiling:
          mode: slowOp
        systemLog:
          verbosity: 1
      ...
```

Find the complete list of options in the official manual. Also refer to these pages in Percona Server for MongoDB documentation:

- Profiling rate limit
- Percona memory engine
- Data-at-rest encryption
- Log redaction
- Audit logging.

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.

You should give the ConfigMap a specific name, which is composed of your cluster name and a specific suffix:

- `my-cluster-name-rs0-mongod` for the Replica Set (mongod) Pods,
- `my-cluster-name-cfg-mongod` for the Config Server Pods,
- `my-cluster-name-mongos` for the mongos Pods,

> **Note**
>
> To find the cluster name, you can use the following command:
>
> ```
> kubectl get psmdb
> ```

For example, let’s define a `mongod.conf` configuration file and put there several MongoDB options we used in the previous example:

```
operationProfiling:
  mode: slowOp
systemLog:
  verbosity: 1
```

You can create a ConfigMap from the `mongod.conf` file with the `kubectl create configmap` command. It has the following syntax:

```
kubectl create configmap <configmap-name> <resource-type=resource-name>
```

The following example defines `my-cluster-name-rs0-mongod` as the ConfigMap name and the `mongod.conf` file as the data source:

```
kubectl create configmap my-cluster-name-rs0-mongod --from-file=mongod.conf=mongod.conf
```

To view the created ConfigMap, use the following command:

```
kubectl describe configmaps my-cluster-name-rs0-mongod
```

> **Note**
>
> Do not forget to restart Percona Server for MongoDB to ensure the cluster has updated the configuration (see details on how to connect in the Install Percona Server for MongoDB on Kubernetes page).

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 a specific suffix:

- `my-cluster-name-rs0-mongod` for the Replica Set Pods,
- `my-cluster-name-cfg-mongod` for the Config Server Pods,
- `my-cluster-name-mongos` for the mongos Pods,

> **Note**
>
> To find the cluster name, you can use the following command:
>
> ```
> kubectl get psmdb
> ```

Configuration options should be put inside a specific key:

- `data.mongod` key for Replica Set (mongod) and Config Server Pods,
- `data.mongos` key for mongos Pods.

Actual options should be encoded with Base64.

For example, let’s define a `mongod.conf` configuration file and put there several MongoDB options we used in the previous example:

```
operationProfiling:
  mode: slowOp
systemLog:
  verbosity: 1
```

You can get a Base64 encoded string from your options via the command line as follows:

in Linux

```
cat mongod.conf | base64 --wrap=0
```

in macOS

```
cat mongod.conf | base64
```

Note

Similarly, you can read the list of options from a Base64 encoded string:

```
echo "ICAgICAgb3BlcmF0aW9uUHJvZmlsaW5nOgogICAgICAgIG1vZGU6IHNsb3dPc\
AogICAgICBzeXN0ZW1Mb2c6CiAgICAgICAgdmVyYm9zaXR5OiAxCg==" | base64 --decode
```

Finally, use a yaml file to create the Secret object. For example, you can create a `deploy/my-mongod-secret.yaml` file with the following contents:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-cluster-name-rs0-mongod
data:
  mongod.conf: "ICAgICAgb3BlcmF0aW9uUHJvZmlsaW5nOgogICAgICAgIG1vZGU6IHNsb3dPc\
   AogICAgICBzeXN0ZW1Mb2c6CiAgICAgICAgdmVyYm9zaXR5OiAxCg=="
```

When ready, apply it with the following command:

```
kubectl create -f deploy/my-mongod-secret.yaml
```

Note

Do not forget to restart Percona Server for MongoDB to ensure the cluster has updated the configuration (see details on how to connect in the Install Percona Server for MongoDB on Kubernetes page).

7.4 Change Percona Backup for MongoDB configuration

The Operator configures Percona Backup for MongoDB (PBM) with the set of default parameters that ensure its correct operation. However, you may want to fine-tune PBM configuration to meet your specific needs. For example, adjust the node priority for backups to use a specific node. Or configure the parallel download from the storage for a physical restore.

To change PBM configuration, edit the `deploy/cr.yaml` Custom Resource manifest and specify the options in the `backup.configuration` subsection.

Examples

This example shows how to adjust the node priority for backups:

```
spec:
  backup:
    ...
    configuration:
      backupOptions:
        priority:
          "cluster1-rs0-0.cluster1-rs0.psmdb.svc.cluster.local:27017": 2.5
          "cluster1-rs0-1.cluster1-rs0.psmdb.svc.cluster.local:27017": 2.5
```

This example shows how to configure the parallel data download for physical restores:

```
spec:
  backup:
    ...
    configuration:
      restoreOptions:
        numDownloadWorkers: 4
        maxDownloadBufferMb: 0
        downloadChunkMb: 32
```

Apply the manifest to pass your configuration to the Operator:

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

Refer to the Custom Resource options for the full list of available options.

7.5 Extend startup logic for Percona Server for MongoDB components

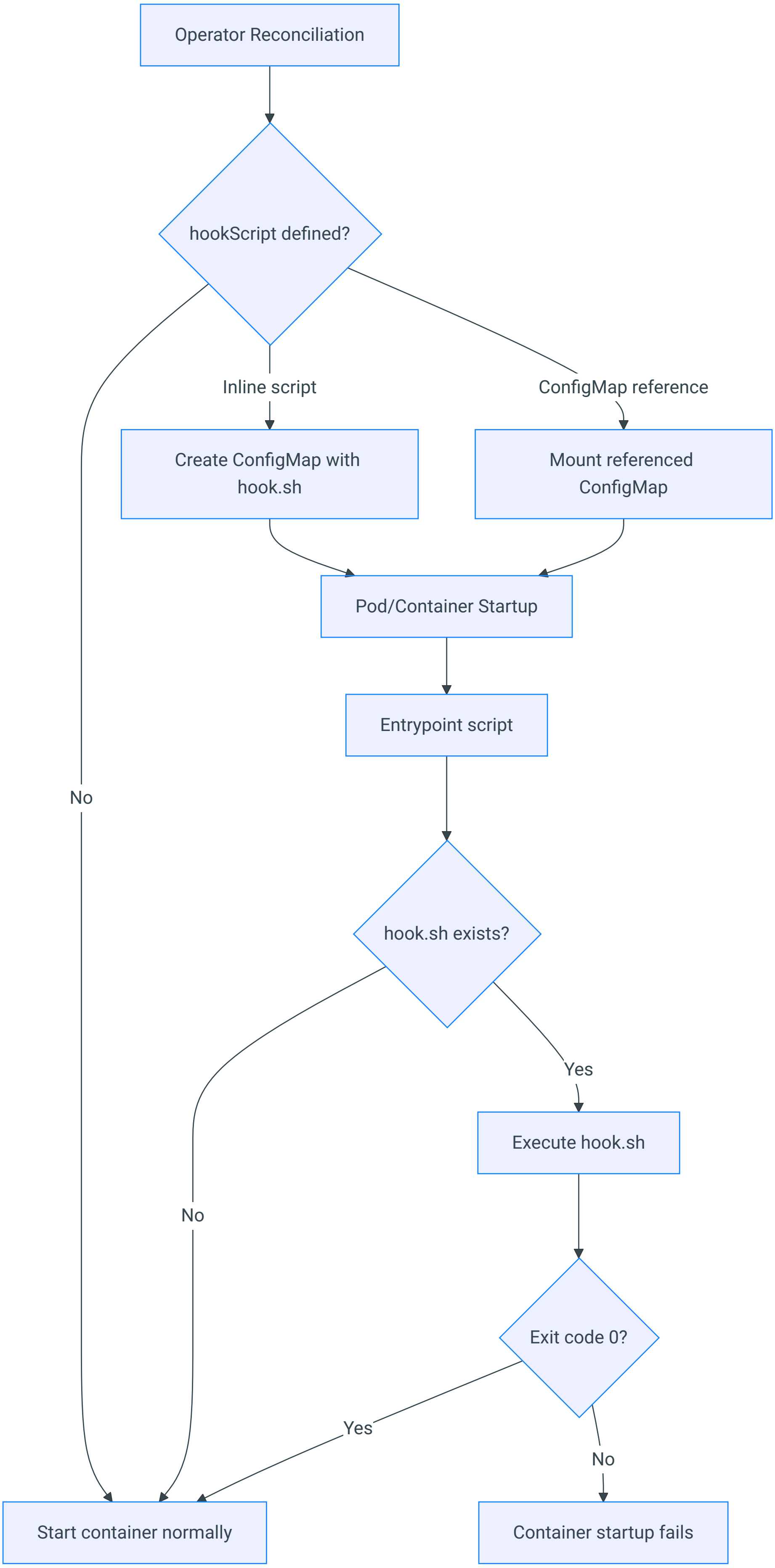
The Operator supports the `hookScript` feature that enables you to run custom shell scripts before MongoDB or PBM agent starts.

Using hook scripts, you can tailor startup behavior to your infrastructure and policies. For example, when you need to:

- Configure Fluent Bit logging for your MongoDB pods.
- Apply security hardening or compliance checks.
- Register nodes in a configuration management database (CMDB), monitoring, or audit systems.
- Modify configuration files dynamically before startup.
- Run specific pre-backup logic for PBM agent.

Hook scripts run inside the container, immediately before the main process (`mongod`, `mongos`, or `pbm-agent`) starts.

How `hookScript` works



1. You define a script either directly in the Custom Resource (CR) or reference a ConfigMap.
2. The Operator creates or mounts a ConfigMap containing `hook.sh`.

3. The container entrypoint (`ps-entry.sh` or `pbm-entry.sh`) checks for the script.
4. If found, it executes `hook.sh` before starting the main process.
5. If the script exits with a non-zero code, the container startup fails.

Where `hookScript` works

You can define hook scripts for:

- Regular replica set members (secondaries, primaries, and config servers).
- Hidden and non-voting nodes.
- Arbiters.
- Mongos nodes.
- PBM backup agent.

Configuration examples

For example, you want to capture the startup time of the container to observe when a `mongod` or `pbm-agent` container was last restarted to troubleshoot issues. The following steps show how to do it by defining the script in the Custom Resource directly and using a ConfigMap.

Directly in the Custom Resource

1. Edit the `deploy/cr.yaml` Custom Resource manifest and define the hook script in the `hookScript` field.

```
spec:
  replicaSets:
    - name: rs0
      hookScript:
        script: |
          #!/usr/bin/env bash
          echo "Startup time: $(date)" > /tmp/date
```

2. Apply the configuration changes.

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

3. Check the startup time of the container.

```
kubectl exec -it <container-name> -n <namespace> -- cat /tmp/date
```

Using a ConfigMap

1. Create a ConfigMap configuration file with the hook script:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: hookscript-configmap
data:
  hook.sh: |
    #!/usr/bin/env bash
    echo "Startup time: $(date)" > /tmp/date
```

2. Create a ConfigMap object in the Kubernetes cluster.

```
kubectl create -f hook-script.yaml -n <namespace>
```

3. Reference the ConfigMap in the Custom Resource.

```
spec:
  replicaSets:
    - name: rs0
      hookScript:
        configMapRef:
          name: hookscript-configmap
```

4. Apply the configuration changes.

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

5. Check the startup time of the container.

```
kubectl exec -it <container-name> -n <namespace> -- cat /tmp/date
```

Best practices

- Keep scripts short and deterministic.
- Validate scripts in a test environment before production.
- Use exit 1 to intentionally block startup if a requirement is not met.
- Avoid long-running operations; they delay Pod readiness.

- Store complex scripts in ConfigMaps for version control.

7.6 Binding Percona Server for MongoDB components to Specific Kubernetes/OpenShift Nodes

The operator does a good job of automatically assigning new pods to nodes to achieve balanced distribution across the cluster. There are situations when you must ensure that pods land on specific nodes: for example, for the advantage of speed on an SSD-equipped machine, or reduce costs by choosing nodes in the same availability zone.

The appropriate (sub)sections (`replsets`, `replsets.arbiter`, `backup`, etc.) of the deploy/cr.yaml file contain the keys which can be used to do assign pods to nodes.

Node selector

The `nodeSelector` contains one or more key-value pairs. If the node is not labeled with each key-value pair from the Pod's `nodeSelector`, the Pod will not be able to land on it.

The following example binds the Pod to any node having a self-explanatory `disktype: ssd` label:

```
nodeSelector:
  disktype: ssd
```

Affinity and anti-affinity

Affinity defines eligible pods that can be scheduled on the node which already has pods with specific labels. Anti-affinity defines pods that are not eligible. This approach is reduces costs by ensuring several pods with intensive data exchange occupy the same availability zone or even the same node or, on the contrary, to spread the pods on different nodes or even different availability zones for high availability and balancing purposes.

Percona Operator for MongoDB provides two approaches for doing this:

- simple way to set anti-affinity for Pods, built-in into the Operator,
- more advanced approach based on using standard Kubernetes constraints.

Simple approach - use antiAffinityTopologyKey of the Percona Operator for MongoDB

Percona Operator for MongoDB provides an `antiAffinityTopologyKey` option, which may have one of the following values:

- `kubernetes.io/hostname` - Pods will avoid residing within the same host,
- `topology.kubernetes.io/zone` - Pods will avoid residing within the same zone,
- `topology.kubernetes.io/region` - Pods will avoid residing within the same region,
- `none` - no constraints are applied.

The following example forces Percona Server for MongoDB Pods to avoid occupying the same node:

```
affinity:
  antiAffinityTopologyKey: "kubernetes.io/hostname"
```

Advanced approach - use standard Kubernetes constraints

The previous method can be used without special knowledge of the Kubernetes way of assigning Pods to specific nodes. Still, in some cases, more complex tuning may be needed. In this case, the `advanced` option placed in the deploy/cr.yaml file turns off the effect of the `antiAffinityTopologyKey` and allows the use of the standard Kubernetes affinity constraints of any complexity:

```yaml
affinity:
   advanced:
     podAffinity:
       requiredDuringSchedulingIgnoredDuringExecution:
       - labelSelector:
           matchExpressions:
           - key: security
             operator: In
             values:
             - S1
         topologyKey: failure-domain.beta.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 explanation of the advanced affinity options [in Kubernetes documentation](#).

Topology Spread Constraints

Topology Spread Constraints allow you to control how Pods are distributed across the cluster based on regions, zones, nodes, and other topology specifics. This can be useful for both high availability and resource efficiency.

Pod topology spread constraints are controlled by the `topologySpreadConstraints` subsection, which can be put into `replsets`, `sharding.configsvrReplSet`, and `sharding.mongos` sections of the `deploy/cr.yaml` configuration file as follows:

```yaml
topologySpreadConstraints:
  - labelSelector:
      matchLabels:
        app.kubernetes.io/name: percona-server-mongodb
    maxSkew: 1
    topologyKey: kubernetes.io/hostname
    whenUnsatisfiable: DoNotSchedule
```

You can see the explanation of these affinity options [in Kubernetes documentation](#).

Tolerations

Tolerations allow Pods having them to be able to land onto nodes with matching *taints*. Toleration is expressed as a `key` with and `operator`, which is either `exists` or `equal` (the equal variant requires a corresponding `value` for comparison).

Toleration should have a specified `effect`, such as the following:

- `NoSchedule` - less strict
- `PreferNoSchedule`

- `NoExecute`

When a *taint* with the `NoExecute` effect is assigned to a Node, any Pod configured to not tolerating this *taint* is removed from the node. This removal can be immediate or after the `tolerationSeconds` interval. The following example defines this effect and the removal interval:

```
tolerations:
- key: "node.alpha.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 6000
```

The [Kubernetes Taints and Toleratins] contains more examples on this topic.

Priority Classes

Pods may belong to some *priority classes*. This flexibility allows the scheduler to distinguish more and less important Pods when needed, such as the situation when a higher priority Pod cannot be scheduled without evicting a lower priority one. This ability can be accomplished by adding one or more PriorityClasses in your Kubernetes cluster, and specifying the `PriorityClassName` in the [deploy/cr.yaml] file:

```
priorityClassName: high-priority
```

See the [Kubernetes Pods Priority and Preemption documentation] to find out how to define and use priority classes in your cluster.

Pod Disruption Budgets

Creating the [Pod Disruption Budget] is the Kubernetes method to limit the number of Pods of an application that can go down simultaneously due to *voluntary disruptions* such as the cluster administrator's actions during a deployment update. Distribution Budgets allow large applications to retain their high availability during maintenance and other administrative activities. The `maxUnavailable` and `minAvailable` options in the [deploy/cr.yaml] file can be used to set these limits. The recommended variant is the following:

```
podDisruptionBudget:
   maxUnavailable: 1
```

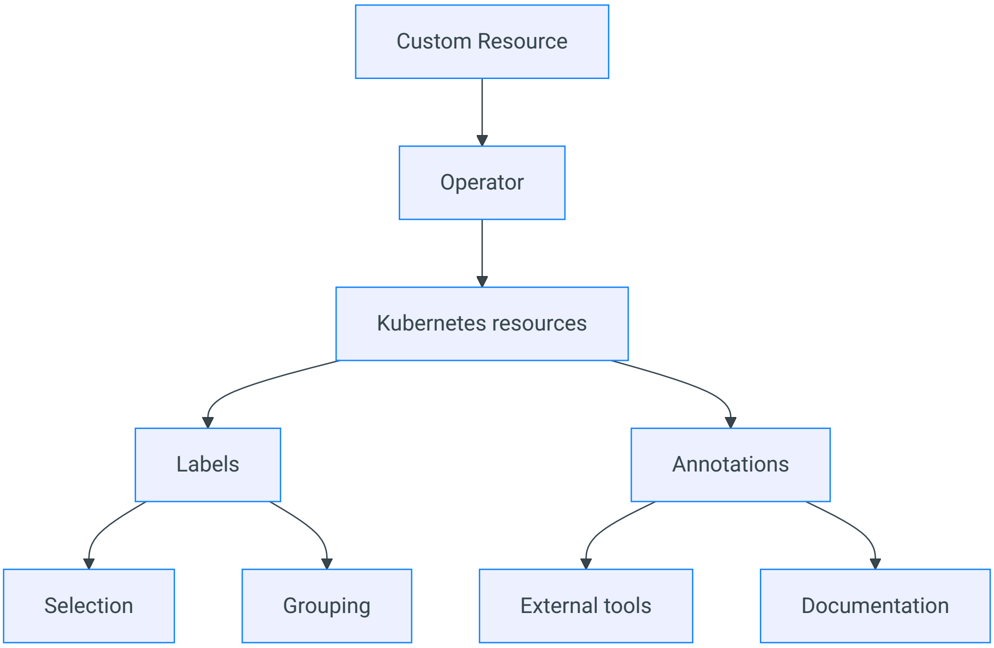
7.7 Labels and annotations

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:

- Custom Resource Definitions
- Custom Resources
- Deployments
- Services
- StatefulSets
- PVCs
- Pods
- ConfigMaps and Secrets
- Backup jobs for scheduled backups

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 MongoDB

Labels

Name	Objects	Description	Example values
`app.kubernetes.io/name`	Services, StatefulSets, Deployments, etc.	Specifies the name of the application for selectors and grouping	percona-server-mongodb
`app.kubernetes.io/instance`	Pods, Services, StatefulSets, Deployments	Identifies a specific instance of the application	my-cluster-name
`app.kubernetes.io/managed-by`	Services, StatefulSets	Indicates the controller managing the object	percona-server-mongodb-operator
`app.kubernetes.io/component`	Pods, Services, StatefulSets	Specifies the component within the application	mongod, mongos, arbiter, external-service , crd
`app.kubernetes.io/part-of`	Services, StatefulSets	Indicates the higher-level application the object belongs to	percona-server-mongodb
`app.kubernetes.io/version`	CustomResourceDefinition	Specifies the version of the Percona Server for MongoDB Operator.	v1.22.0
`app.kubernetes.io/replset`	CustomResourceDefinition	Specifies replica set name in Percona Server for MongoDB Operator.	rs0
`percona.com/backup-ancestor`	Custom Resource	Specifies the name of the backup that was used as a base for the current backup	my-cluster-name-backup-2025-05-23
`percona.com/backup-type`	Custom Resource	Specifies the type of backup being performed (e.g. cron for scheduled backups)	cron
`percona.com/cluster`	Custom Resource	Identifies the MongoDB cluster instance	my-cluster-name
`rack`	Pods, Services, Deployments, StatefulSets	Identifies topology or rack awareness, often for scheduling or affinity	`rack-22`

Annotations

Name	Associated resources	Description	Example values
`iam.amazonaws.com/role`	Pod, PVC, Service	Assigns an AWS IAM role to the resource for permissions.	`role-arn`
`service.beta.kubernetes.io/aws-load-balancer-backend-protocol`	Service	Specifies protocol for AWS load balancer backend.	`http`
`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-applied-secret: "hashvalue"`
`percona.com/last-config-`	Services	Stores the hash of the most recent configuration	

`hash`			
`percona.com/ssl-hash`	Pods	Stores the hash of the most recent TLS configuration	
`percona.com/ssl-internal-hash`	Pods	Stores the hash of the most recent TLS configuration for internal communication	
`percona.com/passwords-updated`	Secrets	Indicates when passwords were last updated in the Secret	
`kubectl.kubernetes.io/restartedAt`	PVC	Records the most recent timestamp when the associated resource was intentionally restarted. This annotation is typically used to trigger rolling restarts of StatefulSets, ensuring Pods or PVCs are refreshed according to Kubernetes best practices.	`2026-02-01T12:34:56Z`

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

To specify labels and annotations for Percona Server for MongoDB Pods, use the `replsets.<name>.annotations` / `replsets.<name>.labels`, `sharding.configsvrReplSet.annotations` / `sharding.configsvrReplSet.labels` and `sharding.mongos.annotations` / `sharding.mongos.labels` options in the `deploy/cr.yaml` Custom Resource. The example configuration for Percona Server for MongoDB database instances looks as follows:

```
apiVersion: psmdb.percona.com/v1
kind: PerconaServerMongoDB
spec:
  replsets:
  - name: rs0
    annotations:
      iam.amazonaws.com/role: role-arn
    labels:
      rack: rack-22
    ...
```

Set labels and annotations for Services

To annotate Services, use the `replsets.<name>.expose.annotations` / `replsets.<name>.expose.labels`, `sharding.configsvrReplSet.expose.annotations` / `sharding.configsvrReplSet.expose.labels` and `sharding.mongos.expose.annotations` / `sharding.mongos.expose.labels` options in the `deploy/cr.yaml` Custom Resource. The example configuration for `mongos` is:

```
spec:
  sharding:
    mongos:
      expose:
        annotations:
          service.beta.kubernetes.io/aws-load-balancer-backend-protocol: http
        labels:
          rack: rack-22
```

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 perconaservermongodbs.psmdb.percona.com --show-labels
```

Sample output

```
NAME                                 CREATED AT             LABELS
perconaservermongodbs.psmdb.percona.com   2025-09-18T08:27:53Z   app.kubernetes.io/component=crd,app.kubernetes.io/name=percona-server-
mongodb,app.kubernetes.io/part-of=percona-server-mongodb-operator,app.kubernetes.io/version=v1.22.0
```

To check **annotations** associated with an object, use the following command:

```
kubectl get <resource> <resource-name> -o jsonpath='{.metadata.annotations}'
```

For example:

```
kubectl get pod my-cluster-name-rs0-0 -o jsonpath='{.metadata.annotations}'
```

Specifying labels and annotations ignored by the Operator

Sometimes various Kubernetes flavors can add their own annotations to the Services managed by the Operator.

The Operator keeps track of all changes to its objects and can remove annotations that it didn't create.

If there are no annotations or labels in the Custom Resource `expose.*` subsections, the Operator does nothing if a new label or annotation is added to the Service object.

If the Service per Pod mode is not used, the Operator **won't remove any annotations and labels** from any Services related to *this expose subsection*. Though, it is still possible to add annotations and labels via the Custom Resource in this case. Use the appropriate `expose.serviceAnnotations` and `expose.serviceLabels` fields.

Else, if the Service per Pod mode is active, the Operator removes unknown annotations and labels from Services *created by the Operator for Pods*.

Yet it is still possible to specify which annotations and labels the Operator should keep. It is useful if a cloud provider adds own labels and annotations. 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.

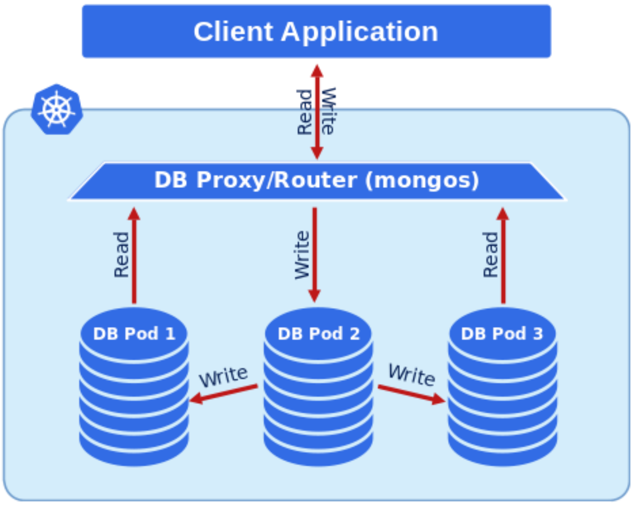
7.8 Exposing the cluster

The Operator provides entry points for accessing the database by client applications in several scenarios. In all cases, the cluster is exposed using regular Kubernetes [Service objects](#), which the Operator configures.

This document describes how to use [Custom Resource manifest options](#) to expose clusters deployed with the Operator.

Using a single entry point in a sharded cluster

By default, the Operator deploys Percona Server for MongoDB as a [sharded cluster](#). This means the database cluster runs special `mongos` Pods - query routers, which act as entry points for client applications:



By default, the Operator creates `mongos` Service with a ClusterIP service type Service. The service type is controlled by the [sharding.mongos.expose.type](#) option in the Custom Resource. The Service works in a round-robin fashion between all the `mongos` Pods.

To view the Service endpoints, run the following command:

```
kubectl get psmdb
```

Expected output

```
NAME              ENDPOINT                                              STATUS   AGE
my-cluster-name   my-cluster-name-mongos.<namespace>.svc.cluster.local      ready    85m
```

To connect to MongoDB, you need to construct the MongoDB connection string URI. For the sharded cluster, specify the `mongos` endpoint for the connection string URI. This is the example format:

```
mongosh "mongodb://userAdmin:userAdminPassword@my-cluster-name-mongos.<namespace name>.svc.cluster.local/admin?ssl=false"
```

Make sure every part of the connection string reflects your environment:

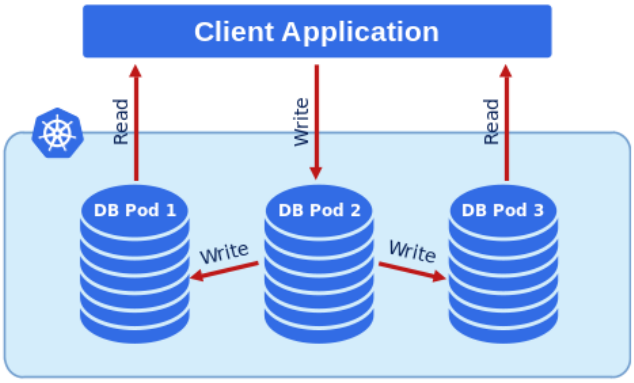
- **userAdmin** and **userAdminPassword**: replace with your admin username and the actual admin password. Get the username and password from the Kubernetes Secret created for your cluster.
- **my-cluster-name**: use the name of your database cluster. Get the name by running `kubectl get psmdb` command
- : the Kubernetes namespace where your cluster is deployed

Warning

A ClusterIP Service endpoint is only reachable inside Kubernetes. If you need to connect from the outside, you need to expose the mongos Pods by using the NodePort or Load Balancer Service types. See the [Connecting from outside Kubernetes](#) section below for details.

Accessing replica set Pods

If Percona Server for MongoDB sharding mode is turned **off**, the Operator deploys Percona Server for MongoDB as a replica set. In this case, an application needs to connect to all the MongoDB Pods of the replica set:



When Kubernetes creates Pods, each Pod gets an IP address in the internal virtual network of the cluster. Since Pod creation and destruction is a dynamic process, binding communication to specific IP addresses would cause problems as the cluster scales or undergoes maintenance. For this reason, you should connect to Percona Server for MongoDB using Kubernetes internal DNS names in the connection string URI.

By default, the Operator creates Services for mongod Pods with a ClusterIP service type Service. The service type is controlled by the replsets.expose.type option. The Service works in a round-robin fashion between all the mongod Pods of the replica set.

You can get the actual Service endpoints by running the following command:

```
kubectl get psmdb
```

Expected output

```
NAME              ENDPOINT                                             STATUS   AGE
my-cluster-name   my-cluster-name-rs0.<namespace>.svc.cluster.local       ready    2m19s
```

To connect to a MongoDB replica set, you need to specify each replica set member for the MongoDB connection string URI. This is the example format:

```
mongosh "mongodb://databaseAdmin:databaseAdminPassword@my-cluster-name-rs0.<namespace name>.svc.cluster.local/admin?
replicaSet=rs0&ssl=false"
```

Make sure every part of the connection string reflects your environment:

- **userAdmin** and **userAdminPassword**: replace with your admin username and the actual admin password. Get the username and password from the Kubernetes Secret created for your cluster.
- **my-cluster-name**: use the name of your database cluster. Get the name by running `kubectl get psmdb` command
- : the Kubernetes namespace where your cluster is deployed

Warning

A ClusterIP Service endpoint is only reachable inside Kubernetes. If you need to connect from the outside, you need to expose the mongod Pods by using the NodePort or Load Balancer Service types. See the Connecting from outside Kubernetes section below for details.

Connecting from outside Kubernetes

If you need to connect to a cluster from outside Kubernetes, you cannot reach the Pods using Kubernetes internal DNS names. To make the Pods accessible from outside the cluster, Percona Operator for MongoDB can create Kubernetes Services with external access.

To expose Pods externally, configure the following option in the Custom Resource:

- Set `expose.enabled` to `true` to enable exposing the Pods via Services.
- Set `expose.type` to specify the type of Service to use:
  - `ClusterIP`: Exposes the Pod with an internal static IP address. This makes the Service reachable only from within the Kubernetes cluster.

- `NodePort`: Exposes the Pod on each Kubernetes Node's IP address at a static port. A ClusterIP Service is automatically created, and the Node port routes traffic to it. The Service is reachable from outside the cluster using the Node address and port number, but the address is bound to a specific Kubernetes Node.

  The `expose.externalTrafficPolicy` Custom Resource option in `replsets`, `sharding.configsvrReplSet`, and `sharding.mongos` subsections controls how external traffic is routed:

  - `Local`: Traffic is routed to node-local endpoints. External requests will be dropped if there is no available Pod on the Node.
  - `Cluster`: Traffic can be routed to any Node in the cluster, but this adds extra latency and does not preserve the client IP address.
- `LoadBalancer`: Exposes the Pod externally using a cloud provider's load balancer. Both ClusterIP and NodePort Services are automatically created in this variant.

If the NodePort type is used, the URI looks like this:

`mongodb://databaseAdmin:databaseAdminPassword@<node1>:<port1>,<node2>:<port2>,<node3>:<port3>/admin?replicaSet=rs0&ssl=false`

All Node addresses should be *directly* reachable by the application.

Service per Pod

To make all database Pods accessible, Percona Operator for MongoDB can assign a Kubernetes Service to each Pod. This Service per Pod option allows your application to handle Cursor tracking instead of relying on a single Service. This solves the problem of `CursorNotFound` errors that occur when a Service transparently cycles between mongos instances while your client is still iterating a cursor on a large collection.

This feature can be enabled for both sharded and non-sharded clusters by setting the sharding.mongos.expose.servicePerPod Custom Resource option to `true` in the deploy/cr.yaml file.

If this feature is enabled with the `expose.type: NodePort`, the created Services look like this:

```
kubectl get svc
NAME                     TYPE         CLUSTER-IP      EXTERNAL-IP    PORT(S)                  AGE
my-cluster-name-mongos-0   NodePort     10.38.158.103   <none>         27017:31689/TCP          12s
my-cluster-name-mongos-1   NodePort     10.38.155.250   <none>         27017:31389/TCP          12s
...
```

Controlling hostnames in replset configuration

Starting from v1.14, the Operator configures replica set members using local fully-qualified domain names (FQDN), which are resolvable and available only from inside the Kubernetes cluster. Exposing the replica set using the options described above will not affect hostname usage in the replica set configuration.

> **Note**
>
> Before v1.14, the Operator used the exposed IP addresses in the replica set configuration in the case of the exposed replica set.

It is still possible to restore the old behavior. For example, it may be useful to have the replica set configured with external IP addresses for multi-cluster deployments. The `clusterServiceDNSMode` field in the Custom Resource controls this Operator behavior. You can set `clusterServiceDNSMode` to one of the following values:

1. `Internal`: Use local FQDNs (i.e., `cluster1-rs0-0.cluster1-rs0.psmdb.svc.cluster.local`) in replica set configuration even if the replica set is exposed. **This is the default value.**
2. `ServiceMesh`: Use a special FQDN using the Pod name (i.e., `cluster1-rs0-0.psmdb.svc.cluster.local`), assuming it's resolvable and available in all clusters. This mode is designed for service mesh environments where Pod names are resolvable across clusters. Starting with version 1.22.0, the integration with service meshes is improved. See application protocol support to learn more.
3. `External`: Use exposed IP addresses in replica set configuration if the replica set is exposed; otherwise, use local FQDN. **This copies the behavior of the Operator v1.13.**

!!! warning

```
Be careful with the `clusterServiceDNSMode=External` variant. Using IP addresses instead of DNS hostnames is discouraged in
MongoDB. IP addresses make reconfiguration and recovery more complicated, and are **generally problematic in scenarios where
IP addresses change**. In particular, if you delete and recreate the cluster with `clusterServiceDNSMode=External` without
deleting its volumes (having `percona.com/delete-psmdb-pvc` finalizer unset), your cluster will crash and there will be no
straightforward way to recover it.
```

If backups are enabled in your cluster, you need to restart replica set and config server replica set Pods after changing `clusterServiceDNSMode`. This option changes the hostnames inside the replica set configuration. Since `pbm-agents` running in your cluster don't discover this change, you must restart them too. You may see errors in `backup-agent` container logs, and your backups may not work until you restart the agents.

To make a manual restart, run the `kubectl rollout restart sts <clusterName>-<replsetName>` command for each replica set in the `spec.replsets`. For sharded cluster, also restart config servers with the `kubectl rollout restart sts <clusterName>-cfg`.

Alternatively, you can simply [restart your cluster](#).

Application protocol support for service mesh integrations

Starting from version 1.22.0, the Operator automatically sets `appProtocol: mongo` on Service ports for all Services (replica sets, config servers, and mongos). This configuration is essential when using service mesh implementations such as Istio.

MongoDB uses a server-first protocol, meaning the server sends data first when establishing a connection. Without explicit protocol specification, service meshes cannot properly detect the protocol and may fail to establish Mutual TLS (mTLS) connections. By setting `appProtocol: mongo`, the Operator tells the service mesh which protocol to expect, enabling proper protocol detection and ensuring mTLS connections work correctly.

The `appProtocol` field is set automatically by the Operator and cannot be configured manually.

Exposing replica set with split-horizon DNS

[Split-horizon DNS](#) provides each replica set Pod with a set of DNS URIs for external usage. This allows you to communicate with replica set Pods both from inside the Kubernetes cluster and from outside Kubernetes.

Split-horizon can be configured via the `replset.splitHorizons` subsection in the Custom Resource options. Set it in the `deploy/cr.yaml` configuration file as follows:

```yaml
    ...
    replsets:
      - name: rs0
        expose:
          enabled: true
          type: LoadBalancer
        splitHorizons:
          cluster1-rs0-0:
            external: rs0-0.mycluster.xyz
            external-2: rs0-0.mycluster2.xyz
          cluster1-rs0-1:
            external: rs0-1.mycluster.xyz
            external-2: rs0-1.mycluster2.xyz
          cluster1-rs0-2:
            external: rs0-2.mycluster.xyz
            external-2: rs0-2.mycluster2.xyz
```

URIs for external usage are specified as key-value pairs, where the key is an arbitrary name and the value is the actual URI. The URI may include a port number. If no port is specified, the default MongoDB port (27017) is used.

Starting with version 1.22.0, the Operator adds the DNS records defined in the `splitHorizons` subsection to the certificates it generates.

Split horizon has following limitations:

- duplicating domain names in horizons is not allowed by MongoDB
- using IP addresses in horizons is not allowed by MongoDB
- horizons should be set for *all Pods of a replica set* or not set at all

7.9 Local Storage support for the Percona Operator for MongoDB

Among the wide rage of volume types, supported by Kubernetes, there are two volume types which allow Pod containers to access part of the local filesystem on the node the *emptyDir* and *hostPath*.

emptyDir

A Pod emptyDir volume is created when the Pod is assigned to a Node. The volume is initially empty and is erased when the Pod is removed from the Node. The containers in the Pod can read and write the files in the emptyDir volume.

The `emptyDir` options in the deploy/cr.yaml file can be used to turn the emptyDir volume on by setting the directory name.

The `emptyDir` is useful when you use Percona Memory Engine.

hostPath

A hostPath volume mounts an existing file or directory from the host node's filesystem into the Pod. If the pod is removed, the data persists in the host node's filesystem.

The `volumeSpec.hostPath` subsection in the deploy/cr.yaml file may include `path` and `type` keys to set the node's filesystem object path and to specify whether it is a file, a directory, or something else (e.g. a socket):

```
volumeSpec:
  hostPath:
    path: /data
    type: Directory
```

Please note, you must created the hostPath manually and should have following attributes:

- access permissions,
- ownership,
- SELinux security context.

The `hostPath` volume is useful when you perform manual actions during the first run and require improved disk performance. Consider using the tolerations settings to avoid a cluster migration to different hardware in case of a reboot or a hardware failure.

More details can be found in the official hostPath Kubernetes documentation.

7.10 Replica set members and their usage

Percona Server for MongoDB replica set is a number of `mongod` instances working together to ensure data durability and system resilience. Such configuration enhances fault tolerance and keeps your database accessible even during failures.

A replica set consists of one **primary** node and several **secondary** nodes. The primary node accepts all write operations, while secondary nodes replicate the data set to maintain redundancy. Secondary nodes can serve read queries, which helps distribute the read load. Secondary nodes can also have additional configuration, like be non-voting or hidden.

Percona Server for MongoDB replication mechanism is based on elections, when replica set nodes [choose which node](#) becomes the primary. For elections to be successful, the number fo voting members must be odd.

By default, the Operator creates Percona Server for MongoDB replica set with three members, one primary and the remaining secondaries. This is the minimal recommended configuration. A replica set can have up to 50 members with the maximum of 7 voting members.

Replica set member types

Besides the primary and regular secondaries in a MongoDB replica set, you can have special member configurations like hidden, arbiter, and non-voting members.

- **Arbiter**: An arbiter node participates in elections but does not store data. You may want to add arbiter nodes if cost constraints prevent you from adding another secondary node.
- **Non-voting**: This type of node stores a full copy of the data but does not participate in elections. This is useful for scaling read capacity beyond the seven-member voting limit of a replica set.
- **Hidden**: A hidden node is a secondary member that holds data but is invisible to client applications. It is added as a voting member and can participate in elections. It is useful for tasks like backups or running batch jobs that might otherwise interfere with primary operations.

Arbiter nodes

An Arbiter node participates in the replica set elections but does not store any data. Its primary role is to act as a tiebreaker in a replica set with an even number of data-bearing nodes, ensuring that a primary can always be elected. By not storing data, Arbiter nodes require minimal resources, which can help reduce your overall costs. An does not demand a persistent volume.

To add an Arbiter node, you can update your `deploy/cr.yaml` file by adding an `arbiter` section under `replsets` and setting the `enabled` and `size` options to your desired values.

The following example configuration will create a cluster with 4 data instances and 1 Arbiter:

```
....
replsets:
  ....
  size: 4
  ....
  arbiter:
    enabled: true
    size: 1
    ....
```

Find the description of other available options in the replsets.arbiter section of the Custom Resource options reference.

Prevent Arbiter nodes on the same Kubernetes hosts with data-bearing replica set members

By default, Arbiter nodes are allowed to run on the same Kubernetes hosts as your data nodes. This may be reasonable in terms of the number of Kubernetes Nodes required for the cluster. But as a result it increases possibility to have 50/50 votes division in case of network partitioning. To prevent this, you can apply an anti-affinity constraint, which forces arbiter nodes to be scheduled on separate nodes:

```
....
arbiter:
  enabled: true
  size: 1
  affinity:
    antiAffinityTopologyKey: "kubernetes.io/hostname"
    advanced:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchLabels:
              app.kubernetes.io/component: mongod
              app.kubernetes.io/instance: cluster1
              app.kubernetes.io/managed-by: percona-server-mongodb-operator
              app.kubernetes.io/name: percona-server-mongodb
              app.kubernetes.io/part-of: percona-server-mongodb
              app.kubernetes.io/replset: rs0
          topologyKey: kubernetes.io/hostname
```

Non-voting nodes

A non-voting node is a secondary member that stores a full copy of the data but does not participate in elections for the primary node. Non-voting nodes enable you to deploy a replica set with more than seven data-bearing nodes. You can also add a non-voting node to a remote location where network latency might make it unsuitable for voting.

You can add non-voting nodes by setting the `replsets.nonvoting.enabled` and `replsets.nonvoting.size` options in your `deploy/cr.yaml` file.

In this example, the Operator will create a cluster with 3 data instances and 1 non-voting instance:

```
....
replsets:
  ....
  size: 3
  ....
  nonvoting:
    enabled: true
    size: 1
    ....
```

Find the description of other available options in the [replsets.nonvoting section](#) of the [Custom Resource options reference](#).

Note that you can add a non-voting node in the edge location through the `externalNodes` option. Please see [cross-site replication documentation](#) for details.

Hidden nodes

Hidden nodes are secondary members that hold a full copy of the data but are not visible to client applications. Hidden nodes always have a 0 priority and therefore, cannot become a primary. But hidden members are added as voting members and may, therefore, vote in primary elections. Read more how the Operator [manages voting members in replica set](#).

Hidden nodes are useful for tasks like backups or reporting, as they do not affect primary operations. Client applications will not connect to hidden nodes because they are not listed in the replica set's SRV record.

To add a hidden node with the Operator, set the setting the `replsets.hidden.enabled` and `replsets.hidden.size` options in the `deploy/cr.yaml` file:

This configuration example creates a cluster with 3 data instances and 2 hidden nodes:

```
....
replsets:
  ....
  size: 3
  ....
  hidden:
    enabled: true
    size: 2
    ....
```

Find the description of other available options in the [replsets.hidden section](#) of the [Custom Resource options reference](#).

Manage voting members in replica set

Since hidden nodes can participate in elections, the Operator enforces rules to ensure the odd number of voting members and maintain a stable and compliant replica set configuration:

- If the total number of voting members is even, the Operator converts one node to non-voting to maintain an odd number of voters. The node to convert is typically the last Pod in the list
- If the number of voting members is odd and not more than 7, all nodes participate in elections.
- If the number of voting members exceeds 7, the Operator automatically converts some nodes to non-voting to stay within MongoDB's limit of 7 voting members.

To inspect the current configuration, connect to the cluster with clusterAdmin privileges and run:

```
rs.config() command
```

7.11 Percona Server for MongoDB Sharding

About sharding

Sharding provides horizontal database scaling, distributing data across multiple MongoDB Pods. It is useful for large data sets when a single machine's overall processing speed or storage capacity turns out to be not enough. Sharding allows splitting data across several machines with a special routing of each request to the necessary subset of data (so-called *shard*).

A MongoDB Sharding involves the following components:

- `shard` - a replica set which contains a subset of data stored in the database (similar to a traditional MongoDB replica set),
- `mongos` - a query router, which acts as an entry point for client applications,
- `config servers` - a replica set to store metadata and configuration settings for the sharded database cluster.

> **Note**
>
> Percona Operator for MongoDB 1.6.0 supported only one shard of a MongoDB cluster; still, this limited sharding support allowed using `mongos` as an entry point instead of provisioning a load-balancer per replica set node. Multiple shards are supported starting from the Operator 1.7.0. Also, before the Operator 1.12.0 mongos were deployed by the Deployment object, and starting from 1.12.0 they are deployed by the StatefulSet one.

Turning sharding on and off

Sharding is controlled by the `sharding` section of the `deploy/cr.yaml` configuration file and is turned on by default.

To enable sharding, set the `sharding.enabled` key to `true`. This will turn existing MongoDB replica set nodes into sharded ones).

To disable sharding, set the `sharding.enabled` key to `false`. If backups are disabled (the `backup.enabled` Custom Resource option set to `false`), the Operator will turn sharded MongoDB instances into unsharded one by one, so the database cluster will operate without downtime. If backups are enabled (the `backup.enabled` Custom Resource option is `true`), the Operator will pause the cluster (to avoid Percona Backup for MongoDB misconfiguration), update the instances, and then unpause it back.

Configuring instances of a sharded cluster

When sharding is turned on, the Operator runs replica sets with config servers and mongos instances. Their number is controlled by `configsvrReplSet.size` and `mongos.size` keys, respectively.

Config servers have `cfg` replica set name by default, which is used by the Operator in StatefulSet and Service names. If this name needs to be customized (for example when migrating MongoDB cluster from barebone installation to Kubernetes), you can override the default `cfg` variant using `replsets.configuration` Custom Resource option in `deploy/cr.yaml` as follows:

```
...
configuration: |
  replication:
    replSetName: customCfgRS
    ...
```

> **Note**
>
> Config servers for now can properly work only with WiredTiger engine, and sharded MongoDB nodes can use either WiredTiger or InMemory one.

By default replsets section of the `deploy/cr.yaml` configuration file contains only one replica set, `rs0`. You can add more replica sets with different names to the `replsets` section in a similar way. Please take into account that having more than one replica set is possible only with the sharding turned on.

> **Note**
>
> The Operator will be able to remove a shard only when it contains no application (non-system) collections.

Checking connectivity to sharded and non-sharded cluster

With sharding turned on, you have `mongos` service as an entry point to access your database. If you do not use sharding, you have to access `mongod` processes of your replica set.

To connect to Percona Server for MongoDB you need to construct the MongoDB connection URI string. It includes the credentials of the admin user, which are stored in the [Secrets](#) object.

1. List the Secrets objects

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

   The Secrets object you are interested in has the `my-cluster-name-secrets` name by default.

2. View the Secret contents to retrieve the admin user credentials.

   ```
   kubectl get secret my-cluster-name-secrets -o yaml
   ```

   The command returns the YAML file with generated Secrets, including the `MONGODB_DATABASE_ADMIN_USER` and `MONGODB_DATABASE_ADMIN_PASSWORD` strings, which should look as follows:

   Sample output

   ```
   ...
   data:
     ...
     MONGODB_DATABASE_ADMIN_PASSWORD: aDAzQ0pCY3NSWEZ2ZUIzS1I=
     MONGODB_DATABASE_ADMIN_USER: ZGF0YWJhc2VBZG1pbg==
   ```

   The actual login name and password on the output are base64-encoded. To bring it back to a human-readable form, run:

   ```
   echo 'MONGODB_DATABASE_ADMIN_USER' | base64 --decode
   echo 'MONGODB_DATABASE_ADMIN_PASSWORD' | base64 --decode
   ```

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

   ```
   kubectl run -i --rm --tty percona-client --image=percona/percona-server-mongodb:7.0.30-16 --restart=Never -- bash -il
   ```

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

4. Now run `mongosh` tool inside the `percona-client` command shell using the admin user credentialds you obtained from the Secret, and a proper namespace name instead of the `<namespace name>` placeholder. The command will look different depending on whether sharding is on (the default behavior) or off:

   if sharding is on

   ```
   mongosh "mongodb://databaseAdmin:databaseAdminPassword@my-cluster-name-mongos.<namespace name>.svc.cluster.local/admin?ssl=false"
   ```

   if sharding is off

   ```
   mongosh "mongodb+srv://databaseAdmin:databaseAdminPassword@my-cluster-name-rs0.<namespace name>.svc.cluster.local/admin?replicaSet=rs0&ssl=false"
   ```

7.12 Transport encryption (TLS/SSL)

7.12.1 Transport Layer Security (TLS)

Percona Operator for MongoDB uses Transport Layer Security (TLS) cryptographic protocol for the following types of communication:

- External - to enable client applications communicate with the cluster
- Internal - for communication between Percona Server for MongoDB instances in the cluster. The internal certificate is also used as an authorization method.

You control TLS usage with the `tls.mode` option in the Custom Resource. This setting defines how Percona Server for MongoDB cluster handles TLS for both internal and external connections. You can choose from the following modes:

- `allowTLS`: The cluster accepts both TLS and non-TLS incoming connections, but does not use TLS for internal communication.
- `preferTLS` (default): The cluster uses TLS for internal communication and accepts both TLS and non-TLS external connections.
- `requireTLS`: The cluster enforces TLS encryption for all connections and accepts only TLS connections.
- `disabled`: The cluster completely disables TLS for all connections.

Example configuration:

```
...
spec:
  ...
  tls:
    mode: preferTLS
```

TLS Certificates

TLS security can be configured in several ways:

- The Operator generates long-term certificates automatically during the cluster creation if there are no certificate secrets available. When generating certificates, the Operator creates two Secrets objects named `<cluster-name>-ssl` and `<cluster-name>-ssl-internal`. These Secrets are also referenced in the `secrets.ssl` and `secrets.sslInternal` options in the Custom Resource.

This is the default behavior. If you need new certificates, and you must renew them manually.

To allow certificates automatically generated by the Operator, the tls.allowInvalidCertificates Custom Resource option is set to `true` by default. You can set it to `false` when using other certificate generation methods, such as using cert-manager.

- The Operator can use a specifically installed *cert-manager*, which will automatically generate and renew short-term TLS certificates
- You can generate TLS certificates manually.

For testing purposes, you can use pre-generated certificates available in the `deploy/ssl-secrets.yaml` file. But we strongly recommend **to not use them on any production system**!

TLS configuration

The following sections provide guidelines how to:

- Configure TLS security with the Operator using cert-manager
- Generate certificates manually
- Update certificates
- Disable TLS temporarily

To use TLS for external traffic, you need to additionally configure your client application. See this blog post for detailed instruction with examples. Also, you can check the official MongoDB documentation.

For clients outside of your Kubernetes-based environment, you must also expose your cluster.

7.12.2 Install and use the *cert-manager*

About the *cert-manager*

The [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* and deploy the operator, the operator requests a certificate from the *cert-manager*. The *cert-manager* acts as a self-signed issuer and generates certificates. The Percona Operator self-signed issuer is local to the operator namespace. This self-signed issuer is created because Percona Server for MongoDB requires all certificates issued by the same CA (Certificate authority).

Self-signed issuer allows you to deploy and use the Percona Operator without creating a cluster issuer separately.

Install the *cert-manager*

The steps to install the *cert-manager* are the following:

- create a namespace,
- disable resource validations on the cert-manager namespace,
- install the cert-manager.

The following commands perform all the needed actions:

```
kubectl apply -f https://github.com/jetstack/cert-manager/releases/download/v1.19.3/cert-manager.yaml --validate=false
```

After the installation, you can 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:

```
NAME                                       READY   STATUS    RESTARTS   AGE
cert-manager-7d59dd4888-tmjqq              1/1     Running   0          3m8s
cert-manager-cainjector-85899d45d9-8ncw9   1/1     Running   0          3m8s
cert-manager-webhook-84fcdcd5d-697k4       1/1     Running   0          3m8s
```

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.

7.12.3 Generate TLS certificates manually

> **Warning**
>
> Manual certificate generation didn't work in Operator version 1.16.0. This issue is fixed starting from version 1.16.1.

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 MongoDB nodes
- `server-key.pem` - Private key for the server certificate
- `client.pem` - Client certificate for external connections
- `client-key.pem` - Private key for the client certificate
- `ca.pem` - Certificate Authority certificate
- `ca-key.pem` - Certificate Authority private key

Certificate requirements

You need to create **two sets** of certificates:

1. **External certificates** - for client connections from outside the cluster. This set is mandatory
2. **Internal certificates** - for communication between MongoDB nodes within the cluster. You can omit generating a separate of certificates for internal communication. In this case, the operator reuses the external certificates for both external and internal communication.

Generating two separate sets of certificates is not mandatory but highly recommended. Using different certificates for internal and external connections gives you more control and improves security. For example, you can choose to renew or replace external certificates more often than internal ones, or set different expiration dates for each. This makes managing and securing your cluster easier in the long run.

After creating the certificates, you'll create two Kubernetes Secrets and reference them in your cluster configuration.

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

Procedure

Generate certificates

Replace `my-cluster-name` and `my-namespace` with your actual cluster name and namespace in the commands below.

Sharded cluster (sharding is enabled)

1. Set your cluster variables

```
CLUSTER_NAME=my-cluster-name
NAMESPACE=my-namespace
```

2. Create the Certificate Authority (CA)

This command creates a root Certificate Authority that will sign all your certificates:

```
cat <<EOF | cfssl gencert -initca - | cfssljson -bare ca
  {
    "CN": "Root CA",
    "names": [
      {
        "O": "PSMDB"
      }
    ],
    "key": {
      "algo": "rsa",
      "size": 2048
    }
  }
EOF
```

3. Create CA configuration file that defines how the CA will sign certificates:

```
cat <<EOF > ca-config.json
  {
    "signing": {
      "default": {
        "expiry": "87600h",
        "usages": ["signing", "key encipherment", "server auth", "client auth"]
      }
    }
  }
EOF
```

4. Generate the certificate for internal MongoDB node communication, including all shard components:

```
cat <<EOF | cfssl gencert -ca=ca.pem  -ca-key=ca-key.pem -config=./ca-config.json - | cfssljson -bare server
  {
    "hosts": [
      "localhost",
      "${CLUSTER_NAME}-rs0",
      "${CLUSTER_NAME}-rs0.${NAMESPACE}",
      "${CLUSTER_NAME}-rs0.${NAMESPACE}.svc.cluster.local",
      "*.${CLUSTER_NAME}-rs0",
      "*.${CLUSTER_NAME}-rs0.${NAMESPACE}",
      "*.${CLUSTER_NAME}-rs0.${NAMESPACE}.svc.cluster.local",
      "${CLUSTER_NAME}-mongos",
      "${CLUSTER_NAME}-mongos.${NAMESPACE}",
      "${CLUSTER_NAME}-mongos.${NAMESPACE}.svc.cluster.local",
      "*.${CLUSTER_NAME}-mongos",
      "*.${CLUSTER_NAME}-mongos.${NAMESPACE}",
      "*.${CLUSTER_NAME}-mongos.${NAMESPACE}.svc.cluster.local",
      "${CLUSTER_NAME}-cfg",
      "${CLUSTER_NAME}-cfg.${NAMESPACE}",
      "${CLUSTER_NAME}-cfg.${NAMESPACE}.svc.cluster.local",
      "*.${CLUSTER_NAME}-cfg",
      "*.${CLUSTER_NAME}-cfg.${NAMESPACE}",
      "*.${CLUSTER_NAME}-cfg.${NAMESPACE}.svc.cluster.local"
    ],
    "names": [
      {
        "O": "PSMDB"
      }
    ],
    "CN": "${CLUSTER_NAME/-rs0}",
    "key": {
      "algo": "rsa",
      "size": 2048
    }
  }
EOF
```

5. Bundle the server certificate with the CA certificate:

```
cfssl bundle -ca-bundle=ca.pem -cert=server.pem | cfssljson -bare server
```

6. Create a Kubernetes Secret for internal cluster communication:

```
kubectl create secret generic my-cluster-name-ssl-internal --from-file=tls.crt=server.pem --from-file=tls.key=server-key.pem --from-file=ca.crt=ca.pem --type=kubernetes.io/tls
```

7. Generate the certificate for external client connections, including all shard components:

```
cat <<EOF | cfssl gencert -ca=ca.pem  -ca-key=ca-key.pem -config=./ca-config.json - | cfssljson -bare client
  {
    "hosts": [
      "${CLUSTER_NAME}-rs0",
      "${CLUSTER_NAME}-rs0.${NAMESPACE}",
      "${CLUSTER_NAME}-rs0.${NAMESPACE}.svc.cluster.local",
      "*.${CLUSTER_NAME}-rs0",
      "*.${CLUSTER_NAME}-rs0.${NAMESPACE}",
      "*.${CLUSTER_NAME}-rs0.${NAMESPACE}.svc.cluster.local",
      "${CLUSTER_NAME}-mongos",
      "${CLUSTER_NAME}-mongos.${NAMESPACE}",
      "${CLUSTER_NAME}-mongos.${NAMESPACE}.svc.cluster.local",
      "*.${CLUSTER_NAME}-mongos",
      "*.${CLUSTER_NAME}-mongos.${NAMESPACE}",
      "*.${CLUSTER_NAME}-mongos.${NAMESPACE}.svc.cluster.local",
      "${CLUSTER_NAME}-cfg",
      "${CLUSTER_NAME}-cfg.${NAMESPACE}",
      "${CLUSTER_NAME}-cfg.${NAMESPACE}.svc.cluster.local",
      "*.${CLUSTER_NAME}-cfg",
      "*.${CLUSTER_NAME}-cfg.${NAMESPACE}",
      "*.${CLUSTER_NAME}-cfg.${NAMESPACE}.svc.cluster.local"
    ],
    "names": [
      {
        "O": "PSMDB"
      }
    ],
    "CN": "${CLUSTER_NAME/-rs0}",
    "key": {
      "algo": "rsa",
      "size": 2048
    }
  }
EOF
```

8. Create a Kubernetes Secret for external client connections:

```
kubectl create secret generic my-cluster-name-ssl --from-file=tls.crt=client.pem --from-file=tls.key=client-key.pem --from-file=ca.crt=ca.pem --type=kubernetes.io/tls
```

Replica set only (no sharding)

1. Set your cluster variables

```
CLUSTER_NAME=my-cluster-name
NAMESPACE=my-namespace
```

2. Create the Certificate Authority (CA)

   This command creates a root Certificate Authority that will sign all your certificates:

```
cat <<EOF | cfssl gencert -initca - | cfssljson -bare ca
  {
    "CN": "Root CA",
    "names": [
      {
        "O": "PSMDB"
      }
    ],
    "key": {
      "algo": "rsa",
      "size": 2048
    }
  }
EOF
```

3. Create a CA configuration file that defines how the CA will sign certificates:

```
cat <<EOF > ca-config.json
  {
    "signing": {
      "default": {
        "expiry": "87600h",
        "usages": ["signing", "key encipherment", "server auth", "client auth"]
      }
    }
  }
EOF
```

4. Generate the certificate for internal MongoDB node communication:

```
cat <<EOF | cfssl gencert -ca=ca.pem  -ca-key=ca-key.pem -config=./ca-config.json - | cfssljson -bare server
  {
    "hosts": [
      "localhost",
      "${CLUSTER_NAME}-rs0",
      "${CLUSTER_NAME}-rs0.${NAMESPACE}",
      "${CLUSTER_NAME}-rs0.${NAMESPACE}.svc.cluster.local",
      "*.${CLUSTER_NAME}-rs0",
      "*.${CLUSTER_NAME}-rs0.${NAMESPACE}",
      "*.${CLUSTER_NAME}-rs0.${NAMESPACE}.svc.cluster.local"
    ],
    "names": [
      {
        "O": "PSMDB"
      }
    ],
    "CN": "${CLUSTER_NAME/-rs0}",
    "key": {
      "algo": "rsa",
      "size": 2048
    }
  }
EOF
```

5. Bundle the server certificate with the CA certificate:

```
cfssl bundle -ca-bundle=ca.pem -cert=server.pem | cfssljson -bare server
```

6. Create a Kubernetes Secret for internal cluster communication:

```
kubectl create secret generic my-cluster-name-ssl-internal --from-file=tls.crt=server.pem --from-file=tls.key=server-key.pem --from-file=ca.crt=ca.pem --type=kubernetes.io/tls
```

7. Generate the certificate for external client connections:

```
cat <<EOF | cfssl gencert -ca=ca.pem  -ca-key=ca-key.pem -config=./ca-config.json - | cfssljson -bare client
  {
    "hosts": [
      "${CLUSTER_NAME}-rs0",
      "${CLUSTER_NAME}-rs0.${NAMESPACE}",
      "${CLUSTER_NAME}-rs0.${NAMESPACE}.svc.cluster.local",
      "*.${CLUSTER_NAME}-rs0",
      "*.${CLUSTER_NAME}-rs0.${NAMESPACE}",
      "*.${CLUSTER_NAME}-rs0.${NAMESPACE}.svc.cluster.local"
    ],
    "names": [
      {
        "O": "PSMDB"
      }
    ],
    "CN": "${CLUSTER_NAME/-rs0}",
    "key": {
      "algo": "rsa",
      "size": 2048
    }
  }
EOF
```

8. Create a Kubernetes Secret for external client connections:

```
kubectl create secret generic my-cluster-name-ssl --from-file=tls.crt=client.pem --from-file=tls.key=client-key.pem --from-file=ca.crt=ca.pem --type=kubernetes.io/tls
```

Configure your cluster

After creating the Secrets, add them to your cluster configuration in the `deploy/cr.yaml` file:

```
spec:
  secrets:
    ssl: my-cluster-name-ssl          # External certificate secret
    sslInternal: my-cluster-name-ssl-internal  # Internal certificate secret
```

Important notes

1. If you only create the external certificate, the Operator will use it for both external and internal communications instead of generating a separate internal certificate.
2. The commands above use `rs0` as the replica set name (the default). If you set a different name in the `replsets.name` Custom Resource option, update the commands accordingly.

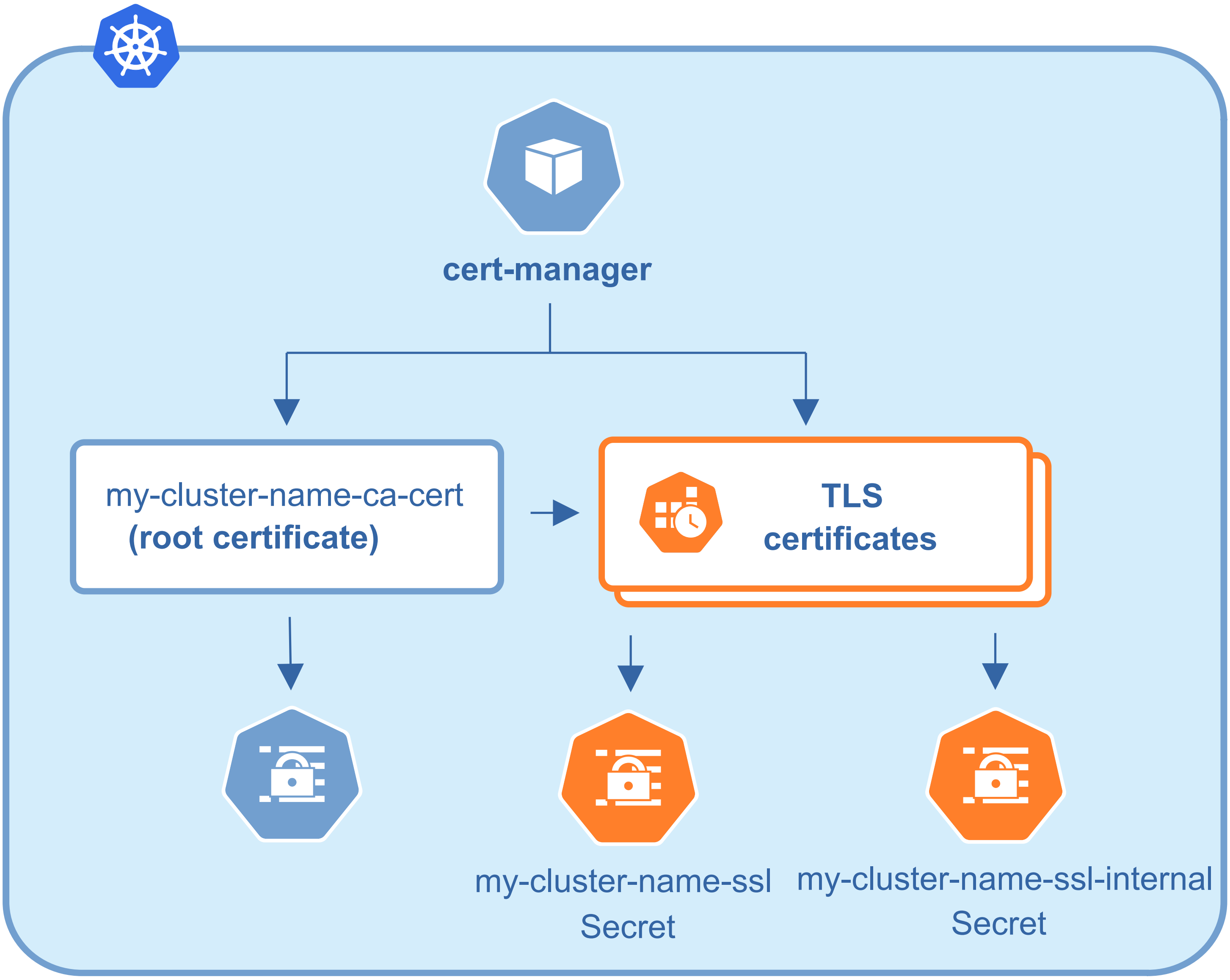
Additional resources

- Check the sample certificates in `deploy/ssl-secrets.yaml` for reference
- Review MongoDB certificate requirements in the upstream documentation

7.12.4 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

1. First, check the necessary secrets names (`my-cluster-name-ssl` and `my-cluster-name-ssl-internal` by default):

    ```
    kubectl get certificate
    ```

    You will have the following response:

    ```
    NAME                           READY   SECRET                         AGE
    my-cluster-name-ssl            True    my-cluster-name-ssl            49m
    my-cluster-name-ssl-internal   True    my-cluster-name-ssl-internal   49m
    ```

This command is available if you have cert-manager installed; if not, you can still check the necessary secrets names with `kubectl get secrets` command.

1. Optionally you can also check that the certificates issuer is up and running:

```
kubectl get issuer
```

The response should be as follows:

```
NAME                              READY   AGE
my-cluster-name-psmdb-issuer      True    61m
my-cluster-name-psmdb-ca-issuer   True    61m
```

Again, this command is provided by cert-manager; if you don't have it installed, you can still use `kubectl get secrets`.

> **Note**
>
> The presence of two issuers has the following meaning. The `my-cluster-name-psmdb-ca-issuer` issuer is used to create a self signed CA certificate (`my-cluster-name-ca-cert`), and then the `my-cluster-name-psmdb-issuer` issuer is used to create SSL certificates (`my-cluster-name-ssl` and `my-cluster-name-ssl-internal`) signed by the `my-cluster-name-ca-cert` CA certificate.

2. Now use the following command to find out the certificates validity dates, substituting Secrets names if necessary:

```
{
  kubectl get secret/my-cluster-name-ssl-internal -o jsonpath='{.data.tls\.crt}' | base64 --decode | openssl x509 -noout -
dates
  kubectl get secret/my-cluster-name-ssl -o jsonpath='{.data.ca\.crt}' | base64 --decode | openssl x509 -noout -dates
  }
```

The resulting output will be self-explanatory:

```
notBefore=Apr 25 12:09:38 2022 GMT notAfter=Jul 24 12:09:38 2022 GMT
notBefore=Apr 25 12:09:38 2022 GMT notAfter=Jul 24 12:09:38 2022 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`). Optionally you can also generate a new TLS certificate and a key for it, but those can be generated later on step 6.
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/my-cluster-name-ssl-internal -o jsonpath='{.data.ca\.crt}' | base64 --decode > ca.pem.old
kubectl get secret/my-cluster-name-ssl-internal -o jsonpath='{.data.tls\.crt}' | base64 --decode > tls.pem.old
kubectl get secret/my-cluster-name-ssl-internal -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 *old* TLS certificate (`tls.pem.old`) and key (`tls.key.old`), but a *new combined* `ca.pem` (`ca.pem.combined`):

```
kubectl delete secret/my-cluster-name-ssl-internal
kubectl create secret generic my-cluster-name-ssl-internal --from-file=tls.crt=tls.pem.old --from-file=tls.key=tls.key.old
--from-file=ca.crt=ca.pem.combined --type=kubernetes.io/tls
```

5. The cluster will go through a rolling reconciliation, but it will do it without problems, as every node has old TLS certificate/key, and both new and old CA certificates.
6. If new TLS certificate and key weren't generated on step 1, do that now.
7. Create a new Secrets object for the second time: use new TLS certificate (`server.pem` in the example) and its key (`server-key.pem`), and again the combined CA certificate (`ca.pem.combined`):

```
kubectl delete secret/my-cluster-name-ssl-internal
kubectl create secret generic my-cluster-name-ssl-internal --from-file=tls.crt=server.pem --from-file=tls.key=server-key.pem --from-file=ca.crt=ca.pem.combined --type=kubernetes.io/tls
```

8. The cluster will go through a rolling reconciliation, but it will do it without problems, as 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. Joiner node also has a combined CA certificate, so it can authenticate against older TLS certificate.
9. Create a final Secrets object: use new TLS certificate (`server.pmm`) and its key (`server-key.pem`), and just the new CA certificate (`ca.pem`):

```
kubectl delete secret/my-cluster-name-ssl-internal
kubectl create secret generic my-cluster-name-ssl-internal --from-file=tls.crt=server.pem --from-file=tls.key=server-key.pem --from-file=ca.crt=ca.pem --type=kubernetes.io/tls
```

10. The cluster will go through a rolling reconciliation, 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.

Update certificates with downtime

If your certificates have been already expired (or if you continue to use the Operator version prior to 1.9.0), you should move through the *pause - update Secrets - unpause* route as follows.

1. Pause the cluster [in a standard way](), and make sure it has reached its paused state.
2. If cert-manager is used, delete issuer and TLS certificates:

```
{
  kubectl delete issuer/my-cluster-name-psmdb-ca-issuer issuer/my-cluster-name-psmdb-issuer
  kubectl delete certificate/my-cluster-name-ssl certificate/my-cluster-name-ssl-internal
  }
```

3. Delete Secrets to force the SSL reconciliation:

```
kubectl delete secret/my-cluster-name-ssl secret/my-cluster-name-ssl-internal
```

4. Check certificates to make sure reconciliation have succeeded.
5. Unpause the cluster [in a standard way](), and make sure it has reached its running state.

Modify certificates generation

There may be reasons to tweak the certificates generation, making it better fit some needs. Of course, maximum flexibility can be obtained with manual certificates generation, but sometimes slight tweaking the already automated job may be enough.

The following example shows how to increase CA duration with cert-manager for a cluster named `cluster1`:

1. Delete the `psmdb` Custom Resource in the proper namespace (this will cause deletion of all Pods of the cluster, but later you will recreate the cluster using the same `deploy/cr.yaml` file from which it was originally created).

> **Note**
>
> you may need to make sure that `finalizers.percona.com/delete-psmdb-pvc` is not set if you want to preserver Persistent Volumes with the data.

Deletion command should look as follows:

```
kubectl -n <namespace_name> delete psmdb cluster1
```

2. Deletion takes time. Check that all Pods disappear with `kubectl -n <namespace_name> get pods` command, and delete certificate related resources:

```
kubectl -n <namespace_name> delete issuer.cert-manager.io/cluster1-psmdb-ca-issuer issuer.cert-manager.io/cluster1-psmdb-issuer certificate.cert-manager.io/cluster1-ssl-internal certificate.cert-manager.io/cluster1-ssl certificate.cert-manager.io/cluster1-ca-cert secret/cluster1-ca-cert secret/cluster1-ssl secret/cluster1-ssl-internal
```

3. Create your own custom CA:

my_new_ca.yml

```yaml
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: cluster1-psmdb-ca-issuer
spec:
  selfSigned: {}
---
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: cluster1-ca-cert
spec:
  commonName: cluster1-ca
  duration: 10000h0m0s
  isCA: true
  issuerRef:
    kind: Issuer
    name: cluster1-psmdb-ca-issuer
  renewBefore: 730h0m0s
  secretName: cluster1-ca-cert
```

Apply it as usual, with the `kubectl -n <namespace_name> apply -f my_new_ca.yml` command.

4. Recreate the cluster from the original `deploy/cr.yaml` configuration file:

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

5. Verify certificate duration in usual way.

7.12.5 Run Percona Server for MongoDB without TLS

You can run Percona Server for MongoDB without TLS. For example, for testing or demonstration purposes. However, we recommend that you have the TLS protocol enabled.

You can start a new cluster without TLS or disable the TLS protocol for a running cluster. See the corresponding sections for steps.

Disable TLS for a new cluster

To disable TLS protocol for a new cluster, edit the `deploy/cr.yaml` Custom Resource manifest as follows:

- set the `tls.mode` key to `disabled`
- set the `unsafeFlags.tls` to `true`.

```
...
spec:
  ...
  unsafeFlags
    tls: true
    ...
  tls:
    mode: disabled
```

Apply the manifest:

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

Disable TLS for a running cluster

To disable TLS protocol for a running cluster, follow these steps:

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 `my-cluster-name`, the command is:

   ```
   kubectl patch psmdb my-cluster-name -n <namespace> --type json -p='[{"op":"add","path":"/spec/pause","value":true}]'
   ```

2. Wait for the cluster to be paused. Check the status with the `kubectl get psmdb` command:

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

   Expected output

   ```
   NAME              ENDPOINT                                          STATUS   AGE
   my-cluster-name   my-cluster-name-mongos.default.svc.cluster.local:27017   paused   3m
   ```

3. Disable the TLS protocol by setting the following configuration in the `deploy/cr.yaml` Custom Resource manifest:

   ```
   ...
   spec:
     ...
     unsafeFlags
       tls: true
       ...
     tls:
       mode: disabled
   ```

4. Apply the changes:

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

5. Now resume the cluster with the `kubectl patch` command:

```
kubectl patch psmdb my-cluster-name -n <namespace> --type json -p='[{"op":"add","path":"/spec/pause","value":false}]'
```

6. Wait for the cluster to be resumed. Check the status with the `kubectl get psmdb` command.

Re-enable TLS

To re-enable TLS protocol for a running cluster, follow these steps:

1. Pause the cluster. Edit the `deploy/cr.yaml` Custom Resource manifest and set `spec.pause` key to `true`:

```
spec:
  pause: true
```

2. Apply the changes:

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

3. Wait for the cluster to be paused. Check the status with the `kubectl get psmdb` command:

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

4. Enable the TLS protocol by setting the following configuration in the `deploy/cr.yaml` Custom Resource manifest:

```
...
spec:
  ...
  unsafeFlags
    tls: false
    ...
  tls:
    mode: preferTLS
```

5. Apply the changes:

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

6. Now resume the cluster. Edit the `deploy/cr.yaml` Custom Resource manifest and set the `spec.pause` key to `false`:

```
spec:
  pause: false
```

7. Apply the changes:

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

8. Wait for the cluster to be resumed. Check the status with the `kubectl get psmdb` command.

> **Note**
>
> If you also need to disable authentication in addition to TLS, see [Disable authentication](). Note that TLS must be disabled before you can disable authentication.

7.13 Data at rest encryption

7.13.1 Data-at-rest encryption

Version added: 1.1.0

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 the application's code. If an unauthorized user gains access to the storage, they can't read the data files.

To learn more about data-at-rest-encryption in Percona Server for MongoDB, see the Data-at-rest encryption documentation.

Data-at-rest encryption is turned on by default when you deploy a new cluster. The Operator implements it in one of the following ways:

- uses an encryption key stored in a Secret
- gets encryption key from the HashiCorp Vault key storage.

Use an encryption key Secret

You can configure data-at-rest encryption in Percona Server for MongoDB using an encryption key stored in a Kubernetes Secret. This is the default approach when you are not using an external key management store such as HashiCorp Vault.

How it works:

The Operator uses a single encryption key to protect MongoDB data files on disk. You tell the Operator which Secret holds that key by setting the `secrets.encryptionKey` option in the cluster Custom Resource.

The Operator creates a Secret with the specified name and generates a random 32-byte key (base64-encoded), if such a Secret doesn't already exist. You can also use your own key and create a Secret object with it; the key must be a 32-character string encoded in base64. Reference your Secret in the Custom Resource for the Operator to use it.

Next, enable encryption and choose the cipher mode in the MongoDB configuration for all Pods where you want to encrypt data: regular replica set members (primary and secondary) as well as arbiter, hidden and/or non-voting replica set members, and config servers if you use sharding. The Operator mounts the key from the Secret into the MongoDB pods so the database can encrypt and decrypt data at rest.

Once configured, encryption is transparent to your applications; no code changes are required.

Use HashiCorp Vault to store and manage encryption keys

Version added: 1.13.0

You can configure the Operator to use HashiCorp Vault - a universal, secure and reliable way to store and distribute secrets without depending on the operating system, platform or cloud provider.

Using Vault for data-at-rest encryption you benefit from:

- **Centralized key management** – Keys live in a dedicated secrets store instead of Kubernetes Secrets. You can enforce different access policies to better comply with your organization's security requirements.
- **Audit and compliance** – Vault can log and audit access to the encryption key, which helps with compliance and security reviews.
- **Flexibility** – You can run Vault on-premises, in the cloud, or as a managed service, independent of the Kubernetes cluster.

  The Operator does not install or configure Vault; you run and manage Vault yourself and then reference it in the cluster Custom Resource.

How it works:

You create a Kubernetes Secret that holds the token that the Operator uses to authenticate to Vault. In the Custom Resource you set `secrets.vault` to that Secret's name and add Vault-related options to the MongoDB configuration (server address, secret path, token mount path, and optionally TLS). The Operator reads the encryption key from Vault and makes it available to the `mongod` pods. Percona Server for MongoDB uses that key to encrypt and decrypt data on disk, the same as with an encryption key stored in a Kubernetes Secret. Encryption remains transparent to your applications.

Supported secrets engine

The Operator expects the encryption key to be stored in Vault's **KV (Key-Value) secrets engine**. Percona Server for MongoDB supports the **KV version 2** (`kv-v2`) with versioning enabled. You enable the secrets engine at a path (for example `secret/`). This is where the encryption keys are stored. The Custom Resource configuration must reference that path so the Operator can read the key. Other Vault secrets engines (for example Transit or KMIP) are not used for this integration.

Configuration guides

Choose a setup guide based on your security requirements:

- Configure data-at-rest encryption using an encryption key Secret
- Configure data-at-rest encryption using HashiCorp Vault without TLS
- Configure data-at-rest encryption using HashiCorp Vault with TLS

7.13.2 Configure data-at-rest encryption using an encryption key Secret

Data-at-rest encryption using a Kubernetes Secret allows you to securely store the encryption key used by Percona Server for MongoDB directly in your cluster, without relying on an external key management service. This guide explains how to configure the Operator to use a Secret for managing your MongoDB encryption key.

Configuration steps

1. Set the encryption key Secret name.

   In your Custom Resource file, set `secrets.encryptionKey` to the name of the Secret that will hold the encryption key. The default Secret name is `my-cluster-name-mongodb-encryption-key`. If you use your own key and Secret, reference that Secret name here:

   ```
   secrets:
     ...
     encryptionKey: my-cluster-name-mongodb-encryption-key
   ```

2. Enable encryption and set the cipher mode in the configuration passed to MongoDB. Set these options in every relevant `configuration` block:

   - `replsets.configuration`,
   - `replsets.nonvoting.configuration` if you use non-voting members,
   - `replsets.hidden.configuration` if you use hidden members,
   - `replsets.arbiter.configuration` if you use arbiter nodes,
   - `sharding.configsvrReplSet.configuration` if you use sharding.

   Add or update the `security` section in each block with this configuration:

   - `enableEncryption` – Set to `true` to turn on data-at-rest encryption (this is the default).
   - `encryptionCipherMode` – Use `AES256-CBC` (default) or `AES256-GCM`.

   ```
   ...
   configuration: |
     ...
     security:
       enableEncryption: true
       encryptionCipherMode: "AES256-CBC"
       ...
   ```

3. Apply the updated Custom Resource:

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

   The Operator creates or uses the encryption key Secret and rolls out the configuration to the MongoDB pods. This triggers the rolling restart of your database Pods. After the rollout completes, data written by MongoDB will be encrypted at rest.

4. Check that the encryption is enabled. Execute into a Percona Server for MongoDB Pod as as a user with sufficient administrative privileges (`databaseAdmin` or `clusterAdmin`) and run the following command against the `admin` database:

   ```
   db.serverStatus().encryptionAtRest
   ```

   Expected output

   ```
   {
     "encryptionEnabled": true,
     "encryptionCipherMode": "AES256-CBC",
     "encryptionKeyId": "local"
   }
   ```

7.13.3 Configure encryption with HashiCorp Vault

7.13.3.1 Configure data-at-rest encryption using HashiCorp Vault without TLS

This guide walks you through deploying and configuring HashiCorp Vault to work with Percona Operator for MongoDB to enable data-at-rest encryption using HTTP protocol.

If you want to add an extra layer of security for communication between Percona Server for MongoDB and the Vault server, you can also set up HashiCorp Vault with TLS enabled. For details, see the guide for configuring Vault with TLS.

> **Important**
>
> You can enable data at rest encryption with HashiCorp Vault only when you create a new cluster. This is because Percona Server for MongoDB doesn't allow enabling / disabling or changing encryption settings for a running cluster. Every time you restart the server, the encryption settings must be the same. When you define Vault configuration for a cluster, the Operator uses that instead of the default encryption key Secret.

Assumptions

1. This guide is provided as a best effort and builds upon procedures described in the official Vault documentation. Since Vault's setup steps may change in future releases, this document may become outdated; we cannot guarantee ongoing accuracy or responsibility for such changes. For the most up-to-date and reliable information, please always refer to the official Vault documentation.
2. In the following sections we deploy the Vault server in High Availability (HA) mode on Kubernetes via Helm with TLS enabled. The HA setup uses Raft storage backend and consists of 3 replicas for redundancy. Using Helm is not mandatory. Any supported Vault deployment (on-premises, in the cloud, or a managed Vault service) works as long as the Operator can reach it.
3. This guide uses Vault Helm chart version 0.30.0. You may want to change it to the required version by setting the `VAULT_HELM_VERSION` variable.

Prerequisites

Before you begin, ensure you have the following tools installed:

- `kubectl` - Kubernetes command-line interface
- `helm` - Helm package manager
- `jq` - JSON processor

Prepare your environment

1. Export the namespace and other variables as environment variables to simplify further configuration:

```
export NAMESPACE="vault"
export CLUSTER_NAMESPACE="psmdb" # The namespace where the database cluster will be deployed
export VAULT_HELM_VERSION="0.30.0"
export SERVICE="vault"
export POLICY_NAME="psmdb-policy"
export WORKDIR="/tmp/vault"
```

2. Create a working directory for configuration files:

```
mkdir -p $WORKDIR
```

3. It is a good practice to isolate workloads in Kubernetes using namespaces. Create namespaces with the following command:

   - For Vault server:

```
kubectl create namespace vault
```

   - For Percona Server for MongoDB cluster:

```
kubectl create namespace psmdb
```

Install Vault

For this setup, we install Vault in Kubernetes using the Helm 3 package manager in High Availability (HA) mode with Raft storage backend.

1. Add and update the Vault Helm repository:

```
helm repo add hashicorp https://helm.releases.hashicorp.com
helm repo update
```

2. Install Vault in HA mode:

```
helm upgrade --install ${SERVICE} hashicorp/vault \
  --disable-openapi-validation \
  --version ${VAULT_HELM_VERSION} \
  --namespace ${NAMESPACE} \
  --set "global.enabled=true" \
  --set "global.tlsDisable=true" \
  --set "global.platform=kubernetes" \
  --set "server.ha.enabled=true" \
  --set "server.ha.replicas=3" \
  --set "server.ha.raft.enabled=true" \
  --set "server.ha.raft.setNodeId=true" \
  --set-string "server.ha.raft.config=cluster_name = \"vault-integrated-storage\"
ui = true
listener \"tcp\" {
  tls_disable = 1
  address = \"[::]:8200\"
  cluster_address = \"[::]:8201\"
}
storage \"raft\" {
  path = \"/vault/data\"
}
disable_mlock = true
service_registration \"kubernetes\" {}"
```

This command does the following:

- Installs HashiCorp Vault in High Availability (HA) mode without TLS in your Kubernetes cluster
- Sets up Raft as the backend storage with three replicas for fault tolerance
- Configures the Vault TCP listener for HTTP communication (port 8200)

Sample output

```
NAME: vault
LAST DEPLOYED: Wed Feb 11 17:06:00 2026
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

Your release is named vault. To learn more about the release, try:

  $ helm status vault
  $ helm get manifest vault
```

3. Wait for all Vault pods to be running:

```
kubectl wait --for=jsonpath='{.status.phase}'=Running pod \
-l app.kubernetes.io/name=${SERVICE} -n $NAMESPACE --timeout=300s
```

4. Retrieve the Pod names where Vault is running:

```
kubectl -n $NAMESPACE get pod -l app.kubernetes.io/name=${SERVICE} -o jsonpath='{range .items[*]}{.metadata.name}{"\n"}
{end}'
```

Sample output

```
vault-0
vault-1
vault-2
```

Initialize and unseal Vault

1. After Vault is installed, you need to initialize it. Run the following command to initialize the first pod:

```
kubectl exec -it pod/vault-0 -n $NAMESPACE -- vault operator init -key-shares=1 -key-threshold=1 -format=json >
${WORKDIR}/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 to a local file. The file includes unseal keys and a root token.

2. 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[]" < ${WORKDIR}/vault-init)
```

Now, unseal the first Vault pod:

```
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    raft
Cluster Name    vault-cluster-55062a37
Cluster ID      37d0c2e4-8f47-14f7-ca49-905b66a1804d
HA Enabled      true
```

3. Add the remaining Pods to the Vault cluster. Run the following for loop:

```
for POD in vault-1 vault-2; do
  kubectl -n "$NAMESPACE" exec $POD -- sh -c '
    vault operator raft join http://vault-0.vault-internal:8200
  '
done
```

The command connects to each Vault Pod (`vault-1` and `vault-2`) and issues the `vault operator raft join` command, which:

- Joins the Pods to the Vault Raft cluster, enabling HA mode
- Connects to the cluster leader (`vault-0`) over HTTP
- Ensures all nodes participate in the Raft consensus and share storage responsibilities

Sample output

```
Key                    Value
---                    -----
Joined Raft cluster    true
Leader Address         http://vault-0.vault-internal:8200
```

4. Unseal the remaining Pods. Use this for loop:

```
for POD in vault-1 vault-2; do
    kubectl -n "$NAMESPACE" exec $POD -- sh -c "
        vault operator unseal \"$unsealKey\"
    "
done
```

Expected 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       raft
HA Enabled         true
```

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.

Run the following command on a leader node. The remaining ones will synchronize from the leader.

1. Extract the Vault root token from the file where you saved the init response output:

```
cat ${WORKDIR}/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 `secret` mount-path:

```
vault secrets enable --version=2 -path=secret kv
```

Sample output

```
Success! Enabled the kv secrets engine at: secret/
```

5. (Optional) You can also enable audit. This is not mandatory, but useful:

```
vault audit enable file file_path=/vault/vault-audit.log
```

Expected output

```
Success! Enabled the file audit device at: file/
```

6. Exit the Vault Pod:

```
exit
```

Create a non-root token

Using the root token for authentication is not recommended, as it poses significant security risks. Instead, you should create a dedicated, non-root token for the Operator to use when accessing Vault. The permissions for this token are controlled by an access policy. Before you create a token you must first create the access policy.

1. Create a policy for accessing the kv engine path and define the required permissions in the `capabilities` parameter:

```
kubectl -n "$NAMESPACE" exec vault-0 -- sh -c "
  vault policy write $POLICY_NAME - <<EOF
path \"secret/data/*\" {
  capabilities = [\"create\", \"read\", \"update\", \"delete\", \"list\"]
}
path \"secret/metadata/*\" {
  capabilities = [\"read\"]
}
path \"secret/*\" {
  capabilities = [\"read\"]
}
EOF
"
```

2. Now create a token with a policy.

```
kubectl -n "${NAMESPACE}" exec pod/vault-0 -- vault token create -policy="${POLICY_NAME}" -format=json > "${WORKDIR}/vault-token.json"
```

3. Export the non-root token as an environment variable:

```
export NEW_TOKEN=$(jq -r '.auth.client_token' "${WORKDIR}/vault-token.json") && echo "$NEW_TOKEN"
```

Sample output

```
hvs.CAESINO******************************************T2Y
```

Create a Secret for Vault

To enable Vault for the Operator, create a Secret object for it using the Vault token. Note that you must create the Secret in the namespace where the Operator and the database cluster is running.

Run the following command:

```
kubectl create secret generic my-cluster-name-vault --from-literal=token=$NEW_TOKEN -n $CLUSTER_NAMESPACE
```

Check that the Secret is created:

```
kubectl get secret -n $CLUSTER_NAMESPACE
```

Deploy the Operator

Follow the Quickstart guide to install the Operator Deployment.

Check that it is up and running with this command:

```
kubectl get deploy -n $CLUSTER_NAMESPACE
```

Sample output

```
NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
percona-server-mongodb-operator 1/1     1            1           162m
```

Reference the Secret in your Custom Resource manifest

Now, reference the Vault Secret in the Operator Custom Resource manifest. You also need the following Vault-related information:

- A Vault server name and port. If Vault is deployed in a separate namespace, use the fully qualified name in the format `<service-name>.<namespace>.svc.cluster.local` (`vault.vault.svc.cluster.local` in our example)
- Path to the token file. When you apply the new configuration, the Operator creates the required directories and places the token file there.
- The secrets mount path in the format `<mount-path>/data/dc/<cluster name>/<path>`, where:
  - the `<cluster name>` is your real cluster name.
  - the `<path>` is where encryption keys are stored. Specify the replica set name value (`rs0` in our example) when you add options to the `replsets.configuration` section. Specify the `cfg` value when you add options to the `sharding.configsvrReplSet.configuration` section.

Note

In a sharded cluster, you must specify the Vault configuration in both `replsets.configuration` and `sharding.configsvrReplSet.configuration` sections. If you use arbiter, non-voting and / or hidden nodes and want to encrypt the data there, specify the Vault configuration for these members too.

1. Modify your `deploy/cr.yaml` as follows:
    a. Set the `secrets.vault` key to the name of your Secret created on the previous step.
    b. Add Vault-specific options to the `replsets.configuration`, `replsets.nonvoting.configuration`, and `sharding.configsvrReplSet.configuration` keys:

```yaml
...
secrets:
  vault: my-cluster-name-vault
...
replsets:
- name: rs0
  size: 3
...
  configuration: |
     security:
       enableEncryption: true
       vault:
         serverName: vault.vault.svc.cluster.local
         port: 8200
         tokenFile: /etc/mongodb-vault/token
         secret: secret/data/dc/<cluster name>/rs0
         disableTLSForTesting: true
       ...
sharding:
  configsvrReplSet:
    ...
    configuration: |
      security:
        enableEncryption: true
        vault:
          serverName: vault.vault.svc.cluster.local
          port: 8200
          tokenFile: /etc/mongodb-vault/token
          secret: secret/data/dc/<cluster name>/cfg
          disableTLSForTesting: true
         ...
```

2. Apply your modified `cr.yaml` file:

```
kubectl apply -f deploy/cr.yaml -n $CLUSTER_NAMESPACE
```

Verify encryption

Check that the encryption is enabled. Execute into a Percona Server for MongoDB Pod as as a user with sufficient administrative privileges (`databaseAdmin` or `clusterAdmin`) and run the following command against the `admin` database:

```
db.serverStatus().encryptionAtRest
```

Expected output

```
{
  encryptionEnabled: true,
  encryptionCipherMode: 'AES256-CBC',
  encryptionKeyId: {
    vault: { path: 'secret/data/dc/my-cluster-name/rs0', version: '4' }
  }
}
```

7.13.3.2 Configure data-at-rest encryption using HashiCorp Vault with TLS

This guide walks you through deploying and configuring HashiCorp Vault to work with Percona Operator for MongoDB to enable data-at-rest encryption using HTTPS protocol.

By default, Percona Server for MongoDB and Vault communicate over an unencrypted HTTP protocol. You can enable encrypted HTTPS protocol with TLS as an additional security layer to protect the data transmitted between Vault and your Percona Server for MongoDB nodes. HTTPS ensures that sensitive information, such as encryption keys and secrets, cannot be intercepted or tampered with on the network.

> **Important**
>
> You can enable data at rest encryption with HashiCorp Vault only when you create a new cluster. This is because Percona Server for MongoDB doesn't allow enabling / disabling or changing encryption settings for a running cluster. Every time you restart the server, the encryption settings must be the same. When you define Vault configuration for a cluster, the Operator uses that instead of the default encryption key Secret.

Assumptions

1. This guide is provided as a best effort and builds upon procedures described in the official Vault documentation. Since Vault's setup steps may change in future releases, this document may become outdated; we cannot guarantee ongoing accuracy or responsibility for such changes. For the most up-to-date and reliable information, please always refer to the official Vault documentation.
2. In the following sections we deploy the Vault server in High Availability (HA) mode on Kubernetes via Helm with TLS enabled. The HA setup uses Raft storage backend and consists of 3 replicas for redundancy. Using Helm is not mandatory. Any supported Vault deployment (on-premises, in the cloud, or a managed Vault service) works as long as the Operator can reach it.
3. This guide uses Vault Helm chart version 0.30.0. You may want to change it to the required version by setting the `VAULT_HELM_VERSION` variable.

Prerequisites

Before you begin, ensure you have the following tools installed:

- `kubectl` - Kubernetes command-line interface
- `helm` - Helm package manager
- `jq` - JSON processor

Prepare your environment

1. Export the namespace and other variables as environment variables to simplify further configuration:

   ```
   export NAMESPACE="vault"
   export CLUSTER_NAMESPACE="psmdb"
   export VAULT_HELM_VERSION="0.30.0"
   export SERVICE="vault"
   export CSR_NAME="vault-csr"
   export SECRET_NAME_VAULT="vault-secret"
   export POLICY_NAME="psmdb-policy"
   export WORKDIR="/tmp/vault"
   ```

2. Create a working directory for configuration files:

   ```
   mkdir -p $WORKDIR
   ```

3. It is a good practice to isolate workloads in Kubernetes using namespaces. Create namespaces with the following command:

   - For Vault server:

   ```
   kubectl create namespace vault
   ```

   - For Percona Server for MongoDB cluster:

```
kubectl create namespace psmdb
```

Generate TLS certificates

To use TLS, you'll need the following certificates:

- A private key for the Vault server
- A certificate for the Vault server signed by the Kubernetes CA
- The Kubernetes CA certificate

These files store sensitive information. Make sure to keep them in a secure location.

Generate the private key

Generate a private key for the Vault server:

```
openssl genrsa -out ${WORKDIR}/vault.key 2048
```

Create the Certificate Signing Request (CSR)

A Certificate Signing Request (CSR) is a file that contains information about your server and the certificate you need. You create it using your private key, and then submit it to Kubernetes to get a certificate signed by the Kubernetes Certificate Authority (CA). The signed certificate proves your server's identity and enables secure TLS connections.

1. Create the Certificate Signing Request configuration file:

   Specify the certificate details that Kubernetes needs to sign your certificate:

   - **Request settings** (`[req]`): References the sections for certificate extensions and distinguished name. The distinguished name section is left empty as Kubernetes will populate it automatically.
   - **Certificate extensions** (`[v3_req]`): Defines how the certificate can be used. `serverAuth` allows the certificate for server authentication, while `keyUsage` specifies the cryptographic operations the certificate supports (non-repudiation, digital signature, and key encipherment).
   - **Subject Alternative Names** (`[alt_names]`): Lists all DNS names and IP addresses where your Vault service can be accessed. This includes the service name, fully qualified domain names (FQDNs) for different Kubernetes DNS contexts (namespace-scoped, cluster-scoped with `.svc`, and fully qualified with `.svc.cluster.local`), and the localhost IP address.

   ```
   cat > "${WORKDIR}/csr.conf" <<'EOF'
   [req]
   default_bits = 2048
   prompt = no
   encrypt_key = yes
   default_md = sha256
   distinguished_name = kubelet_serving
   req_extensions = v3_req
   [ kubelet_serving ]
   O = system:nodes
   CN = system:node:*.vault.svc.cluster.local
   [ v3_req ]
   basicConstraints = CA:FALSE
   keyUsage = nonRepudiation, digitalSignature, keyEncipherment, dataEncipherment
   extendedKeyUsage = serverAuth, clientAuth
   subjectAltName = @alt_names
   [alt_names]
   DNS.1 =*.vault-internal
   DNS.2 = *.vault-standby
   DNS.3 =*.vault-internal.vault.svc.cluster.local
   DNS.4 = *.vault-standby.vault.svc.cluster.local
   DNS.5 =*.vault
   DNS.6 = vault.vault.svc.cluster.local
   IP.1 = 127.0.0.1
   EOF
   ```

2. Generate the CSR. The following command creates the Certificate Signing Request file using your private key and the configuration file.

The `-subj` parameter specifies the distinguished name directly: the Common Name (CN) identifies your Vault service using the Kubernetes node naming convention (`system:node:${SERVICE}.${NAMESPACE}.svc`), and the Organization (O) field is set to `system:nodes`, which Kubernetes requires to recognize and sign the certificate. The command combines these subject fields with the certificate extensions defined in the configuration file to produce the complete CSR.

```
openssl req -new -key $WORKDIR/vault.key \
  -subj "/CN=system:node:${SERVICE}.${NAMESPACE}.svc;/O=system:nodes" \
  -out $WORKDIR/server.csr -config $WORKDIR/csr.conf
```

Issue the certificate

To get your certificate signed by Kubernetes, you need to submit the CSR through the Kubernetes API. The CSR file you generated with OpenSSL must be wrapped in a Kubernetes CertificateSigningRequest resource.

1. Create the CSR YAML file to send it to Kubernetes:

   This YAML file creates a Kubernetes CertificateSigningRequest object that contains your CSR. The file embeds the base64-encoded CSR content and specifies:

   - The signer name (`kubernetes.io/kubelet-serving`) that tells Kubernetes which CA should sign the certificate
   - The groups field (`system:authenticated`) that identifies who can approve this CSR
   - The certificate usages that define how the certificate can be used (digital signature, key encipherment, and server authentication)

   ```
   cat > $WORKDIR/csr.yaml <<EOF
   apiVersion: certificates.k8s.io/v1
   kind: CertificateSigningRequest
   metadata:
     name: ${CSR_NAME}
   spec:
     groups:
     - system:authenticated
     request: $(cat $WORKDIR/server.csr | base64 | tr -d '\n')
     signerName: kubernetes.io/kubelet-serving
     usages:
     - digital signature
     - key encipherment
     - server auth
   EOF
   ```

2. Create the CertificateSigningRequest (CSR) object:

   ```
   kubectl create -f ${WORKDIR}/csr.yaml
   ```

3. Approve the CSR in Kubernetes:

   ```
   kubectl certificate approve ${CSR_NAME}
   ```

4. Confirm the certificate was issued:

   ```
   kubectl get csr ${CSR_NAME}
   ```

   Sample output

   ```
   NAME        AGE   SIGNERNAME                      REQUESTOR       REQUESTEDDURATION   CONDITION
   vault-csr   16s   kubernetes.io/kubelet-serving   minikube-user   <none>              Approved,Issued
   ```

Retrieve the certificates

After Kubernetes approves and signs your CSR, you need to retrieve the signed certificate and the Kubernetes CA certificate. These certificates are required to configure TLS for your Vault server.

1. Retrieve the signed certificate from the CertificateSigningRequest object. The certificate is base64-encoded in Kubernetes, so you decode it and save it to a file.

```
kubectl get csr ${CSR_NAME} -o jsonpath='{.status.certificate}' | base64 -d > $WORKDIR/vault.crt
```

2. Retrieve Kubernetes CA certificate:

   This command retrieves the Kubernetes cluster's Certificate Authority (CA) certificate from your `kubeconfig` file. The CA certificate is needed to verify that the signed certificate is valid and was issued by the Kubernetes CA. The command uses `kubectl config view` with flags to get the raw, flattened configuration and extract the CA certificate data, which is also base64-encoded.

```
kubectl config view \
  --raw \
  --minify \
  --flatten \
  -o jsonpath='{.clusters[].cluster.certificate-authority-data}' \
  | base64 -d > ${WORKDIR}/vault.ca
```

Store certificates in Kubernetes secrets

Create a TLS secret in Kubernetes to store the certificates and key:

```
kubectl create secret generic ${SECRET_NAME_VAULT} \
  --namespace ${NAMESPACE} \
  --from-file=vault.key=$WORKDIR/vault.key \
  --from-file=vault.crt=$WORKDIR/vault.crt \
  --from-file=vault.ca=$WORKDIR/vault.ca
```

Install Vault with TLS

For this setup, we install Vault in Kubernetes using the [Helm 3 package manager](#) in High Availability (HA) mode with Raft storage backend and with TLS enabled.

1. Add and update the Vault Helm repository:

```
helm repo add hashicorp https://helm.releases.hashicorp.com
helm repo update
```

2. Install Vault with TLS enabled:

```
helm upgrade --install ${SERVICE} hashicorp/vault \
  --disable-openapi-validation \
  --version ${VAULT_HELM_VERSION} \
  --namespace ${NAMESPACE} \
  --set "global.enabled=true" \
  --set "global.tlsDisable=false" \
  --set "global.platform=kubernetes" \
  --set server.extraEnvironmentVars.VAULT_CACERT=/vault/userconfig/${SECRET_NAME_VAULT}/vault.ca \
  --set "server.extraEnvironmentVars.VAULT_TLSCERT=/vault/userconfig/${SECRET_NAME_VAULT}/vault.crt" \
  --set "server.extraEnvironmentVars.VAULT_TLSKEY=/vault/userconfig/${SECRET_NAME_VAULT}/vault.key" \
  --set "server.volumes[0].name=userconfig-${SECRET_NAME_VAULT}" \
  --set "server.volumes[0].secret.secretName=${SECRET_NAME_VAULT}" \
  --set "server.volumes[0].secret.defaultMode=420" \
  --set "server.volumeMounts[0].mountPath=/vault/userconfig/${SECRET_NAME_VAULT}" \
  --set "server.volumeMounts[0].name=userconfig-${SECRET_NAME_VAULT}" \
  --set "server.volumeMounts[0].readOnly=true" \
  --set "server.ha.enabled=true" \
  --set "server.ha.replicas=3" \
  --set "server.ha.raft.enabled=true" \
  --set "server.ha.raft.setNodeId=true" \
  --set-string "server.ha.raft.config=cluster_name = \"vault-integrated-storage\"
ui = true
listener \"tcp\" {
  tls_disable = 0
  address = \"[::]:8200\"
  cluster_address = \"[::]:8201\"
  tls_cert_file = \"/vault/userconfig/${SECRET_NAME_VAULT}/vault.crt\"
  tls_key_file  = \"/vault/userconfig/${SECRET_NAME_VAULT}/vault.key\"
  tls_client_ca_file = \"/vault/userconfig/${SECRET_NAME_VAULT}/vault.ca\"
}
storage \"raft\" {
  path = \"/vault/data\"
}
disable_mlock = true
service_registration \"kubernetes\" {}"
```

This command does the following:

- Installs HashiCorp Vault in High Availability (HA) mode with secure TLS enabled in your Kubernetes cluster.
- Configures Vault pods to use certificates from a Kubernetes Secret via volume mounts for secure HTTPS communication between Vault and clients.
- Sets up Raft as the backend storage with three replicas for fault tolerance, and configures the Vault TCP listener to enforce TLS with your specified certificate files.

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 $NAMESPACE get pod -l app.kubernetes.io/name=${SERVICE} -o jsonpath='{range .items[*]}{.metadata.name}{"\n"}
{end}'
```

Sample output

```
vault-0
vault-1
vault-2
```

Initialize and unseal Vault

1. After Vault is installed, you need to initialize it. Run the following command to initialize the first pod:

   ```
   kubectl exec -it pod/vault-0 -n $NAMESPACE -- vault operator init -key-shares=1 -key-threshold=1 -format=json >
   ${WORKDIR}/vault-init
   ```

   The command does the following:

   - Connects to the Vault Pod
   - Initializes Vault server with TLS enabled
   - Creates 1 unseal key share which is required to unseal the server
   - Outputs the init response to a local file. The file includes unseal keys and root token.

2. 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[]" < ${WORKDIR}/vault-init)
   ```

   Now, unseal Vault. Run the following command:

   ```
   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                  1.19.0
   Build Date               2025-03-04T12:36:40Z
   Storage Type             raft
   Cluster Name             vault-integrated-storage
   Cluster ID               ed275c91-e227-681b-5aaa-f7a9fc19e37e
   Removed From Cluster     false
   HA Enabled               true
   HA Cluster               <https://vault-0.vault-internal:8201>
   HA Mode                  active
   Active Since             2025-12-15T13:36:42.542059059Z
   Raft Committed Index     37
   Raft Applied Index       37
   ```

3. Add the remaining Pods to the Vault cluster. If you have another secret name, replace the `vault-secret` with your value in the following for loop:

   ```
   for POD in vault-1 vault-2; do
     kubectl -n "$NAMESPACE" exec $POD -- sh -c '
       vault operator raft join -address=https://${HOSTNAME}.vault-internal:8200 \
         -leader-ca-cert="$(cat /vault/userconfig/vault-secret/vault.ca)" \
         -leader-client-cert="$(cat /vault/userconfig/vault-secret/vault.crt)" \
         -leader-client-key="$(cat /vault/userconfig/vault-secret/vault.key)" \
         https://vault-0.vault-internal:8200;
     '
   done
   ```

   The command connects to each Vault Pod (`vault-1` and `vault-2`) and issues the `vault operator raft join` command, which: * Joins the Pods to the Vault Raft cluster, enabling HA mode. * Uses the necessary TLS certificates to securely connect to the cluster leader (`vault-0`). * Ensures all nodes participate in the Raft consensus and share storage responsibilities.

   Sample output

   ```
   Key       Value
   ---       -----
   Joined    true
   ```

4. Unseal the remaining Pods. Use this for loop:

```
for POD in vault-1 vault-2; do
    kubectl -n "$NAMESPACE" exec $POD -- sh -c "
        vault operator unseal \"$unsealKey\"
    "
done
```

Expected output

```
Key                      Value
---                      -----
Seal Type                shamir
Initialized              true
Sealed                   true
Total Shares             1
Threshold                1
Unseal Progress          0/1
Unseal Nonce             n/a
Version                  1.19.0
Build Date               2025-03-04T12:36:40Z
Storage Type             raft
Removed From Cluster     false
HA Enabled               true
```

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.

Run the following command on a leader node. The remaining ones will synchronize from the leader.

1. Extract the Vault root token from the file where you saved the init response output:

```
cat ${WORKDIR}/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 `secret` mount-path:

```
vault secrets enable --version=2 -path=secret kv
```

Sample output

```
Success! Enabled the kv secrets engine at: secret/
```

5. (Optional) You can also enable audit. This is not mandatory, but useful:

```
vault audit enable file file_path=/vault/vault-audit.log
```

Expected output

```
Success! Enabled the file audit device at: file/
```

6. Exit the Vault Pod:

```
exit
```

Create a non-root token

Using the root token for authentication is not recommended, as it poses significant security risks. Instead, you should create a dedicated, non-root token for the Operator to use when accessing Vault. The permissions for this token are controlled by an access policy. Before you create a token you must first create the access policy.

1. Create a policy for accessing the kv engine path and define the required permissions in the `capabilities` parameter:

```
kubectl -n "$NAMESPACE" exec vault-0 -- sh -c "
  vault policy write $POLICY_NAME - <<EOF
path \"secret/data/*\" {
  capabilities = [\"create\", \"read\", \"update\", \"delete\", \"list\"]
}
path \"secret/metadata/*\" {
  capabilities = [\"read\"]
}
path \"secret/*\" {
  capabilities = [\"read\"]
}
EOF
"
```

2. Now create a token with a policy.

```
kubectl -n "${NAMESPACE}" exec pod/vault-0 -- vault token create -policy="${POLICY_NAME}" -format=json > "${WORKDIR}/vault-token.json"
```

3. Export the non-root token as an environment variable:

```
export NEW_TOKEN=$(jq -r '.auth.client_token' "${WORKDIR}/vault-token.json")
```

4. Verify the token:

```
echo "New Vault Token: $NEW_TOKEN"
```

Sample output

```
hvs.CAESINO******************************************T2Y
```

Create a Secret for Vault

To enable Vault for the Operator, create a Secret object for it using the Vault token and the path to TLS certificates. Note that you must create the Secret in the namespace where the Operator and the database cluster is running.

Run the following command:

```
kubectl create secret generic my-cluster-name-vault --from-literal=token=$NEW_TOKEN --from-file=ca.crt=${WORKDIR}/vault.ca -n $CLUSTER_NAMESPACE
```

Check that the Secret is created:

```
kubectl get secret -n $CLUSTER_NAMESPACE
```

Deploy the Operator

Follow the Quickstart guide to install the Operator Deployment.

Check that it is up and running with this command:

```
kubectl get deploy -n $CLUSTER_NAMESPACE
```

Sample output

```
NAME                              READY   UP-TO-DATE   AVAILABLE   AGE
percona-server-mongodb-operator   1/1     1            1           162m
```

Reference the Secret in your Custom Resource manifest

Now, reference the Vault Secret in the Operator Custom Resource manifest. You also need the following Vault-related information:

- A Vault server name and port. If Vault is deployed in a separate namespace, use the fully qualified name in the format `<service-name>.<namespace>.svc.cluster.local`.
- Path to the token file. When you apply the new configuration, the Operator creates the required directories and places the token file there.
- Path to TLS certificates
- Contents of the `ca.cert` certificate file
- The secrets mount path in the format `<mount-path>/data/dc/<cluster name>/<path>`, where:
- the `<cluster name>` is your real cluster name
- the `<path>` is where keys are stored. Specify the `rs0` value when you add options to the `replsets.configuration` and `replsets.nonvoting.configuration` sections. Specify the `cfg` value when you add options to the `sharding.configsvrReplSet.configuration` section.

Note

In a sharded cluster, you must specify the Vault configuration in both `replsets.configuration` and `sharding.configsvrReplSet.configuration` sections.

1. Modify your `deploy/cr.yaml` as follows:
    a. Set the `secrets.vault` key to the name of your Secret created on the previous step.
    b. Add Vault-specific options to the `replsets.configuration`, `replsets.nonvoting.configuration`, and `sharding.configsvrReplSet.configuration` keys:

```yaml
secrets:
  vault: my-cluster-name-vault
...
replsets:
- name: rs0
  size: 3
...
  configuration: |
     security:
       enableEncryption: true
       vault:
         serverName: vault.vault.svc.cluster.local
         port: 8200
         tokenFile: /etc/mongodb-vault/token
         secret: secret/data/dc/<cluster name>/rs0
         serverCAFile: /etc/mongodb-vault/ca.crt
       ...
sharding:
  configsvrReplSet:
    ...
    configuration: |
      security:
        enableEncryption: true
        vault:
          serverName: vault.vault.svc.cluster.local
          port: 8200
          tokenFile: /etc/mongodb-vault/token
          secret: secret/data/dc/<cluster name>/cfg
          serverCAFile: /etc/mongodb-vault/ca.crt
        ...
```

2. Apply your modified `cr.yaml` file:

```
kubectl apply -f deploy/cr.yaml -n $CLUSTER_NAMESPACE
```

Verify encryption

Check that the encryption is enabled. Execute into a Percona Server for MongoDB Pod as as a user with sufficient administrative privileges (`databaseAdmin` or `clusterAdmin`) and run the following command against the `admin` database:

```
db.serverStatus().encryptionAtRest
```

Expected output

```
{
  encryptionEnabled: true,
  encryptionCipherMode: 'AES256-CBC',
  encryptionKeyId: {
    vault: { path: 'secret/data/dc/my-cluster-name/rs0', version: '4' }
  }
}
```

7.14 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)
- Is PMM enabled, and the PMM Version
- Operator version
- Mongo version
- Percona Backup for MongoDB (PBM) version
- Is sharding enabled (starting from the Operator version 1.13)
- Is Hashicorp Vault enabled (starting from the Operator version 1.13)
- Is the Operator deployed in a cluster-wide mode (starting from the Operator version 1.13)
- Is Volume Expansion enabled (starting from the Operator version 1.19)
- Are multi-cluster Services enabled (starting from the Operator version 1.19)
- Does the Operator manage custom MongoDB users and/or custom MongoDB roles (starting from the Operator version 1.19)
- Is the Operator deployed with Helm
- Are sidecar containers used
- Are backups used, are point-in-time recovery and/or scheduled physical backup features used, if so
- How large is the cluster

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:

- if you install the Operator with helm, use the following installation command:

```
helm install my-db percona/psmdb-db --version 1.22.0 --namespace my-namespace --set disable_telemetry="true"
```

- if you don't use helm for installation, you have to 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"
  ...
```

7.15 Configure Operator environment variables

You can configure the Percona Operator for MongoDB by setting environment variables in the Operator Deployment. This lets you tune logging, scope the namespaces that are watched, and adjust reconciliation concurrency without rebuilding images.

You can set environment variables in the following ways:

- For installations via `kubectl`, edit the Operator Deployment manifest (`deploy/operator.yaml`) before applying it, or modify the existing Deployment using `kubectl patch` or `kubectl edit`.
- For Helm installations, set environment variables through Helm values.
- For installations on OpenShift, configure environment variables through the OLM 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"
```

`LOG_LEVEL`

Sets the verbosity level of Operator logs.

Value type	Default	Example
string	`"INFO"`	`"DEBUG"`

Valid values are:

- “DEBUG” - Most verbose, includes detailed debugging information
- “INFO” - Standard informational messages (default)
- “WARN” - Warning messages only
- “ERROR” - Error messages only

Example configuration:

```
env:
  - name: LOG_LEVEL
    value: "DEBUG"
```

`WATCH_NAMESPACE`

Specifies which namespaces the Operator watches for Custom Resources.

Value type	Default	Example
string	Operator namespace	`"psmdb,psmdb-prod"` or `""`

Notes:

* A comma-separated list limits the Operator to those namespaces.
* An empty string (`""`) enables cluster-wide mode (watch all namespaces).

Example configuration for multi-namespace mode:

```
env:
  - name: WATCH_NAMESPACE
    value: "psmdb,psmdb-prod"
```

`DISABLE_TELEMETRY`

Disables telemetry data collection.

Value type	Default	Example
string	`"false"`	`"true"`

Example configuration:

```
env:
  - name: DISABLE_TELEMETRY
    value: "true"
```

`MAX_CONCURRENT_RECONCILES`

Controls the maximum number of concurrent reconciliation operations.

Value type	Default	Example
string	`"1"`	`"3"`

Example configuration:

```
env:
  - name: MAX_CONCURRENT_RECONCILES
    value: "3"
```

Automatic environment variables

The following values are set by Kubernetes or the deployment manifest and should not be changed:

* `POD_NAME` - Set from `metadata.name`.
* `OPERATOR_NAME` - Set to `percona-server-mongodb-operator`.

Update environment variables

Using `kubectl patch`

To update environment variables and keep the existing ones, use the full `env` list in your patch:

```
kubectl patch deployment percona-server-mongodb-operator \
  -p '{"spec":{"template":{"spec":{"containers":[{"name":"percona-server-mongodb-operator","env":[
    {"name":"POD_NAME","valueFrom":{"fieldRef":{"fieldPath":"metadata.name"}}},
    {"name":"OPERATOR_NAME","value":"percona-server-mongodb-operator"},
    {"name":"LOG_LEVEL","value":"DEBUG"}
  ]}]}}}}'
```

Using `kubectl edit`

```
kubectl edit deployment percona-server-mongodb-operator
```

Then update the `env` section in the container specification.

7.16 Cluster component environment variables

The Operator sets specific environment variables on cluster components to pass service metadata, authentication details, and integration settings. Use this page as a reference for what is injected by default. If you want to add your own variables, see Define custom environment variables.

MongoDB (`mongod`) containers

These variables are injected into every MongoDB container (replica sets and config servers):

Variable	Purpose
`SERVICE_NAME`	Cluster name.
`NAMESPACE`	Kubernetes namespace of the cluster.
`MONGODB_PORT`	Port of the `mongod` process.
`MONGODB_REPLSET`	Replica set name.
`LOGCOLLECTOR_ENABLED`	`"true"` when log collector is enabled.

`mongos` containers (sharding)

Variable	Purpose
`MONGODB_PORT`	Port of the `mongos` process.

Backup agent sidecar (PBM)

These variables are injected into the PBM backup agent sidecar:

Variable	Purpose
`PBM_AGENT_MONGODB_USERNAME`	Backup user name (from Secret).
`PBM_AGENT_MONGODB_PASSWORD`	Backup user password (from Secret).
`PBM_MONGODB_REPLSET`	Replica set name.
`PBM_MONGODB_PORT`	MongoDB port.
`PBM_AGENT_SIDECAR`	Marks the container as a sidecar.
`PBM_AGENT_SIDECAR_SLEEP`	Sidecar retry delay (seconds).
`SHARDED`	Set to `"TRUE"` when sharding is enabled.
`POD_NAME`	Pod name (from `metadata.name`).
`PBM_MONGODB_URI`	MongoDB connection string URI constructed by the Operator.
`PBM_AGENT_TLS_ENABLED`	`"true"` when TLS is enabled.

Scheduled backup job

The scheduled backup job container gets the namespace so it can create backup objects:

Variable	Purpose
`NAMESPACE`	Namespace where the backup is created.

PMM sidecar

The PMM sidecar supports two modes (PMM2 and PMM3). The Operator injects the following variables.

PMM2

Variable	Purpose
`PMM_SERVER`	PMM server address.
`DB_TYPE`	Database type (`mongodb`).
`DB_USER`	Monitoring user (from Secret).
`DB_PASSWORD`	Monitoring password (from Secret).
`DB_HOST`	MongoDB host (defaults to `localhost`).
`DB_CLUSTER`	Cluster name.
`DB_PORT`	MongoDB port.
`DB_PORT_MIN`	Minimum PMM agent port.
`DB_PORT_MAX`	Maximum PMM agent port.
`PMM_USER`	PMM user for custom login (optional).
`PMM_PASSWORD`	PMM password or API key (optional).
`CLUSTER_NAME`	Cluster name (or custom PMM cluster name).
`PMM_AGENT_PRERUN_SCRIPT`	PMM pre-run script.
`POD_NAME`	Pod name.
`POD_NAMESPASE`	Pod namespace (note the spelling).
`PMM_AGENT_SERVER_ADDRESS`	PMM server address.
`PMM_AGENT_LISTEN_PORT`	PMM agent listen port.
`PMM_AGENT_PORTS_MIN`	Min PMM agent port.
`PMM_AGENT_PORTS_MAX`	Max PMM agent port.
`PMM_AGENT_CONFIG_FILE`	PMM agent config path.
`PMM_AGENT_SERVER_INSECURE_TLS`	Allows insecure TLS.
`PMM_AGENT_LISTEN_ADDRESS`	PMM agent listen address.
`PMM_AGENT_SETUP_NODE_NAME`	Node name used by PMM.
`PMM_AGENT_SETUP`	Enables auto-setup.
`PMM_AGENT_SETUP_FORCE`	Forces auto-setup.

`PMM_AGENT_SETUP_NODE_TYPE`	Node type (`container`).
`PMM_AGENT_SETUP_METRICS_MODE`	Metrics mode (`push`).
`PMM_ADMIN_CUSTOM_PARAMS`	Extra `pmm-admin` parameters.
`PMM_AGENT_SERVER_USERNAME`	PMM server user (optional).
`PMM_AGENT_SERVER_PASSWORD`	PMM server password (optional).
`PMM_AGENT_SIDECAR`	Sidecar flag (v1.10+).
`PMM_AGENT_SIDECAR_SLEEP`	Sidecar delay (v1.10+).
`PMM_AGENT_PATHS_TEMPDIR`	Temporary dir override (v1.15+).

PMM3

Variable	Purpose
`DB_TYPE`	Database type (`mongodb`).
`DB_USER`	Monitoring user (from Secret).
`DB_PASSWORD`	Monitoring password (from Secret).
`DB_HOST`	MongoDB host (defaults to `localhost`).
`DB_CLUSTER`	Cluster name.
`DB_PORT`	MongoDB port.
`CLUSTER_NAME`	Cluster name (or custom PMM cluster name).
`POD_NAME`	Pod name.
`POD_NAMESPACE`	Pod namespace.
`PMM_AGENT_SERVER_ADDRESS`	PMM server address.
`PMM_AGENT_SERVER_USERNAME`	PMM server username (`service_token`).
`PMM_AGENT_SERVER_PASSWORD`	PMM server token (from Secret).
`PMM_AGENT_LISTEN_PORT`	PMM agent listen port.
`PMM_AGENT_PORTS_MIN`	Min PMM agent port.
`PMM_AGENT_PORTS_MAX`	Max PMM agent port.
`PMM_AGENT_CONFIG_FILE`	PMM agent config path.
`PMM_AGENT_SERVER_INSECURE_TLS`	Allows insecure TLS.
`PMM_AGENT_LISTEN_ADDRESS`	PMM agent listen address.
`PMM_AGENT_SETUP_NODE_NAME`	Node name used by PMM.
`PMM_AGENT_SETUP`	Enables auto-setup.
`PMM_AGENT_SETUP_FORCE`	Forces auto-setup.
`PMM_AGENT_SETUP_NODE_TYPE`	Node type (`container`).

`PMM_AGENT_SETUP_METRICS_MODE`	Metrics mode (`push`).
`PMM_ADMIN_CUSTOM_PARAMS`	Extra `pmm-admin` parameters.
`PMM_AGENT_SIDECAR`	Sidecar flag.
`PMM_AGENT_SIDECAR_SLEEP`	Sidecar delay.
`PMM_AGENT_PATHS_TEMPDIR`	Temp dir for PMM agent.
`PMM_AGENT_PRERUN_SCRIPT`	PMM pre-run script.

Log collector sidecars

Fluent Bit (`logs`)

Variable	Purpose
`LOG_DATA_DIR`	Log directory location.
`POD_NAMESPACE`	Pod namespace.
`POD_NAME`	Pod name.

Logrotate (`logrotate`)

Variable	Purpose
`MONGODB_HOST`	MongoDB host (`localhost`).
`MONGODB_PORT`	MongoDB port.
`MONGODB_USER`	Cluster admin user (from Secret).
`MONGODB_PASSWORD`	Cluster admin password (from Secret).
`LOGROTATE_SCHEDULE`	Cron schedule for log rotation (optional).

7.17 Define custom environment variables

Version added: 1.22.0

Custom environment variables let you inject configuration without rebuilding container images. This is useful when you need to:

- Align container behavior with your platform (cache size settings, time zone, locale).
- Pass non-sensitive runtime flags to custom entrypoints.
- Provide credentials or third-party API tokens from a Secret without baking them into images.

You can configure custom environment variables in these ways:

- Set them in the Custom Resource directly
- Load variables from a ConfigMap
- Load variables from a Secret

Supported components

Custom `env` and `envFrom` are supported for these components:

- `mongod` containers: `spec.replsets[].env` and `spec.replsets[].envFrom`
- `mongos` containers: `spec.sharding.mongos.env` and `spec.sharding.mongos.envFrom`
- Log collector container: `spec.logcollector.env` and `spec.logcollector.envFrom`

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 set a time zone to keep logs aligned across containers.

1. Edit the `deploy/cr.yaml` Custom Resource manifest

```
spec:
  replsets:
  - name: rs0
    size: 3
    env:
      - name: TZ
        value: "UTC"
```

2. Apply the changes:

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

The Operator makes a rolling restart of your Pods.

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.

For example, set a custom runtime flag for `mongod` across all replica set pods.

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. Let's name it `mongod-flags.yaml`:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: psmdb-env-config
data:
  MONGODB_EXTRA_FLAGS: "--setParameter diagnosticDataCollectionEnabled=false"
```

3. Create the ConfigMap:

```
kubectl apply -f mongod-flags.yaml -n $NAMESPACE
```

4. Reference it in the Custom Resource:

```
spec:
  replsets:
  - name: rs0
    size: 3
    envFrom:
      - configMapRef:
          name: psmdb-env-config
```

5. Apply the changes:

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

The Operator makes a rolling restart of your Pods.

Load variables from a Secret

Use this 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, `custom-sidecar.yaml`:

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: psmdb-env-secrets
   type: Opaque
   stringData:
     LOG_EXPORT_TOKEN: "your-base64-encoded-token"
   ```

4. Create the Secret object:

   ```
   kubectl apply -f custom-sidecar.yaml -n $NAMESPACE
   ```

5. Reference the Secret in the Custom Resource:

```
spec:
  logcollector:
    enabled: true
    envFrom:
      - secretRef:
          name: psmdb-env-secrets
```

6. Apply the changes:

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

The Operator makes a rolling restart of your Pods.

10.7 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 PerconaServerMongoDB.

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-mongodb-operator` deployment controls the number of concurrent workers that can reconcile resources in Percona Server for MongoDB 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 MongoDB clusters. When the number of workers is greater, the excessive workers will remain idle.

Set the number of reconciliation workers

1. Check the index of the `MAX_CONCURRENT_RECONCILES` environment variable using the following command:

```
kubectl get deployment percona-server-mongodb-operator -o jsonpath='{.spec.template.spec.containers[0].env[?
(@.name=="MAX_CONCURRENT_RECONCILES")].value}'
```

Sample output

```
1
```

2. To set a new value and verify it's been updated, run the following command:

```
kubectl patch deployment percona-server-mongodb-operator \
--type='strategic' \
-o yaml \
-p='{
    "spec": {
        "template": {
            "spec": {
                "containers": [
                    {
                        "name": "percona-server-mongodb-operator",
                        "env": [
                            {
                                "name": "MAX_CONCURRENT_RECONCILES",
                                "value": "2"
                            }
                        ]
                    }
                ]
            }
        }
    }
}'\
-o jsonpath='{.spec.template.spec.containers[0].env[?(@.name=="MAX_CONCURRENT_RECONCILES")].value}'
```

The command does the following:

- Patches the deployment to update the `MAX_CONCURRENT_RECONCILES` environment variable
- Sets the value to `2`.
- Outputs the result

You can set the value to any number greater than 0.

Sample output

```
2
```

7.19 Logging

7.19.1 Persistent logging

In a distributed Kubernetes environment, it's often difficult to debug issues because logs are tied to the lifecycle of individual Pods and containers. If a Pod fails and restarts, its logs are lost, making it hard to identify the root cause of an issue.

Percona Operator for MongoDB addresses this challenge with **persistent logging**, ensuring logs are stored persistently, independent of the Pods. This approach helps ensure that logs are available for review even after a Pod restarts.

The Operator uses [Fluent Bit](), a lightweight log processor with versatile output plugins and forwarding features, to collect logs. Fluent Bit runs as a `logs` sidecar container alongside each database Pod. It gathers logs from the `mongod` container, adds metadata, and saves them in a single file called `mongod.full.log` in the `/data/db/` directory within the Pod's Persistent Volume Claim (PVC). As a result, logs persist across Pod restarts and remain available for later debugging.

Logs are also streamed to standard output, making them accessible via the `kubectl logs` command for quick troubleshooting:

```
kubectl logs my-cluster-name-rs0-0 -c logs
```

Currently, logs are collected only for the `mongod` instances running in the config server and replica set Pods. All other logs are ephemeral, meaning they will not persist after a Pod restart. Logs are rotated daily, stored for 7 days and are deleted afterwards.

Configure log collector

Cluster-level logging is enabled by default and is controlled with the `logcollector.enabled` key in the `deploy/cr.yaml` Custom Resource manifest.

You can additionally configure Fluent Bit using the `logcollector.configuration` subsection in the `deploy/cr.yaml` Custom Resource manifest. This allows you to define custom filters or output plugins to suit your specific logging and monitoring needs.

Note that when you add a new configuration to the `logcollector.configuration`, this triggers a Smart Update.

7.19.2 Log rotation

`logrotate` is a tool to manage the log file growth. MongoDB logs can grow quickly. Without rotation, logs may fill the filesystem and disrupt the database. `logrotate` ensures predictable log retention and disk usage.

`logrotate` runs in a `logcollector` sidecar container within each database Pod. It reads the log files at the specified path and rotates them according to the set of rules.

By default, `logrotate` in Percona Operator for MongoDB works as follows:

- Rotates the `/data/db/logs/mongod.full.log` daily.
- If the log file exceeds 100 MB, it will be rotated on the next run, regardless of the schedule.
- Keeps up to 7 rotated log files.
- Skips missing or empty log files.
- Leaves rotated logs uncompressed.
- Copies the current file to the rotated file and then truncates this file to zero in place.
- Ensures pre-rotation scripts run only once per rotation with `sharedscripts`.
- Before rotating, performs the following in the `prerotate` script:
- Runs the `db.adminCommand({ logRotate: 1 })` command so that MongoDB closes its current log and starts a new one.
- Deletes any `mongod.log.*` files in `/data/db/logs/` that are older than 7 days.

Configure log rotation

You can customize log rotation if you need different retention, size limits, or want to rotate additional files (for example, audit logs). This gives you predictable disk usage and keeps compliance or troubleshooting logs available for as long as you need.

You can configure log rotation in the following ways:

- Override the default logrotate configuration via the Custom Resource
- Add additional configuration via ConfigMap
- Set a custom log rotation schedule

Regardless of the method you use, after you apply the new configuration, the Operator will restart the database Pods.

Override the default logrotate configuration

Use the `logcollector.logrotate.configuration` section in the Custom Resource to completely override the default `logrotate` settings. For example, you can configure it to rotate logs when their size is between 100 KB (minimum) and 200 MB (maximum).

> **Important**
>
> You must provide the full `logrotate` configuration because the Operator replaces the default configuration with the one you provide. Refer to the [default configuration](#) to see the built-in logrotate rules and use it as a guide for your custom settings.

Here's an example configuration:

```yaml
spec:
  logcollector:
    logrotate:
      configuration: |
        /data/db/logs/mongod.full.log {
           daily
           minsize 100K
           maxsize 200M
           rotate 7
           missingok
           nocompress
           notifempty
           copytruncate
           sharedscripts
           prerotate
               # rotate mongod.log using 'db.adminCommand({ logRotate: 1 })'
               mongosh "mongodb://${MONGODB_USER}:${MONGODB_PASSWORD}@${MONGODB_HOST}:${MONGODB_PORT}/admin" \
                       --eval 'db.adminCommand({ logRotate: 1 })'
               find /data/db/logs/ -type f -name 'mongod.log.*' -mtime +7 -delete
           endscript
        }
```

Apply the configuration:

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

Add extra logrotate configuration

You can pass additional configuration options for the default logrotate configuration via a ConfigMap or a Secret. Use `logcollector.logrotate.extraConfig.name` to load an additional `.conf` file from a ConfigMap or Secret. The file name must end with `.conf`.

> **Important**
>
> The `mongodb.conf` name is reserved for the main configuration and you must not use it in your custom ConfigMap or Secret.

For example, you want to also rotate audit log files. Here's how to provide this additional configuration.

1. Create a ConfigMap configuration file. For example, `auditlog.yaml`.

   ```yaml
   apiVersion: v1
   kind: ConfigMap
   metadata:
     name: logrotate-custom
   data:
     audit.conf: |
       /data/db/audit/*.json {
          minsize 10K
          maxsize 20K
          rotate 10
          missingok
          nocompress
          notifempty
          sharedscripts
          copytruncate
         }
   ```

2. Create the ConfigMap:

   ```
   kubectl apply -f auditlog.yaml
   ```

3. Reference the ConfigMap in your Custom Resource and specify the path to audit logs:

```
spec:
  # ...
  replsets:
  - name: rs0
    size: 1
    configuration: |
      auditLog:
        destination: file
        format: JSON
        path: /data/db/audit/audit.json
  logcollector:
    logrotate:
      extraConfig:
        name: logrotate-custom
```

4. Apply the configuration:

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

Set a custom schedule

Starting with version 1.22.0, the default logrotate schedule changed from running once per hour to running once per day.

Use the `logcollector.logrotate.schedule` option in the Custom Resource to set a custom rotation schedule as a cron expression. For example, set the new rotation time to 2:15 a.m every Sunday:

```
spec:
  logcollector:
    logrotate:
      schedule: "0 15 2 * * 0"
```

Apply the configuration:

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

8 Management

8.1 Backup and restore

8.1.1 About backups

You can back up your data in two ways:

- *On-demand*. You can do them manually at any moment.
- *Scheduled backups*. Configure backups and their schedule in the [deploy/cr.yaml](). The Operator makes them automatically according to the specified schedule.

To make backups and restores, the Operator uses the [Percona Backup for MongoDB (PBM)]() tool. The Operator runs PBM as [a sidecar container]() to the database Pods. It configures PBM in the following cases:

- when it creates a new cluster if you defined the [backup storage configuration]() for it.
- when you configure the backup storage for a backup
- when you [start a restore on a new cluster]() and defined the backup storage configuration within the `backupSource` subsection of the Restore resource.

How backup and restore work

`pbm-agent` processes running as sidecar containers in every database Pod watch the PBM control collections and react on changes. Read more about the control collections in the [PBM documentation]().

When a new document is added to PBM control collections, one of the `pbm-agent` processes starts a backup or a restore operation. This `pbm-agent` is randomly chosen among the secondary nodes. For more details on how the election process works, refer to the [PBM documentation]().

Backup flow

When you create a backup object, the Operator adds a new document to the PBM control collections. All `pbm-agent` processes continuously monitor these collections for changes. When a new backup request appears, one of the `pbm-agent` processes on secondary MongoDB nodes starts making a backup.

The `pbm-agent` reads the backup type from the document, copies data based on the backup type and uploads it to the configured storage:

- Logical backup: PBM reads database data and uploads it.
- Physical backup: PBM copies data files from `dbPath` and uploads them.

Restore flow

The restore flow depends on the backup type:

From logical backup

When you create the Restore object, the following occurs:

1. The Operator writes a new document into the PBM control collections.
2. The Operator shuts down `mongos` Pods (for sharded clusters) so that clients cannot access the database when the restore is in progress.
3. A `pbm-agent` restores data from the backup into the corresponding collections.
4. For a selective restore, PBM restores only the specified namespaces.
5. For point-in-time recovery, PBM replays oplog chunks on top of the restored data to bring the database to the time defined in the `restore_to_time` field.

From physical backup

To restore the database from a physical backup, a `pbm-agent` requires the access to the `mongod` binaries. Therefore, when you create the Restore object, the following occurs:

1. The Operator prepares the cluster:
2. It terminates `mongos` Pods (for sharded clusters) to prevent the clients from accessing the database when the restore is in progress.
3. It terminates arbiter nodes.
4. It updates the StatefulSet by moving PBM binaries into the `mongod` container and removes the PBM sidecar. This triggers a rolling restart of the database Pods.
5. It triggers the restore using the PBM CLI.
6. The restore is made by the `pbm-agent` process running inside the `mongod` container.

7. The `pbm-agent` wipes the `dbPath`, downloads the backup files from the storage, and copies the files into the data directory. It also applies oplog chunks from the backup snapshot to maintain data consistency.
8. During the restore, PBM restarts the `mongod` process several times as it switches between restore phases and starts `mongod` with the newly restored data files.
9. After a successful restore, the Operator recreates the StatefulSet with the regular configuration so PBM runs as a sidecar again. All database Pods are terminated and recreated. The Operator also restarts arbiter nodes and `mongos` Pods.

Point-in-time recovery from a physical backup

1. The Operator follows the same preparation steps as for a physical restore.
2. It ensures that Pod 0 is the primary. If it is not, the Operator makes it primary and waits for it to report the `PRIMARY` status.
3. PBM restores the backup and then applies oplog chunks to reach the target time.
4. After the successful restore, the Operator recreates the StatefulSet with its regular configuration.

For hands-on steps, see Restore the cluster from a previously saved backup.

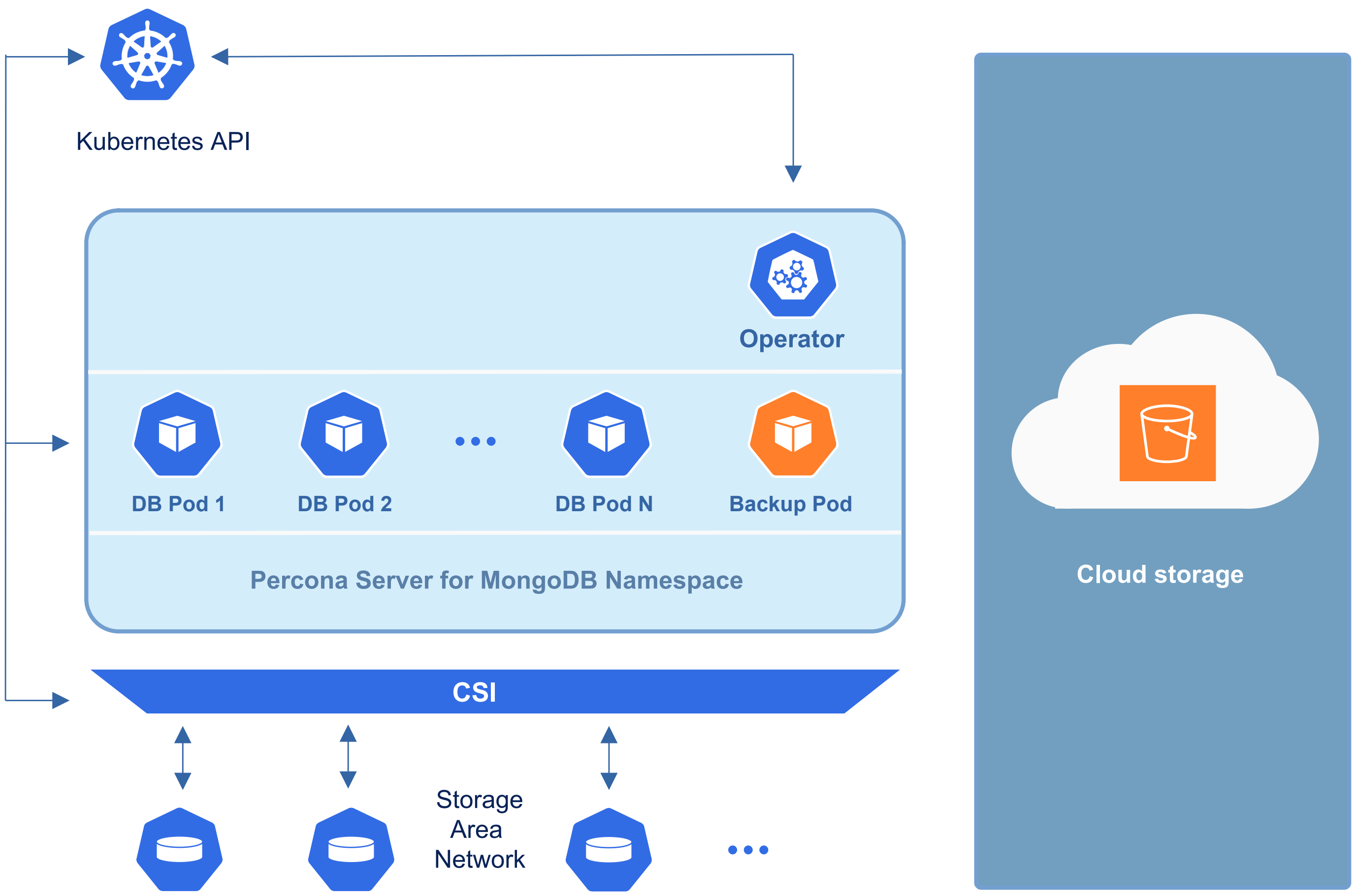
Backup retention

Each backup object has the `delete-backup` finalizer, so deleting the object also removes the backup files from storage. You can control how many backups to keep with the backup.tasks.retention retention setting. For details, see Configure retention policy.

Backup storage

You can store Percona Server for MongoDB backups outside the Kubernetes cluster using the following remote backup storages:

- Amazon S3 storage
- Google Cloud storage
- MinIO and S3-compatible storages
- Microsoft Azure Blob storage
- Remote file server



Multiple backup storages

Starting with version 1.20.0, the Operator natively supports multiple backup storages, inheriting this feature from Percona Backup for MongoDB (PBM). This means you don't have to wait till the Operator reconfigures a cluster after you select a different storage for a backup or a restore. And you can make a point-in-time recovery from any backup stored on any storage - PBM and the Operator maintain the data consistency for you.

Find more information in the Multiple storages for backups chapter.

Backup types

Backup type	Version added	Status	Description	Important considerations
Full logical	Initial	GA	Queries Percona Server for MongoDB for database data and writes this data to the remote storage	- Uses less storage but is slower than physical backups - Supports selective restore since 1.18.0 - Supports point-in-time recovery - Incompatible for restores with backups made with Operator versions before 1.9.0. Make a new backup after the upgrade to the Operator 1.9.0.
Full physical	1.14.0	GA (1.16.0)	Copies physical files from MongoDB `dbPath` data directory to remote storage	- Faster backup/restore than logical - Better for large datasets - Supports point-in-time recovery since 1.15.0
Physical incremental	1.20.0	Tech preview	Copies only data changed after the previous backup	- Speeds up backup/restore - Reduces network load and storage consumption - Requires a base incremental backup to start the incremental chain - Base backup and increments must bet taken from the same node

	- New base backup is needed if a node is down or if the cluster was restored from a backup

8.1.2 Configure storage for backups

8.1.2.1 Configure storage for backups

You can configure storage for backups in the `backup.storages` subsection of the Custom Resource, using the deploy/cr.yaml configuration file.

The Operator provides several storage types for different storages. To help you choose the right storage type, consider these points:

- **S3** - Use this storage type for native AWS S3 and for S3-compatible storage services that support Signature Version 4 (SigV4) used in the AWS SDK 2.
- **minio** - Use this storage type for MinIO and other S3-compatible services that don't support SigV4 or require endpoint configuration that works better with the `minio` storage type.
- **gcp** - Use this storage type for Google Cloud Storage.
- **azure** - Use this storage type for Microsoft Azure Blob storage.
- **filesystem** - Use this storage type for uploading backups to a remote file server.

For each backup storage, create a Kubernetes Secret object with credentials and reference it in the Custom Resource.

Storage setup guides

Use the page for your storage type:

- Amazon S3 storage
- MinIO and S3-compatible storages
- Google Cloud storage
- Microsoft Azure Blob storage
- Remote file server

8.1.2.2 Amazon S3 storage

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

- `metadata.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:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-cluster-name-backup-s3
type: Opaque
data:
  AWS_ACCESS_KEY_ID: UkVQTEFDRS1XSVRILUFXUy1BQ0NFU1MtS0VZ
  AWS_SECRET_ACCESS_KEY: UkVQTEFDRS1XSVRILUFXUy1TRUNSRVQtS0VZ
```

1. Create the Kubernetes Secret object with this file:

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

2. Configure the storage in the Custom Resource. Modify the `backup.storages` subsection of the `deploy/cr.yaml` file. Give your storage a name (the default name is `s3-us-west`) and define the following information:

   - `type` - make sure the type is `s3`
   - `bucket` - where the data will be stored
   - `region` - location of the bucket
   - `credentialsSecret` - the name of the Secret you created previously

   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.

   Here's the example:

   ```
   ...
   backup:
     ...
     storages:
       s3-us-west:
         type: s3
         s3:
           bucket: S3-BACKUP-BUCKET-NAME-HERE
           region: us-west-2
           credentialsSecret: my-cluster-name-backup-s3
     ...
   ```

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

Plan for large backups

Make sure your storage has enough free space for backups, especially for large databases.

Also, note that S3 has an upload limit of 10,000 parts per file. If your backup is very large, you may hit this limit, which increases the size of each chunk. Large chunk sizes can use a lot of memory on the S3 server. To avoid issues:

a. Check your storage free space before backing up.
b. Be aware of the 10,000 part limit for S3 uploads.
c. Adjust your backup chunk size and memory settings in the Custom Resource if necessary for large backups.

3. Apply the configuration:

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

Automating access to Amazon S3 based on IAM roles

Using AWS EC2 instances for backups makes it possible to automate access to AWS S3 buckets based on Identity Access Management (IAM) roles for Service Accounts with *no need to specify the S3 credentials explicitly*.

You can either use the *IAM instance profile* or configure *IAM roles for Service Accounts*. Both approaches are AWS-specific so you should follow the official Amazon documentation.

Using IAM instance profile

Follow these steps:

1. Create the IAM instance profile and the permission policy within where you specify the access level that grants the access to S3 buckets.
2. Attach the IAM profile to an EC2 instance.
3. Configure an S3 storage bucket in the Custom Resource and verify the connection from the EC2 instance to it.
4. *Do not provide* `s3.credentialsSecret` for the storage in `deploy/cr.yaml`.

Using IAM role for service account

IRSA is the native way for the cluster running on Amazon Elastic Kubernetes Service (AWS EKS) to access the AWS API using permissions configured in AWS IAM roles.

Assuming that you have deployed the Operator and the database cluster on EKS, and your EKS cluster has OpenID Connect issuer URL (OIDC) enabled, the high-level steps to configure it are the following:

1. Create an IAM role for your OIDC, and attach to the created role the policy that defines the access to an S3 bucket. See official Amazon documentation for details.
2. Find the service accounts used for the Operator and the database cluster. Service account for the Operator is `percona-server-mongodb-operator` (it is set by the `serviceAccountName` key in the `deploy/operator.yaml` or `deploy/bundle.yaml` manifest). The cluster's default account is `default` (it can be set with the `serviceAccountName` Custom Resource option in the `replsets`, `sharding.configsvrReplSet`, and `sharding.mongos` subsections of the `deploy/cr.yaml` manifest).
3. Annotate both service accounts with the needed IAM roles. The commands should look as follows:

```
kubectl -n <cluster namespace> annotate serviceaccount default eks.amazonaws.com/role-arn:
arn:aws:iam::111122223333:role/my-role --overwrite
kubectl -n <operator namespace> annotate serviceaccount percona-server-mongodb-operator eks.amazonaws.com/role-arn:
arn:aws:iam::111122223333:role/my-role --overwrite
```

   Don't forget to substitute the `<operator namespace>` and `<cluster namespace>` placeholders with the real namespaces, and use your IAM role instead of the `eks.amazonaws.com/role-arn: arn:aws:iam::111122223333:role/my-role` example.

4. Configure an S3 storage bucket in the Custom Resource and verify the connection from the EC2 instance to it. *Do not provide* `s3.credentialsSecret` for the storage in `deploy/cr.yaml`.

Note

If IRSA-related credentials are defined, they have the priority over any IAM instance profile. S3 credentials in a secret, if present, override any IRSA/IAM instance profile related credentials and are used for authentication instead.

8.1.2.3 MinIO and S3-compatible storages

Use the `minio` storage type for MinIO and other S3-compatible storages. It helps with connectivity and compatibility issues when your S3 implementation doesn't support SigV4 or requires endpoint configuration that works better with the `minio` storage type.

To use the `minio` storage type, create a Secret object with your access credentials. You can use the [`deploy/backup-s3.yaml`](deploy/backup-s3.yaml) file as the example.

You must specify the following information:

- `metadata.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 your 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:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-cluster-name-backup-minio
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-s3.yaml -n <namespace>
   ```

2. Configure the storage in the Custom Resource. Modify the `backup.storages` subsection of the `deploy/cr.yaml` file. Give the name to the storage (by default, `minio`). You will later use it to refer this storage when making backups and restores.

   Specify the following configuration:

   - Set `type` to `minio`.
   - Specify the bucket name for the `minio.bucket` option
   - Specify the location of the bucket for the `minio.region` option
   - Specify the name of the Secret you created previously for the `minio.credentialsSecret` option
   - Specify the `minio.endpointUrl` option if your storage requires it. This points to your storage service and is specific to your cloud provider. For example, the endpoint value for MinIO is `minio.psmdb.svc.cluster.local:9000/minio/`

   > **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.

   Here's the example configuration for MinIO storage:

   ```
   backup:
     storages:
       minio:
         type: minio
         minio:
           bucket: MINIO-BACKUP-BUCKET-NAME-HERE
           credentialsSecret: my-cluster-name-backup-minio
           endpointUrl: minio.psmdb.svc.cluster.local:9000/minio/
           region: us-east-1
   ```

3. Apply the configuration:

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

Configure TLS verification with custom certificates for S3 storage

You can use your organization's custom TLS / SSL certificates and instruct the Operator to securely verify TLS communication with your custom S3 storage.

To configure TLS verification with custom certificates, do the following:

1. Create the Secret object that contains the CA bundle needed to verify the S3 endpoint's TLS certificate.
2. Modify the S3 storage configuration in the Custom Resource and specify the following information:
   - `storages.<NAME>.minio.caBundle.name` is the name of the Secret object you created previously
   - `storages.<NAME>.minio.caBundle.key` is the name of the file in the Secret containing the CA bundle.

   Here's the example configuration:

   ```
   ...
   backup:
     ...
     storages:
       minio:
         type: minio
         minio:
           bucket: MINIO-BACKUP-BUCKET-NAME-HERE
           region: us-east-1
           credentialsSecret: my-cluster-name-backup-minio
           caBundle:
             name: minio-ca-bundle
             key: ca.crt
     ...
   ```

   After you apply the configuration, the Operator passes your custom certificate configuration to `pbm-agents`. `pbm-agents` then use it to securely verify TLS communication with S3 storage during backups and restores.

You may use several S3 storages for backups and may have TLS / SSL certificates for secure communication with each storage. In this case, the Operator merges the certificates into a single `ca-bundle.crt` file and passes it to PBM. When connecting to a specific S3 storage, PBM finds the corresponding certificate and uses it to securely verify TLS communication with this storage.

8.1.2.4 Google Cloud storage

To use [Google Cloud Storage (GCS)] as an object store for backups, you need the following information:

- a GCS bucket name. Refer to the GCS bucket naming guidelines for bucket name requirements
- authentication keys for your service account in JSON format.

> **Note**
>
> You can still use the S3-compatible implementation of GCS with HMAC. Refer to the Amazon S3 storage setup section for steps.
>
> However, we don't recommend their usage due to a known issue in PBM and encourage you to switch to using service accounts keys after the upgrade to the Operator version 1.21.0.

Configuration steps

1. Create a service account, if you don't have it already.

2. Add JSON service keys for the service account. As the result a service account key file in JSON format with the private key and related information is automatically downloaded on your machine.

3. Encode your keys in base64 format. You need to encode the service account email and the private key. You can get these values from the service account key file you downloaded when you created the service account keys.

   The following command shows how to encode a private key. Replace the placeholder with your private key and service account email:

```
echo -n "-----BEGIN PRIVATE KEY-----\nPRIVATE_KEY\n-----END PRIVATE KEY-----\n" | base64
```

4. Create the Kubernetes Secret configuration file and specify the encoded GCS credentials within:

gcp-cs-secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: gcp-cs-secret-key
type: Opaque
data:
  GCS_CLIENT_EMAIL: base_64_encoded_email
  GCS_PRIVATE_KEY: base_64_encoded_key
```

5. Create the Kubernetes Secrets object. Replace the `<namespace>` placeholder with your value:

```
kubectl apply -f gcp-cs-secret.yaml -n <namespace>
```

6. Configure the GCS storage in the `deploy/cr.yaml` Custom Resource. Specify the following information:

   - Set `storages.NAME.type` to `gcs` (substitute the `NAME` part with some arbitrary name you will later use to refer this storage when making backups and restores).
   - Specify the bucket name for the `storages.NAME.gcs.bucket` option
   - Specify the Secrets object name you created for the `storages.NAME.gcs.credentialsSecret` option

```
backup:
  storages:
    gcp-cs:
      type: gcs
      gcs:
        bucket: GCS-BACKUP-BUCKET-NAME-HERE
        credentialsSecret: gcp-cs-secret
```

7. Apply the configuration:

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

8.1.2.5 Microsoft Azure Blob storage

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

- `metadata.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 keys to access Azure Blob 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:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-cluster-azure-secret
type: Opaque
data:
  AZURE_STORAGE_ACCOUNT_NAME: UkVQTEFDRS1XSVRILUFXUy1BQ0NFU1MtS0VZ
  AZURE_STORAGE_ACCOUNT_KEY: UkVQTEFDRS1XSVRILUFXUy1TRUNSRVQtS0VZ
```

1. Create the Kubernetes Secret object with this file:

   ```
   kubectl apply -f deploy/backup-azure.yaml
   ```

2. Configure the storage in the Custom Resource. Modify the `backup.storages` subsection of the Custom Resource.

   - `storages.NAME.type` - set to `azure` (substitute the `NAME` part with a name you will use to refer to this storage)
   - `storages.NAME.azure.credentialsSecret` - specify the name of your Secret (`my-cluster-azure-secret` in the example)
   - `storages.NAME.azure.container` - specify the name of the Azure container
   - `storages.NAME.azure.prefix` is the path (sub-folder) inside the container. If prefix is not set, backups are stored in the root directory.

   These and other options within the `storages.NAME.azure` subsection are further described in the Operator Custom Resource options.

   Here is an example:

   ```
   ...
   backup:
     ...
     storages:
       azure-blob:
         type: azure
         azure:
           container: <your-container-name>
           prefix: psmdb
           credentialsSecret: my-cluster-azure-secret
         ...
   ```

3. Apply the configuration:

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

8.1.2.6 Remote file server

You can use the `filesystem` backup storage type to mount a *remote file server* to a local directory as a *sidecar volume*, and make Percona Backup for MongoDB use this directory as a storage for backups.

The approach is based on using common [Network File System (NFS) protocol](). Particularly, this storage type is useful in network-restricted environments without S3-compatible storage, or in cases with a non-standard storage service that still supports NFS access.

1. Add the remote storage as a [sidecar volume]() in the `replset` section of the Custom Resource (and also in `configsvrReplSet` in case of a sharded cluster). You will need to specify the server hostname and some directory on it, as in the following example:

   ```
   replsets:
   - name: rs0
     ...
     sidecarVolumes:
     - name: backup-nfs-vol
       nfs:
         server: "nfs-service.storage.svc.cluster.local"
         path: "/psmdb-my-cluster-name-rs0"
     ...
   ```

   The `backup-nfs-vol` name specified above will be used to refer this sidecar volume in the backup section.

2. Now put the mount point (the local directory path to which the remote storage will be mounted) and the name of your sidecar volume into the `backup.volumeMounts` subsection of the Custom Resource:

   ```
   backup:
     ...
     volumeMounts:
     - mountPath: /mnt/nfs/
       name: backup-nfs-vol
     ...
   ```

3. Finally, storage of the `filesystem` type needs to be configured in the `backup.storages` subsection. It needs only the mount point:

   ```
   backup:
     enabled: true
     ...
     storages:
       backup-nfs:
         type: filesystem
         filesystem:
           path: /mnt/nfs/
   ```

8.1.3 Multiple storages for backups

You can define several storage locations for backups in the Operator. However, previously you were limited to only a single storage for point-in-time recovery, because Percona Backup for MongoDB (PBM) couldn't maintain oplog consistency across multiple storages. Also, you had to wait for the Operator to reconfigure the cluster and sync metadata after you make the next backup or a restore to a different storage.

This behavior is improved. The Operator differentiates the storages as the main storage and profiles. The difference between them is that the Operator uses the main storage to save both backups and oplog chunks for point-in-time recovery. Profiles are used only for backups. This is done for data consistency and to enable point-in-time recovery from a backup on any storage.

Define the main storage

When you configure only one storage, the Operator automatically uses it as the main one until you add more. When you add another storage, you must mark which one is the main using the `main: true` flag in the `deploy/cr.yaml` Custom Resource manifest.

```
storages:
  s3-us-west:
    main: true
    type: s3
```

Note that you can have only one main storage. All other storages are added as profiles.

To check the list of profiles, connect to the database Pod and run the `pbm profile list` command. For example, for the cluster `cluster1`, the command looks as follows:

```
kubectl exec cluster1-rs0-0 -c backup-agent -- pbm profile list
```

You can run other [profile management commands](#) in the same way.

Change the main storage

You can change the main storage by reassigning the `main:true` flag for another one. The Operator then:

- Resyncs the metadata for the new main storage
- Deletes the profile for it
- Adds the previous main storage as a profile

Pass storage configuration via restore objects

Usually you define the storage configuration within the `deploy/cr.yaml` Custom Resource manifest. You can also pass it to the Operator within the `backupSource` option of a Restore object. For example, when you restore the failed site after a disaster. The Operator then checks the current configuration and:

- If there is no storage configured there, it uses the one from the Restore object as the main storage. After the restore it reverts the PBM configuration. You must define the main storage in the `deploy/cr.yaml` file to run further backups.
- If the `deploy/cr.yaml` Custom Resource manifest has the storage configured and it differs from the one from the Restore object, the Operator adds the storage from the Restore object as a profile.

Backup metadata resync

The Operator resyncs the metadata in the following cases:

For the main storage:

- When the main storage changes
- When you manually start a resync using the `pbm config --force resync`

For the profile storage:

- When you start a restore from a backup on a profile
- When you manually resync the metadata on a profile using the `pbm profile sync <storage-name>` command

For the main storage and profiles:

* When you added an annotation `percona.com/resync-pbm=true` to the `deploy/cr.yaml` Custom Resource manifest.

Note that resync is a resource consuming task and we don't recommend to run it manually. Read more when you need to run it in [PBM documentation](#)

The improved support for multiple backup storages brings the following benefits:

* Enables you to make a point-in-time recovery from any storage with guaranteed data consistency
* Reduces the load on the cluster for reconfiguration when the storage for a next backup changes

Upgrade considerations

You must specify the main storage during the upgrade. If you use a single storage, it will automatically be marked as main in the Custom Resource manifest. If you use multiple storages, you must define one of them as main.

The following command shows how to set the `s3-us-west` storage as the main one:

```
kubectl patch psmdb my-cluster-name --type=merge --patch '{
    "spec": {
      "crVersion": "1.20.0",
      "image": "percona/percona-server-mongodb:7.0.18-11",
      "backup": {
        "image": "percona/percona-backup-mongodb:2.9.1",
        "storages": {
          "s3-us-west": {
            "main": true
          }
        }
      },
      "pmm": {
        "image": "percona/pmm-client:2.44.1"
      }
    }
  }'
```

For more information about the upgrades, see the [Update documentation](#)

8.1.4 Storing operations logs for point-in-time recovery

Point-in-time recovery enables you to roll back your cluster to a specific date and time. Starting from the Operator version 1.15.0, you can do a point-in-time recovery from both logical and physical backups.

During point-in-time recovery, the Operator first restores a backup and then applies an operations log (oplog) on top of it. The oplog is the changes that occurred to the operations up to the defined moment.

Preconditions for point-in-time recovery

1. To make a point-in-time recovery, the Operator must start saving oplog events. Set the [backup.pitr.enabled](#) key in the `deploy/cr.yaml` configuration file to enable saving oplog:

   ```
   backup:
     ...
     pitr:
       enabled: true
   ```

2. You must have a full backup to use point-in-time recovery. Without a full backup, Percona Backup for MongoDB will not upload operations logs. You must have a full backup for a new cluster and for a cluster that you restored from a backup.

After you enabled point-in-time recovery, it takes 10 minutes for a first oplog chunk to be uploaded. The default time period between uploads is 10 minutes. You can adjust this time by setting the new duration for the `backup.pitr.oplogSpanMin` option.

PBM saves the oplog to the cloud storage.

Point-in-time recovery with multiple storages

Version 1.20.0 and above

The Operator natively supports multiple storages for backups inheriting this functionality from Percona Backup for MongoDB. This allows you to enable point-in-time recovery and make backups on a storage of your choice. PBM saves oplog only to the main storage to ensure data consistency for all backups on all storages. As a result, you can make a point-in-time restore from any backup on any storage.

Version 1.19.1 and earlier

You must have a single storage defined in the spec.backup.storages option to enable point-in-time recovery. This is because PBM writes oplog to the same bucket where the backup snapshot is saved.

If you defined several storages and try to enable point-in-time recovery, PBM won't know where to save oplog and can't therefore guarantee data consistency for the restore. Therefore, point-in-time recovery is not allowed for multiple storages. You will see the error message in the Operator logs.

8.1.5 Make a backup

8.1.5.1 Making scheduled backups

You can automate the backup process with scheduled backups. Define a schedule and the Operator runs backups automatically according to it. This provides reliability and efficiency to your backups strategy and ensures your data is timely and regularly backed up with no gaps.

Considerations

1. The `percona.com/delete-backup` finalizer applies for an incremental base backup but is ignored for increments. This means that when an incremental base backup is deleted, PBM also deletes all increments that derived from it from the backup storage. There is the limitation that the Backup resource for the base incremental backup is deleted but the Backup resources for increments remain in the Operator. This is because the Operator doesn't control their deletion outsourcing this task to PBM. This limitation will be fixed in future releases.
2. Starting with Operator version 1.17.0, the backup label changed from `ancestor` to `percona.com/backup-ancestor`. The Operator automatically deletes backups with the new `percona.com/backup-ancestor` label, but it does not remove older backups that use the `ancestor` label. To free up storage, you need to manually delete backups created with Operator versions before 1.17.0. For instructions, see Delete backups.

To configure scheduled backups, modify the `backups` section of the deploy/cr.yaml Custom Resource manifest. Specify the following configuration:

1. `backup.enabled` - set to `true`,
2. `backup.storages` subsection - define at least one configured storage.
3. `backup.tasks` subsection - specify the following configuration:
    - `name` - specify a backup name. You will need this name when you restore from this backup.
    - `schedule` - specify the desired backup schedule in crontab format.
    - `enabled` - set this key to `true`. This enables making the `<backup name>` backup along with the specified schedule.
    - `storageName` - specify the name of your already configured storage.
    - `retention` - configure the retention policy: how many backups to keep in the storage. This setting is optional. It applies to base incremental backups but is ignored for increments. Read more about it in the Configure retention section.
    - `type` - specify what type of backup to make. If you leave it empty, the Operator makes a **logical** backup by default.

Configure retention

Use the `backup.tasks.retention` subsection to configure the retention policy for backups. Specify the following parameters:

- `backup.tasks.retention.type` - the retention strategy. The default (and currently only supported strategy) is `count`, which keeps the most recent `backup.tasks.retention.count` backups and removes older ones.
- `backup.tasks.retention.count` - how many backups to keep. Older backups are removed from the storage. See Considerations for details on how this applies to incremental backups.
- `backup.tasks.retention.deleteFromStorage` - if to delete backup files from storage as well.

Examples

Logical

This example shows how to set up backups to run every Saturday night and store them in Amazon S3:

```yaml
...
backup:
  enabled: true
  storages:
    s3-us-west:
      type: s3
      s3:
        bucket: S3-BACKUP-BUCKET-NAME-HERE
        region: us-west-2
        credentialsSecret: my-cluster-name-backup-s3
  tasks:
   - name: "sat-night-backup"
     enabled: true
     schedule: "0 0 * * 6"
     retention:
        count: 3
        type: count
        deleteFromStorage: true
     type: logical
     storageName: s3-us-west
  ...
```

Physical

This example shows how to set up backups to run every Saturday night and store them in Amazon S3:

```yaml
...
backup:
  enabled: true
  storages:
    s3-us-west:
      type: s3
      s3:
        bucket: S3-BACKUP-BUCKET-NAME-HERE
        region: us-west-2
        credentialsSecret: my-cluster-name-backup-s3
  tasks:
   - name: "sat-night-backup"
     enabled: true
     schedule: "0 0 * * 6"
     retention:
        count: 3
        type: count
        deleteFromStorage: true
     type: physical
     storageName: s3-us-west
  ...
```

Incremental

To run incremental backups, consider the following:

1. You must use the same storage for the base backup and subsequent incremental ones
2. The `percona.com/delete-backup` finalizer and the [`backup.tasks.retention`]() settings are considered for incremental base backups but are ignored for increments. This means that when a base backup is deleted, PBM deletes all increments that derive from it.

There is the limitation that the Backup resource for the base incremental backup is deleted but the Backup resources for increments remain in the Operator. This is because the Operator doesn't control their deletion outsourcing this task to PBM. This limitation will be fixed in future releases.

This example shows how to set up incremental base backups to run every Sunday at 5 a.m. and subsequent incremental backups every night at 1:00 a.m., and store them in Amazon S3:

```yaml
...
backup:
  enabled: true
  storages:
    s3-us-west:
      type: s3
      s3:
        bucket: S3-BACKUP-BUCKET-NAME-HERE
        region: us-west-2
        credentialsSecret: my-cluster-name-backup-s3
  tasks:
   - name: weekly-s3-us-west-incremental
     enabled: true
     schedule: "0 1 * * *"
     type: incremental
     storageName: s3-us-west
     compressionType: gzip
     compressionLevel: 6
   - name: weekly-s3-us-west-incremental-base
     enabled: true
     schedule: "0 5 * * 0"
     retention:
        count: 3
        type: count
        deleteFromStorage: true
     type: incremental-base
     storageName: s3-us-west
     compressionType: gzip
     compressionLevel: 6
  ...
```

Find the Secrets name object on the source cluster in the `spec.secrets` key in the `deploy/cr.yaml. Use this name to recreate the Secrets on the target cluster.

Find more details about secrets in System Users.

8.1.5.2 Making on-demand backups

An on-demand backup is a backup that you start manually at any time. You create a Backup resource and the Operator uses it to make a backup. A backup can be any of the supported backup types.

If you want to run backups automatically, according to the schedule, see Make scheduled backups tutorial.

Here's what you need to do to run on-demand backups:

Modify the Custom Resource manifest

1. Edit the `deploy/cr.yaml` configuration file and specify the following configuration:

    - Set the `backup.enabled` key to `true`,
    - Check that you have defined at least one configured storage in the `backup.storages` subsection.

2. Apply the changes. Don't forget to replace the `<namespace>` placeholder with your namespace:

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

Create a Backup resource

To create a Backup resource, you need a special custom resource manifest. The deploy/backup/backup.yaml is the example manifest that you can use.

1. Specify the following configuration:

    - `metadata.name` is the name of the backup. You will need this name when you restore from this backup. The default name is `backup1`.
    - `spec.clusterName` is the name of your cluster (prior to the Operator version 1.12.0 this key was named `spec.psmdbCluster`). Run `kubectl get psmdb -n <namespace>` to find out the cluster name.
    - `spec.storageName` is the name of your already configured storage.
    - `spec.type` is the backup type. If you leave it empty, the Operator makes a logical backup by default.

    Examples

Logical

```
apiVersion: psmdb.percona.com/v1
kind: PerconaServerMongoDBBackup
metadata:
  finalizers:
  - percona.com/delete-backup
  name: backup1
spec:
  clusterName: my-cluster-name
  storageName: s3-us-west
  type: logical
```

Physical

```
apiVersion: psmdb.percona.com/v1
kind: PerconaServerMongoDBBackup
metadata:
  finalizers:
  - percona.com/delete-backup
  name: backup1
spec:
  clusterName: my-cluster-name
  storageName: s3-us-west
  type: physical
```

Incremental

To make incremental backups, consider the following:

1. Make the incremental base backup first. The Operator needs the base to start the chain of increments and save only changes from previous backup.
2. Use the same storage for base backup and increments.
3. The `percona.com/delete-backup` finalizer is considered for incremental base backup but is ignored for increments. This means that when a base backup is deleted, PBM deletes all increments that derive from it.

There is the limitation that the Backup resource for the base incremental backup is deleted but the Backup resources for increments remain in the Operator. This is because the Operator doesn't control their deletion outsourcing this task to PBM. This limitation will be fixed in future releases.

Here's the configuration example for the base incremental backup

```
apiVersion: psmdb.percona.com/v1
kind: PerconaServerMongoDBBackup
metadata:
  finalizers:
  - percona.com/delete-backup
  name: backup1
spec:
  clusterName: my-cluster-name
  storageName: s3-us-west
  type: incremental-base
```

This configuration example is for subsequent incremental backups:

```
apiVersion: psmdb.percona.com/v1
kind: PerconaServerMongoDBBackup
metadata:
  name: backup1
spec:
  clusterName: my-cluster-name
  storageName: s3-us-west
  type: incremental
```

2. Apply the `backup.yaml` manifest to start a backup:

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

3 You can track the backup process with the `PerconaServerMongoDBBackup` Custom Resource as follows:

```
kubectl get psmdb-backup
```

Expected output

```
NAME      CLUSTER           STORAGE      DESTINATION                                TYPE      SIZE      STATUS   COMPLETED   AGE
backup1   my-cluster-name   s3-us-west   s3://my-bucket/2025-09-23T10:34:59Z   logical   105.44MB   ready    43m         43m
```

It should show the status as `READY` when the backup process is over.

Troubleshooting

If you have any issues with a backup, here's how you can troubleshoot it:

1. View information about a backup:

```
kubectl describe psmdb-backup backup1
```

Expected output

```
Name:         backup1
Namespace:    my-namespace
Labels:       <none>
Annotations:  <none>
API Version:  psmdb.percona.com/v1
Kind:         PerconaServerMongoDBBackup
Metadata:
  Creation Timestamp:  2025-04-22T11:32:21Z
  Finalizers:
    percona.com/delete-backup
  Generation:        1
  Resource Version:  136319
  UID:               46670473-4fd0-465d-b944-1be3717485a0
Spec:
  Cluster Name:  my-cluster-name
  Storage Name:  gcp-cs
  Type:          incremental-base
Status:
  Completed:        2025-04-22T11:32:42Z
  Destination:      s3://my-bucket/demand-backup-incremental/2025-04-22T11:32:26Z
  Last Transition:  2025-04-22T11:32:42Z
  Pbm Name:         2025-04-22T11:32:26Z
  Pbm Pod:          my-cluster-name-rs0-2.my-cluster-name-rs0.demand-backup-incremental-10277.svc.cluster.local:27017
  Pbm Pods:
    rs0:  my-cluster-name-rs0-2.my-cluster-name-rs0.demand-backup-incremental-10277.svc.cluster.local:27017
  Replset Names:
    rs0
  s3:
    Bucket:              my-bucket
    Credentials Secret:  gcp-cs-secret
    Endpoint URL:        https://storage.googleapis.com
    Prefix:              demand-backup-incremental
    Region:              us-east-1
    Server Side Encryption:
  Start:         2025-04-22T11:32:26Z
  State:         ready
  Storage Name:  gcp-cs
  Type:          incremental-base
```

2. Check logs from the backup-agent container of the appropriate Pod as follows. Find the Pod name in the `pbm Pod` field in the output from the previous step. Or use the following command to get the Pod name:

```
kubectl get psmdb-backup -o yaml | grep pbmPod
```

Now connect to the `backup-agent` of this Pod:

```
kubectl logs pod/my-cluster-name-rs0 -c backup-agent
```

3. Access the same container via ssh and carry on Percona Backup for MongoDB diagnostics.

Restore to a new Kubernetes environment

To restore from a backup to a new Kubernetes-based environment, you must create a Secrets object there with the same user passwords as in the original cluster.

Find the Secrets name object on the source cluster in the `spec.secrets` key in the `deploy/cr.yaml. Use this name to recreate the Secrets on the target cluster.

Find more details about secrets in System Users.

8.1.6 Enable server-side encryption for backups

Encrypting database backups is done separately for physical and logical backups. Physical backups are encrypted if data-at-rest encryption is turned on. Logical backups need to be encrypted on the cloud.

There is a possibility to enable server-side encryption for backups stored on S3. Starting from the version 1.15.0, the Operator supports Server Side Encryption either with AWS Key Management Service (KMS), or just encrypt/decrypt backups with AES-256 encryption algorithm with any S3-compatible storage.

To enable server-side encryption for backups, use backup.storages.<storage-name>.s3.serverSideEncryption section in the `deploy/cr.yaml` configuration file.

Encryption with keys stored in AWS KMS

To use the server-side AWS KMS encryption, specify the ID of your customer-managed key and other needed options as follows:

with kmsKeyID in Custom Resource

Set the following Custom Resource options in the `deploy/cr.yaml` configuration file:

```
backup:
  ...
  storages:
    my-s3:
      type: s3
      s3:
        bucket: my-backup-bucket
        serverSideEncryption:
          kmsKeyID: <kms_key_ID>
          sseAlgorithm: aws:kms
```

Here `<kms_key_ID>` should be substituted with the [ID of your customer-managed key]() stored in the AWS KMS. It should look similar to the following example value: `128887dd-d583-43f2-b3f9-d12036d32b12`.

with kmsKeyID in Secret object

You can avoid storing your `kmsKeyID` in Custom Resource, and put it into a dedicated Secrets object. Define your secret in YAML as follows:

deploy/sse-secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: my-cluster-name-sse
type: Opaque
stringData:
  KMS_KEY_ID: <kms_key_ID>
```

Here `<kms_key_ID>` should be substituted with the [ID of your customer-managed key]() stored in the AWS KMS. It should look similar to the following example value: `128887dd-d583-43f2-b3f9-d12036d32b12`.

When the YAML file is ready, apply it to create the Secret:

```
kubectl create -f deploy/sse-secret.yaml
```

After creating the Secret, set the following Custom Resource options in the `deploy/cr.yaml` configuration file:

```
secrets:
  ...
  sse: my-cluster-name-sse
...
backup:
  ...
  storages:
    my-s3:
      type: s3
      s3:
        bucket: my-backup-bucket
        serverSideEncryption:
          sseAlgorithm: aws:kms
```

Encryption with locally-stored keys on any S3-compatible storage

The Operator also supports server-side encryption with customer-provided keys that are stored on the client side. During the backup/restore process, encryption key will be provided by the Operator as part of the requests to the S3 storage, and the S3 storage will use them to encrypt/decrypt the data with the AES-256 encryption algorithm. This allows to use server-side encryption on S3-compatible storages different from AWS KMS (the feature was tested with the [AWS]() and [MinIO]() storages).

To use the server-side encryption with locally-stored keys, specify your encryption key and other needed options:

with encryption key in Custom Resource

Set the following Custom Resource options in the `deploy/cr.yaml` configuration file:

```
backup:
  ...
  storages:
    my-s3:
      type: s3
      s3:
        bucket: my-backup-bucket
        serverSideEncryption:
          sseCustomerAlgorithm: AES256
          sseCustomerKey: <your_encryption_key_in_base64>
    ...
```

Here `<your_encryption_key_in_base64>` should be substituted with the actual encryption key encoded in base64.

with encryption key in Secret object

You can avoid storing your encryption key in Custom Resource, and put it into a dedicated Secrets object. Define your secret in YAML as follows:

deploy/sse-secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: my-cluster-name-sse
type: Opaque
stringData:
  SSE_CUSTOMER_KEY: <your_encryption_key_in_base64>
```

Here `<your_encryption_key_in_base64>` should be substituted with the actual encryption key encoded in base64.

When the YAML file is ready, apply it to create the Secret:

```
kubectl create -f deploy/sse-secret.yaml
```

After creating the Secret, set the following Custom Resource options in the `deploy/cr.yaml` configuration file:

```
secrets:
  ...
  sse: my-cluster-name-sse
...
backup:
  ...
  storages:
    my-s3:
      type: s3
      s3:
        bucket: my-backup-bucket
        serverSideEncryption:
          sseCustomerAlgorithm: AES256
    ...
```

> **Note**
>
> You can use the following command to get a base64-encoded string from a plain text one:
>
> **in Linux**
>
> ```
> echo -n 'plain-text-string' | base64 --wrap=0
> ```
>
> **in macOS**
>
> ```
> echo -n 'plain-text-string' | base64
> ```

8.1.7 Restore from a backup

8.1.7.1 Restore the cluster from a previously saved backup

You can restore from a backup as follows:

- On the same cluster where you made a backup
- On a new cluster deployed in a different Kubernetes-based environment.
- On a new cluster with different replica set names

This document focuses on the restore to the same cluster.

Restore scenarios

You can make the following restores:

- Restore to a specific point in time. A precondition for this restore is to enable saving oplog operations
- Restore from a backup
- Selective restore from a full logical backup

For either type of a restore you need to create a Restore object using the `deploy/backup/restore.yaml` manifest.

You can specify the backup to restore from in two ways: using the `backupName` or the `backupSource` keys. You must use only one of these options in your restore configuration. Specifying them both together is not allowed.

- Use the **`backupName`** option when backup objects exist in the cluster, such as for restoring to the same cluster where the backup was created. When you specify the `backupName`, PBM automatically determines the backup type and performs the corresponding restore procedure.
- Use the **`backupSource`** option when there are no backup objects in the cluster, such as when restoring to a new cluster. You can also use the `backupSource` for restores to the same cluster, instead of the `backupName`. If you specify the `backupSource`, you must manually specify the backup type (`logical`, `physical` or `incremental`) in the configuration.

Considerations

1. Check PBM's considerations to prevent MongoDB clients from accessing the database when the restore is in progress.
2. During the restore, the Operator may delete and recreate Pods. This may cause downtime. The downtime duration depends on the restore type and the database deployment:
    - *Logical restore in an unsharded cluster* results causes downtime for the duration of the data restore. No Pods are deleted or recreated
    - *Logical restore in a sharded cluster* causes downtime for the duration of the data restore and the time needed to refresh sharding metadata on `mongos`. This results in deleting and recreating only `mongos` Pods.
    - *Physical and incremental restore* causes downtime for the entire period required to restore the data and refresh the sharding metadata on `mongos`. The Operator deletes and recreates all Pods - replica set, config server replica set (if present) and mongos Pods.

Before you begin

1. Make sure that the cluster is running.
2. Export your namespace as an environment variable. Replace the `<namespace>` placeholder with your value:

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

3. Get the backup information. List the backups using this command:

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

4. Get cluster information. List available clusters using this command:

    ```
    kubectl get psmdb -n $NAMESPACE
    ```

Restore from a backup

To restore your Percona Server for MongoDB cluster from any backup, define a `PerconaServerMongoDBRestore` custom resource. The `deploy/backup/restore.yaml` manifest is the same for all restore types.

Set 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 psmdb-backup` command. During the restore, PBM automatically determines the backup type and performs the corresponding restore procedure.

Pass this configuration to the Operator:

via the YAML manifest

1. Edit the deploy/backup/restore.yaml file and specify the following keys:

   ```
   apiVersion: psmdb.percona.com/v1
   kind: PerconaServerMongoDBRestore
   metadata:
     name: restore1
   spec:
     clusterName: my-cluster-name
     backupName: backup1
   ```

2. Start the restore with this command:

   ```
   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: psmdb.percona.com/v1
kind: PerconaServerMongoDBRestore
metadata:
  name: restore1
spec:
  clusterName: my-cluster-name
  backupName: backup1
EOF -n $NAMESPACE
```

If a physical restore fails

If a physical restore fails, the Operator does not roll back the changes it made when preparing for the restore. You must either retry the restore or delete the StatefulSet yourself to return the cluster to its normal configuration. Deleting the StatefulSet can revert the cluster to its pre-restore configuration, but data loss or corruption is still possible depending on when the failure occurred.

The Operator cannot guarantee data consistency after a failed restore because it does not know at which stage the failure happened. Data can be lost, corrupted, incomplete, or only partially restored.

You can inspect restore logs by executing into the `mongod` container and checking the PBM logs. PBM keeps only the latest restore logs because it cleans up the data directory during the process.

Make a point-in-time recovery

1. Check a time to restore for a backup. Use the command below to find the latest restorable timestamp:

   ```
   kubectl get psmdb-backup <backup_name> -n $NAMESPACE -o jsonpath='{.status.latestRestorableTime}'
   ```

2. Modify the deploy/backup/restore.yaml manifest and specify the following configuration:

- set the `spec.clusterName` key to the name of your cluster. When restoring to the same cluster where the backup was created, the cluster name will be identical in both the Backup and Restore objects.
- set the `spec.backupName` key to the name of your backup
- configure point-in-time recovery settings in the `pitr` section:
  - `type` - specify one of the following options
    - `date` - roll back to a specific date
    - `latest` - recover to the latest possible transaction
  - `date` - specify the target datetime in the format `YYYY-MM-DD HH:MM:SS` when `type` is set to `date`

  Here is the example configuration of the `restore.yaml` file:

3. Pass this configuration to the Operator.

   via the YAML manifest

   a. Edit the deploy/backup/restore.yaml file.

   ```
   apiVersion: psmdb.percona.com/v1
   kind: PerconaServerMongoDBRestore
   metadata:
     name: restore1
   spec:
     clusterName: my-cluster-name
     backupName: backup1
     pitr:
       type: date
       date: YYYY-MM-DD hh:mm:ss
   ```

   b. Start the restore with this command:

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

   via the command line

   You can skip editing the YAML file and pass its contents to the Operator via the command line. For example:

   ```
   cat <<EOF | kubectl apply -n $NAMESPACE -f-
   apiVersion: psmdb.percona.com/v1
   kind: PerconaServerMongoDBRestore
   metadata:
     name: restore1
   spec:
     clusterName: my-cluster-name
     backupName: backup1
   EOF
   ```

Selective restore

Starting with the version 1.18.0, you can restore a desired subset of data from a **full** logical backup. Selective logical backups are not yet supported.

Selective restores have a number of limitations. Learn more about the current selective restore limitations in Percona Backup for MongoDB documentation.

Selective restores are controlled by the additional `selective` section in the `PerconaServerMongoDBRestore` Custom Resource. There you can specify a specific database or a collection that you wish to restore:

```
spec:
  selective:
    withUsersAndRoles: true
    namespaces:
    - "db1.collection1"
    - "db2.collection2"
```

You can specify several “namespaces” (subsets of data) as a list for the `selective.namespaces` field. You can specify a namespace as follows:

- as a pair of database and collection names to restore just this database and collection. The format is `db1.collection1`
- as a database name with a wildcard to restore everything from the specific database. The format is `database_name.*`
- as a single star “*” to restore all databases and collections

Also, you can use `selective.withUsersAndRoles` set to `true` to restore a custom database with users and roles from a full backup. Read more about this functionality in [PBM documentation](#).

8.1.7.2 Restore from a backup to a new Kubernetes-based environment

You can restore from a backup as follows:

- On the same cluster where you made a backup
- On a new cluster deployed in a different Kubernetes-based environment.
- On a new cluster with different replica set names

This document focuses on the restore on a new cluster deployed in a different Kubernetes environment.

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.

Restore scenarios

This document covers the following restore scenarios:

- Restore from a backup - restore from a full backup without point-in-time
- Point-in-time recovery - restore to a specific time, a specific or a latest transaction or skip a specific transaction during a restore. This ability requires that you configure storing oplog for point-in-time recovery

Preconditions

1. When restoring to a new Kubernetes-based environment, make sure it has a Secrets object with the same user passwords as in the original cluster.
2. To restore from a physical backup, set the corresponding encryption key of the target cluster. Find more details about encryption in Data-at-rest encryption. The name of the required Secrets object can be found out from the `spec.secrets` key in the `deploy/cr.yaml` (`my-cluster-name-secrets` by default).

Before you begin

1. Make sure that the cluster is running.
2. Export your namespace as an environment variable. Replace the `<namespace>` placeholder with your value:

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

3. Get the backup information. List the backups using this command:

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

4. Get cluster information. List available clusters using this command:

   ```
   kubectl get psmdb -n $NAMESPACE
   ```

Restore from a backup

To make a restore, PBM must know where to take the backup from and have access to that storage.

You can define the backup storage in two ways: within the restore object configuration or pre-configure it on the target cluster's `cr.yaml` file.

Approach 1: Define storage configuration in the restore object

If you haven't defined storage in the target cluster's `cr.yaml` file, you can configure it directly in the restore object:

1. Set appropriate keys in the deploy/backup/restore.yaml file:
   - 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:

- the backup type - either `logical` or `physical`
- a `destination` key. Take it from the output of the `kubectl get psmdb-backup` command.
- the necessary storage configuration keys, just like in the `deploy/cr.yaml` file of the source cluster.

```
...
backupSource:
  type: logical
  destination: s3://S3-BUCKET-NAME/BACKUP-NAME
  s3:
    credentialsSecret: my-cluster-name-backup-s3
    region: us-west-2
    endpointUrl: https://URL-OF-THE-S3-COMPATIBLE-STORAGE
```

The `destination` key is composed of three parts in case of S3-compatible storage: the `s3://` prefix, the s3 bucket name, and the actual backup name. For Azure Blob storage, you don't put the prefix, and use your container name as an equivalent of a bucket.

2. Apply the configuration to start the restore:

```
kubectl exec -it my-cluster-name-rs0-2 -c backup-agent -- pbm config --force-resync
```

During the restore process, the Operator:

a. Takes the storage configuration from the Restore object
b. Configures PBM using this configuration
c. Resyncs metadata to update it on the target cluster
d. Performs the restore operation
e. Reverts the PBM configuration back to the one defined in the `cr.yaml` file (if any)

3. As the post-restore step, configure the main storage within the target cluster's `cr.yaml` to be able to make subsequent backups.

Approach 2: The storage is defined on target

You can already define the storage where the backup is stored in the `backup.storages` subsection of your target cluster's `deploy/cr.yaml` file. In this case, reference it by name within the restore configuration.

1. Set appropriate keys in the deploy/backup/restore.yaml file:

- set `spec.clusterName` key to the name of the target cluster to restore the backup on
- specify the storage name in the `storageName` key. The name must match the name in the `backup.storages` subsection of the `deploy/cr.yaml` file.
- configure the `spec.backupSource` subsection with the backup destination

```
...
storageName: s3-us-west
backupSource:
  destination: s3://S3-BUCKET-NAME/BACKUP-NAME
```

2. After configuring the restore object, start the restoration process:

```
kubectl apply -f deploy/backup/restore.yaml
```

Point-in-time recovery

As with the restore from a backup, PBM must know where to take the backup from and have access to the storage. You can define the backup storage in two ways: within the restore object configuration or pre-configure it on the target cluster's `cr.yaml` file.

Approach 1: Define storage configuration in the restore object

You can configure the storage within the restore object configuration:

1. Set appropriate keys in the deploy/backup/restore.yaml file.

- set `spec.clusterName` key to the name of the target cluster to restore the backup on

- put additional restoration parameters to the `pitr` section:

  - `type` key can be equal to one of the following options
    - `date` - roll back to specific date
    - `latest` - recover to the latest possible transaction
  - `date` key is used with `type=date` option and contains value in datetime format

- configure the `spec.backupSource` subsection to point to the cloud storage where the backup is stored. This subsection should include:
  - the [backup type](#) - either `logical` or `physical`
  - a `destination` key. Take it from the output of the `kubectl get psmdb-backup` command.
  - the [necessary storage configuration keys](#), just like in the `deploy/cr.yaml` file of the source cluster.

  ```
  apiVersion: psmdb.percona.com/v1
  kind: PerconaServerMongoDBRestore
  metadata:
    name: restore1
  spec:
    clusterName: my-cluster-name
    pitr:
      type: date
      date: YYYY-MM-DD hh:mm:ss
    backupSource:
      destination: s3://S3-BUCKET-NAME/BACKUP-NAME
      s3:
        credentialsSecret: my-cluster-name-backup-s3
        region: us-west-2
        endpointUrl: https://URL-OF-THE-S3-COMPATIBLE-STORAGE
  ```

2. Run the actual restoration process:

   ```
   kubectl apply -f deploy/backup/restore.yaml
   ```

   During the restore process, the Operator:

   a. Takes the storage configuration from the Restore object
   b. Configures PBM using this configuration
   c. Resyncs metadata to update it on the target cluster
   d. Performs the restore operation
   e. Reverts the PBM configuration back to the one defined in the `cr.yaml` file (if any)

3. As the post-restore step, configure the [main storage](#) within the target cluster's `cr.yaml` to be able to make subsequent backups.

Approach 2: The storage is defined on target

You can [define the storage](#) where the backup is stored in the `backup.storages` subsection of your target cluster's `deploy/cr.yaml` file. In this case, reference it by name within the restore configuration.

1. Set appropriate keys in the [deploy/backup/restore.yaml](#) file.

- set `spec.clusterName` key to the name of the target cluster to restore the backup on
- put additional restoration parameters to the `pitr` section:

  - `type` key can be equal to one of the following options
    - `date` - roll back to specific date
    - `latest` - recover to the latest possible transaction
  - `date` key is used with `type=date` option and contains value in datetime format

- specify the storage name for the `storageName` key. The name must match the name the `backup.storages` subsection of the `deploy/cr.yaml` file.
- configure the `spec.backupSource` subsection with the backup destination

```
...
storageName: s3-us-west
backupSource:
  destination: s3://S3-BUCKET-NAME/BACKUP-NAME
```

2. Though PBM resyncs metadata on the target cluster when you start the restore process, for point-in-time recovery to *the latest possible transaction*, we recommend to run a manual resync before the restore. This ensures PBM has the latest oplog chunks on the target cluster. Connect to one of the database Pods (`my-cluster-name-rs0-2` for example) and run the following command:

```
kubectl exec -it my-cluster-name-rs0-2 -c backup-agent -- pbm config --force-resync
```

3. Start the restore process:

```
kubectl apply -f deploy/backup/restore.yaml
```

Restore from a backup with a prefix in a bucket path

If you defined a prefix (a folder) in a bucket where you store backups, you must specify this prefix in the `spec.backupSource` subsection of the restore configuration.

To illustrate, let's say you defined a prefix `my-prefix` for your AWS s3 bucket `my-example-bucket`. You wish to restore a logical backup `2025-05-19T07:23:46Z`. The pull path to this backup is `"s3://my-example-bucket/my-prefix/2025-05-19T07:23:46Z"`. In this case, your restore configuration looks like this:

Storage defined in a Restore object

```
apiVersion: psmdb.percona.com/v1
kind: PerconaServerMongoDBRestore
metadata:
  name: restore-prefix
spec:
  clusterName: my-cluster-name
  backupSource:
    type: logical
    destination: "s3://my-example-bucket/my-prefix/2025-05-19T07:23:46Z"
    s3:
      credentialsSecret: my-cluster-name-backup-s3
      region: us-east-1
      bucket: backup-testing
      prefix: my-prefix
```

Storage defined on target

Make sure the Custom Resource of the target cluster includes the storage configuration and the defined prefix. Then you reference this storage by name in the restore configuration.

```
apiVersion: psmdb.percona.com/v1
kind: PerconaServerMongoDBRestore
metadata:
  name: restore-prefix
spec:
  clusterName: my-cluster-name
  storageName: us-east-1
  backupSource:
    type: logical
    destination: "s3://my-example-bucket/my-prefix/2025-05-19T07:23:46Z"
```

Apply the configuration to start a restore:

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

8.1.7.3 Restore to a new cluster with different replica set names

!!! admonition "Version added: 1.22.0

You can restore from a backup to a cluster that has different replica set names. The number of shards in the target cluster depends on the restore type:

- For *logical* restores, the target cluster can have the same number of shards as the source where the backup was taken, or more.
- For *physical* and *incremental* restores, the number of shards in the target cluster **must be exactly the same** as in the source cluster.

Before you begin

1. Make sure that the cluster is running.
2. Export your namespace as an environment variable. Replace the `<namespace>` placeholder with your value:

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

3. Get the backup information. List the backups using this command:

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

4. Get cluster information. List available clusters using this command:

   ```
   kubectl get psmdb -n $NAMESPACE
   ```

To restore data to the cluster with different replica set names, configure the name mapping for the `replsetRemapping` subsection in the restore configuration file. Specify each source replica set name as a key, and set the target replica set name as the value

Here's the example configuration:

```
...
metadata:
  name: restore1
spec:
  clusterName: my-cluster-name
  backupName: backup1
  replsetRemapping:
    sourceRs0: targetRs0
    sourceRs1: targetRs1
```

Apply the configuration to start the restore:

```
kubectl apply -f deploy/backup/restore.yaml
```

With this ability you extend the list of environments suitable for the restore lifting the limitation for the target cluster to have the same configuration.

8.1.8 Delete the unneeded backup

You can delete backups in the following way:

- Configure the retention policy and have the Operator delete them according to this policy rules
- Manually

Configure backup retention

Use the `backup.tasks.retention` subsection to configure the retention policy for backups. Specify the following parameters:

- `backup.tasks.retention.type` - the retention strategy. The default (and currently only supported strategy) is `count`, which keeps the most recent `backup.tasks.retention.count` backups and removes older ones.
- `backup.tasks.retention.count` - how many backups to keep. Older backups are removed from the storage. See Considerations for details on how this applies to incremental backups.
- `backup.tasks.retention.deleteFromStorage` - if to delete backup files from storage as well.

Delete manually

To delete a backup manually, you need to specify the backup name. Get the name from the list of available backups returned by the following command:

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

Sample output

```
backup1    some-name    minio        s3://operator-testing/2026-02-03T16:54:32Z                              logical    55.21KB    ready    16m
17m
backup2    some-name    azure-blob   azure://operator-testing/psmdb-scheduled-backup/2026-02-03T16:54:00Z    logical    49.70KB    ready    17m
17m
backup2    some-name    aws-s3       s3://operator-testing/psmdb-scheduled-backup/2026-02-03T16:54:11Z       logical    50.72KB    ready    17m
17m
```

Now, you can delete the desired backup as follows:

```
kubectl delete psmdb-backup/<backup-name> -n <namespace>
```

Delete base backups for point-in-time recovery

You can delete a backup used as a base for point-in-time recovery (PITR) starting from the Operator version 1.15.0. Note that deleting such a backup will delete the stored operations log updates based on this backup.

Delete backups with the legacy `ancestor` label

Backups created before version 1.17.0 have the `ancestor` label. The Operator doesn't automatically delete such backups according to the retention policy. You should manually delete them to free up storage.

To find and remove these legacy backups:

1. List all backups with the `ancestor` label:

   ```
   kubectl get psmdb-backup -l ancestor -n <namespace>
   ```

2. Delete them:

   ```
   kubectl delete psmdb-backup -l ancestor -n <namespace>
   ```

8.2 Scale Percona Server for MongoDB on Kubernetes

One of the great advantages brought by Kubernetes is the ease of an application scaling. Scaling a Deployment up or down ensures new Pods are created and set to available Kubernetes nodes.

Scaling can be vertical and horizontal. Vertical scaling adds more compute or storage resources to MongoDB 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 MongoDB replica set instances, `mongos` and config server replica set instances, 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.

To add more resources to your MongoDB replica set instances, edit the following section in the Custom Resource:

```
spec:
  replsets:
    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 version 1.22.0, the Operator monitors storage usage and scales the storage automatically
- storage resizing with Volume Expansion capability - Starting with version 1.16.0, 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 PVC expansion. To check if your storage supports the expansion capability, run the following command:

```
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 1.22.0, the Operator can automatically resize Persistent Volume Claims (PVCs) for replica sets and config server Pods based on your configured thresholds. The Operator monitors storage usage of all PVCs and when it exceeds the defined threshold, 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

To enable automatic storage resizing, edit the `deploy/cr.yaml` Custom Resource manifest as follows:

1. Make sure each MongoDB container has a storage size set.

   Example for a replica set container:

   ```
   replsets:
   - name: rs0
     volumeSpec:
       persistentVolumeClaim:
         resources:
           requests:
             storage: 3Gi
   ```

2. Configure autoscaling thresholds in the `storageScaling` subsection:

   - `enableVolumeScaling` - set to `true`
   - `autoscaling.enabled` - set to `true`
   - `autoscaling.triggerThresholdPercent` - specify the usage percentage. When the usage exceeds this threshold, this triggers autoscaling
   - `autoscaling.growthStep` - specify how much to increase the storage on
   - `autoscaling.maxSize` - specify the upper limit for storage growth. When this limit is reached, scaling is no longer possible.

   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.

Run the `kubectl get psmdb -o yaml -n <namespace>` to check the current cluster state.

Sample output

```
storageAutoscaling:
  mongod-data-my-cluster-name-rs0-0:
    currentSize: 5123744Ki
    lastResizeTime: "2026-01-23T15:08:59Z"
    resizeCount: 2
```

The `storageAutoscaling` section appears under `.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 Troubleshooting storage .

Storage resizing with Volume Expansion capability

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).

2. Specify new storage size for the `replsets.<NAME>.volumeSpec.persistentVolumeClaim.resources.requests.storage` and/or `configsvrReplSet.volumeSpec.persistentVolumeClaim.resources.requests.storage` options in the Custom Resource.

   This is the example configuration of defining a new storage size in the `deploy/cr.yaml` file:

```
spec:
  ...
  enableVolumeExpansion: true
  ...
  replsets:
    ...
    volumeSpec:
      persistentVolumeClaim:
        resources:
          requests:
            storage: <NEW STORAGE SIZE>
  ...
  configsvrReplSets:
    volumeSpec:
      persistentVolumeClaim:
        resources:
          requests:
            storage: <NEW STORAGE SIZE>
```

3. Apply changes as usual:

```
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 `PerconaServerMongoDB` 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 scaling without Volume Expansion capability

Manual scaling is the way to go if:

- your version of the Operator is older than 1.16.0,
- your volumes have a type that does not support Volume Expansion, or
- you do not rely on automated scaling.

You will need to delete Pods and their persistent volumes one by one to resync the data to the new volumes. **This way you can also shrink the storage.**

Here's how to resize the storage:

1. Update the Custom Resource with the new storage size by editing and applying the `deploy/cr.yaml` file:

```
spec:
  ...
  replsets:
    ...
    volumeSpec:
      persistentVolumeClaim:
        resources:
          requests:
            storage: <NEW STORAGE SIZE>
```

2. Apply the Custom Resource for the changes to come into effect:

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

3. Delete the StatefulSet with the `orphan` option

```
kubectl delete sts <statefulset-name> --cascade=orphan
```

The Pods will not go down and the Operator is going to recreate the StatefulSet:

```
kubectl get sts <statefulset-name>
```

Expected output

```
my-cluster-name-rs0       3/3     39s
```

4. Scale up the cluster (Optional)

Changing the storage size would require us to terminate the Pods, which decreases the computational power of the cluster and might cause performance issues. To improve performance during the operation we are going to change the size of the cluster from 3 to 5 nodes:

```
spec:
  ...
  replsets:
    ...
    size: 5
```

Apply the change:

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

New Pods will already have the new storage size:

```
kubectl get pvc
```

Expected output

```
NAME                              STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
mongod-data-my-cluster-name-cfg-0   Bound    pvc-a2b37f4d-6f11-443c-8670-de82ce9fc335   10Gi       RWO            standard       110m
mongod-data-my-cluster-name-cfg-1   Bound    pvc-ded949e5-0f93-4f57-ab2c-7c5fd9528fa0   10Gi       RWO            standard       109m
mongod-data-my-cluster-name-cfg-2   Bound    pvc-f3a441dd-94b6-4dc0-b96c-58b7851dfaa0   10Gi       RWO            standard       108m
mongod-data-my-cluster-name-rs0-0   Bound    pvc-b183c40b-c165-445a-aacd-9a34b8fff227   19Gi       RWO            standard       49m
mongod-data-my-cluster-name-rs0-1   Bound    pvc-f186426b-cbbe-4c31-860e-97a4dfca3de0   19Gi       RWO            standard       47m
mongod-data-my-cluster-name-rs0-2   Bound    pvc-6beb6ccd-8b3a-4580-b3ef-a2345a2c21d6   19Gi       RWO            standard       45m
```

5. Delete PVCs and Pods with the old storage size one by one. Wait for data to sync before you proceed to the next node.

```
kubectl delete pvc <PVC NAME>
kubectl delete pod <POD NAME>
```

The new PVC is going to be created along with the Pod.

The storage size change takes some time. When it starts, the Operator automatically adds the `pvc-resize-in-progress` annotation to the `PerconaServerMongoDB` 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.

Storage resizing with an external autoscaler

You can 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.

Horizontal scaling

Replica Sets

You can change the size separately for different components of your MongoDB replica set by setting these options in the appropriate subsections:

- replsets.size allows you to set the size of the MongoDB Replica Set,
- replsets.nonvoting.size allows you to set the number of non-voting members,
- replsets.arbiter.size allows you to set the number of Replica Set Arbiter instances,

For example, the following update in `deploy/cr.yaml` sets the size of the MongoDB Replica Set `rs0` to `5` nodes:

```
spec:
  ...
  replsets:
  - name: rs0
    size: 5
    ...
```

Don't forget to apply changes as usual, running the `kubectl apply -f deploy/cr.yaml` command.

> **Note**
>
> The Operator will not allow to scale Percona Server for MongoDB with the `kubectl scale statefulset <StatefulSet name>` command as it puts `size` configuration options out of sync.

Sharding

You can change the size for different components of your MongoDB sharded cluster by setting these options in the appropriate subsections:

- sharding.configsvrReplSet.size allows you to set the number of Config Server instances in a sharded cluster,
- sharding.mongos.size allows you to set the number of mongos instances in a sharded cluster.

Changing the number of shards

You can change the number of shards of an existing cluster by adding or removing members in the spec.replsets subsection.

For example, given the following cluster that has 2 shards:

```
spec:
  ...
  replsets:
  - name: rs0
    size: 3
    ...
  - name: rs1
    size: 3
    ...
```

You can add an extra shard by applying the following configuration:

```yaml
spec:
  ...
  replsets:
  - name: rs0
    size: 3
    ...
  - name: rs1
    size: 3
    ...
  - name: rs2
    size: 3
    ...
```

Similarly, you can reduce the number of shards by removing the `rs1` and `rs2` elements:

```yaml
spec:
  ...
  replsets:
  - name: rs0
    size: 3
    ...
```

> **Note**
>
> The Operator will not allow you to remove existing shards unless they don't have any user-created collections. It is your responsibility to ensure the shard's data is migrated to the remaining shards in the cluster before trying to applying this change.

8.3 Multi-cluster and multi-region deployment

8.3.1 About multi-cluster and multi-region Percona Operator for MongoDB deployments

MongoDB is built for distributed resilience — and Percona Operator for MongoDB unlocks that power across clusters and regions.

This section introduces two powerful deployment models — **multi-cluster** and **multi-region**. It also explains how to configure cross-site replication using the Operator. You'll learn how to structure your clusters, understand the roles of Main and Replica sites, and set up secure, synchronized Percona Server for MongoDB clusters across environments with the Operator.

Deployment models: Multi-cluster vs Multi-region

At a glance, both models involve running Percona Server for MongoDB nodes across multiple environments. But their goals, scope, and setup differ.

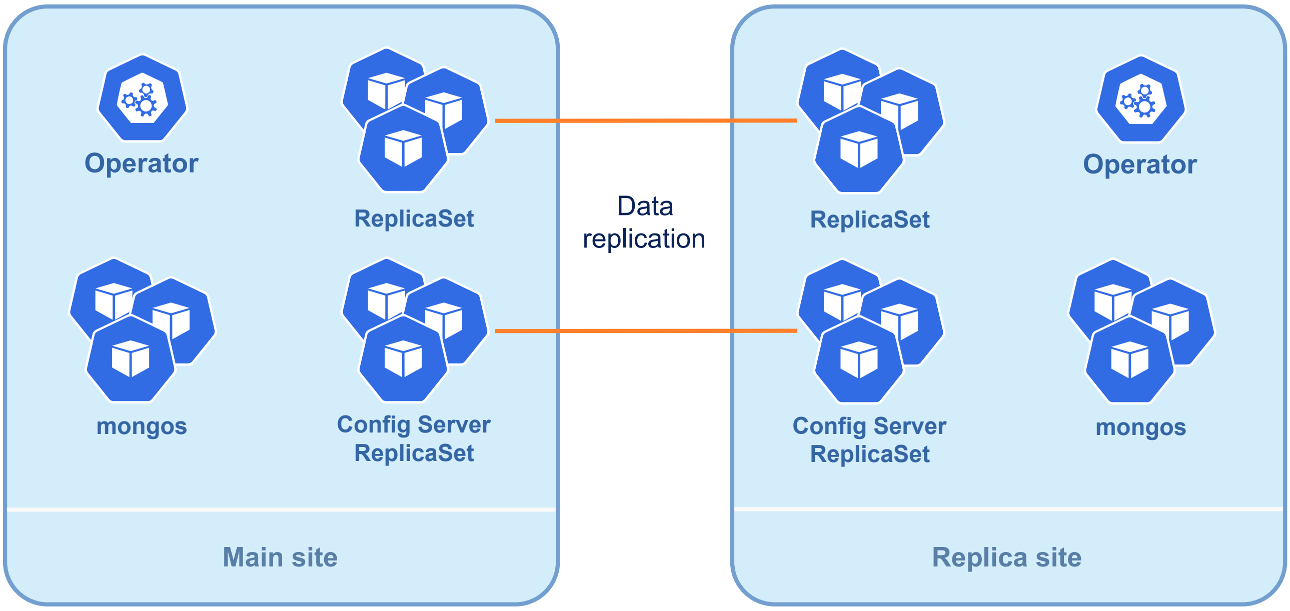
- **Multi-cluster** deployments span multiple Kubernetes clusters, typically within the same cloud provider or region. This model is ideal for high availability, staging/production isolation, or cluster migration.
- **Multi-region** deployments extend MongoDB across geographically distributed data centers or cloud regions. This setup supports disaster recovery, latency optimization, and jurisdictional data control.

While the underlying mechanics such as replica sets, TLS, and service exposure are similar, multi-region deployments introduce additional complexity around DNS, network reachability, and manual configuration.

Cross-site replication

To maintain the same set of data in clusters within multi-cluster or multi-region deployment, the Operator uses the cross-site replication. This means that one cluster is the Main site and another one(s) - the Replica site(s).

The following diagram shows how the data is replicated between the sites



- **Main site**: This is the authoritative cluster. It runs the primary node which accepts the write traffic. The Operator fully controls this site, managing the replica set configuration, backups, user credentials and other operations.
- **Replica site**: These are secondary clusters that host MongoDB nodes and replicate data from the Main site. The Operator deploys this site in passive mode and doesn't control the replica set configuration there. The passive mode is set by the `unmanaged: true` flag in the Custom Resource.

This separation ensures consistency and avoids conflicts when managing distributed deployments.

Why to use multi-cluster or multi-region?

Choosing the right topology depends on your goals. Here are common use cases that you can achieve with these models:

- High availability - Spread MongoDB nodes across clusters to avoid single points of failure. If one cluster goes down, others remain operational.
- Staging vs Production Isolation - Run isolated environments with shared data topology. Test changes safely without impacting production.
- Cluster migration - Move workloads between clusters or cloud providers with minimal downtime.
- Disaster recovery - Replicate data across regions to survive outages. Even if an entire data center fails, your application stays online.
- Geo-distributed applications - Serve users from the nearest region to reduce latency and improve experience.
- Compliance isolation - Keep data within specific jurisdictions to meet regulatory requirements.

Next steps

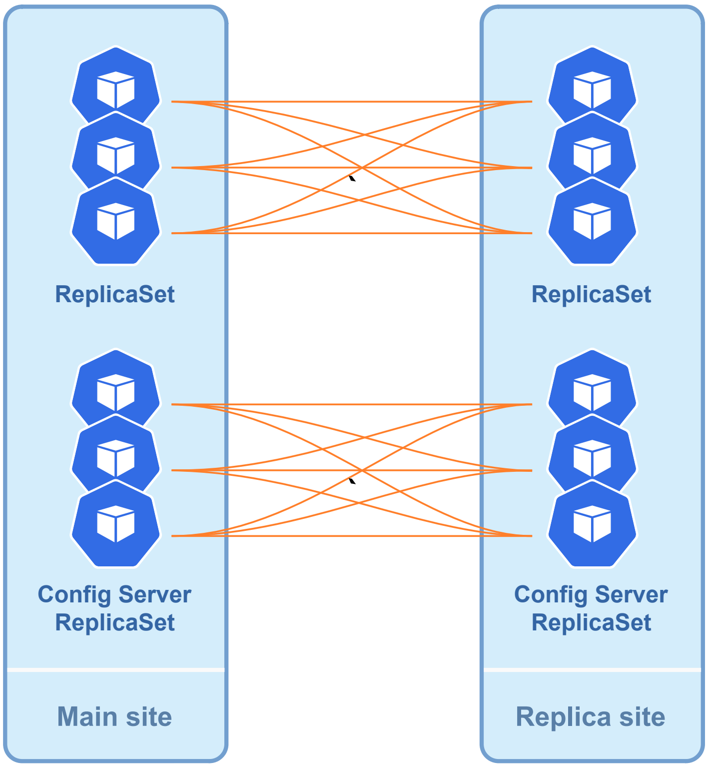
Plan your deployment

8.3.2 Plan your multi-cluster or multi-region deployment

Requirements

Regardless of topology, successful deployments share several technical requirements:

- Network connectivity: Every node in both the Main and the Replica sites must be able to reach each other over the network, like in full mesh:



  To make this happen in Kubernetes clusters, you need to expose all Replica Set nodes (including Config Servers) on both sites through a dedicated Service. To learn more, see the Service per Pod section in the Exposing the cluster chapter.

- User credentials must be the same in both clusters.
- TLS certificates must be the same in both clusters.

You must have a single *Main* cluster but you can have multiple *Replica* clusters as long as you don't have more than 50 members in the Replica Set. This limitation comes from MongoDB itself, for more information please check MongoDB documentation.

Topologies

The Operator automates configuration of *Main* and *Replica* MongoDB sites when you run them on Kubernetes. However, multi-cluster or multi-region deployment is not bound to Kubernetes. Either *Main* or *Replica* can run outside of Kubernetes, be regular MongoDB and be out of the Operator's control.

The following topologies are supported:

Main and Replica clusters on Kubernetes.

This is the Kubernetes-native deployment. You can automate the cluster management using the Multi-Cluster Services.

The Deployment section focuses on this topology and provides the setup steps.

Main cluster on Kubernetes and Replica cluster outside of Kubernetes.

You can deploy or reuse an existing cluster on Kubernetes as the *Main* site and have the *Replica* cluster outside of Kubernetes.

The setup steps are:

1. Deploy the *Main* cluster on a Kubernetes cluster (or use an existing one)
2. Export TLS certificates and user Secrets from the main site and use them to configure the Replica site.
3. Deploy the *Replica* cluster wherever you want.
4. Interconnect sites by adding nodes from the *Replica* cluster to the *Main* cluster as external nodes.

Main cluster outside of Kubernetes and Replica cluster on Kubernetes

You can deploy or reuse an existing cluster outside of Kubernetes as the *Main* site and run the *Replica* cluster on Kubernetes.

The setup steps are:

1. Deploy the *Main* cluster wherever you want (or use an existing one)
2. Export the TLS certificates and user credentials from the main site.
3. Create the Kubernetes Secrets objects: for TLS certificates and for User secrets. You will use these Secrets to deploy the Replica site.
4. Deploy the *Replica* cluster on a Kubernetes cluster using the Secrets you created.
5. Interconnect sites by adding nodes from the *Replica* cluster to the *Main* cluster using the MongoDB client.

Next steps

Multi-cluster services

Deploy the Main site

8.3.3 Multi-cluster Services

8.3.3.1 Multi-cluster Services

Multi-cluster Services (MCS) is a cross-cluster discovery and invocation mechanism that uses the existing Service object.

MCS allows you to create a "fleet" of Kubernetes clusters that share a common identity and are managed as a single logical unit. This enables service discovery and communication across clusters via a virtual IP address, simplifying the process of building multi-region or multi-cluster deployments.

Multi-cluster Services should be supported by the cloud provider. It is natively supported by Google Kubernetes Engine (GKE). Amazon Elastic Kubernetes Service (EKS) provides multi-cluster Services via the AWS Cloud Map.

Use multi-cluster Services

To use multi-cluster Services for your deployment, you must do the following:

- Enable multi-cluster Services with your cloud provider
- Configure the Operator to use multi-cluster Services

MCS can charge cross-site replication with additional limitations specific to the cloud provider. For example, GKE demands all participating Pods to be in the same project. Also, consider using a custom namespace for exporting Services. Using the `default` and `kube-system` Namespaces can cause unintended name conflicts and the resulting unintended grouping.

Enable multi-cluster Services with your cloud provider

To get started, follow the setup guides for your specific cloud provider:

- Enable multi-cluster Services on GKE
- Enable multi-cluster Services on EKS

Configure the Operator to use multi-cluster Services

To work in multi-cluster Kubernetes environment, the Operator must ensure service discovery across clusters.

To do this, the Operator must create the ServiceExport and ServiceImport resources.

A **ServiceExport** is a Kubernetes resource that marks a standard Service for sharing across clusters. When created, ServiceExport signals to the MCS controller that the service with the same name should be made discoverable to other clusters.

The Operator creates the ServiceExport resource for a cluster when the `multiCluster` subsection of the `deploy/cr.yaml` contains the following configuration:

- the `multiCluster.enabled` key is set to `true`
- the `multiCluster.DNSSuffix` string is equal to the cluster domain suffix for multi-cluster Services used by Kubernetes. The default value is `svc.clusterset.local`.

```
...
multiCluster:
  enabled: true
  DNSSuffix: svc.clusterset.local
...
```

For a Service to be exported and become accessible by other clusters of the fleet, it must have the same name and namespace in each cluster. Once exported, the service is recognized as a single combined Service. It can be resolved from any Pod in any fleet cluster via the shared DNS name:

```
SERVICE_NAME.NAMESPACE.svc.clusterset.local
```

It takes approximately five minutes to create ServiceExport and sync with the clusters of the fleet. You can check the list of services for export with the following commands:

```
kubectl get serviceexport
```

Expected output

```
NAME                    AGE
my-cluster-name-cfg     22m
my-cluster-name-cfg-0   22m
my-cluster-name-cfg-1   22m
my-cluster-name-cfg-2   22m
my-cluster-name-mongos  22m
my-cluster-name-rs0     22m
my-cluster-name-rs0-0   22m
my-cluster-name-rs0-1   22m
my-cluster-name-rs0-2   22m
```

A **ServiceImport** is a Kubernetes resource to consume exported services in each importing cluster. This is analogous to the traditional Service type in Kubernetes. The ServiceImport is created automatically by the MCS controller. It contains endpoint information from all clusters that exported the service and enables workloads in one cluster to access services in another using the unified DNS name.

To check the list of services for import, run this command:

```
kubectl get serviceimport
```

Expected output

```
NAME                    TYPE           IP                    AGE
my-cluster-name-cfg     Headless                             22m
my-cluster-name-cfg-0   ClusterSetIP   ["10.73.200.89"]      22m
my-cluster-name-cfg-1   ClusterSetIP   ["10.73.192.104"]     22m
my-cluster-name-cfg-2   ClusterSetIP   ["10.73.207.254"]     22m
my-cluster-name-mongos  ClusterSetIP   ["10.73.196.213"]     22m
my-cluster-name-rs0     Headless                             22m
my-cluster-name-rs0-0   ClusterSetIP   ["10.73.206.24"]      22m
my-cluster-name-rs0-1   ClusterSetIP   ["10.73.207.20"]      22m
my-cluster-name-rs0-2   ClusterSetIP   ["10.73.193.92"]      22m
```

Since ServiceImport is not controlled by the Operator, objects you must check the MCS controller installed by your cloud provider if you need to troubleshoot it.

Next steps

Enable MCS on GKE   Enable MCS on EKS

8.3.3.2 Enable multi-cluster Services on GKE

This document provides instructions how to enable multi-cluster Services on GKE. To learn more about multi-cluster services concept, see Multi-cluster Services.

The recommended approach is to use Workload Identity Federation. Workload Identity Federation allows Kubernetes service accounts to impersonate Google Cloud IAM service accounts. This means:

- You don't need to store and mount service account keys in Pods.
- You can assign fine-grained IAM roles to workloads.
- Authentication is handled securely and natively.

Before you start

1. Check the requirements for MCS on GKE and ensure your infrastructure meets them.
2. Ensure your account has the following roles:
   - roles/container.admin
   - roles/iam.serviceAccountAdmin

Procedure

1. Export your GKE project ID as an environment variable to simplify further configuration

```
PROJECT_ID=<your-project-id>
```

2. Enable the MCS, fleet (hub), Resource Manager, Cloud Service Mesh, and Cloud DNS APIs for your account:

```
gcloud services enable \
multiclusterservicediscovery.googleapis.com \
gkehub.googleapis.com \
cloudresourcemanager.googleapis.com \
trafficdirector.googleapis.com \
dns.googleapis.com \
--project $PROJECT_ID
```

3. Enable multi-cluster Services for your project on GKE:

```
gcloud container fleet multi-cluster-services enable --project $PROJECT_ID
```

4. Create two clusters and enable Workload Identity for them. Let's name the clusters `main` and `replica`:
   - Create the `main` cluster:

```
gcloud container clusters create main-cluster \
  --zone us-central1-a \
  --cluster-version 1.33 \
  --machine-type n1-standard-4 \
  --num-nodes=3 \
  --workload-pool=$PROJECT_ID.svc.id.goog
```

   - Create the `replica` cluster:

```
gcloud container clusters create replica-cluster \
  --zone us-central1-a \
  --cluster-version 1.33 \
  --machine-type n1-standard-4 \
  --num-nodes=3 \
  --workload-pool=$PROJECT_ID.svc.id.goog
```

5. Add clusters to the fleet and enable Workload Identity Federation:
   - Add the main cluster

```
gcloud container fleet memberships register main-cluster \
  --gke-cluster us-central1-a/main-cluster \
  --enable-workload-identity
```

- Add the replica cluster

```
gcloud container fleet memberships register replica-cluster \
  --gke-cluster us-central1-a/replica-cluster \
  --enable-workload-identity
```

6. Enable MCS importer to manage Identity and Access Management (IAM) permissions.

- Extract the Project number and set it as the environment variable:

```
PROJECT_NUMBER=$(gcloud projects describe $PROJECT_ID --format="value(projectNumber)")
```

- Enable IAM permissions:

```
gcloud projects add-iam-policy-binding $PROJECT_ID \
  --member
"principal://iam.googleapis.com/projects/$PROJECT_NUMBER/locations/global/workloadIdentityPools/$PROJECT_ID.svc.id.goog/sub
ject/ns/gke-mcs/sa/gke-mcs-importer" \
  --role "roles/compute.networkViewer"
```

7. Verify that MCS is enabled:

```
gcloud container fleet multi-cluster-services describe --project $PROJECT_ID
```

Sample output

```
createTime: '2021-11-10T09:31:41.578155328Z'
membershipStates:
  projects/166042509722/locations/us-central1/memberships/main-cluster:
    state:
      code: OK
      description: Firewall successfully updated
      updateTime: '2025-09-26T11:01:09.038866570Z'
  projects/166042509722/locations/us-central1/memberships/replica-cluster:
    state:
      code: OK
      description: Firewall successfully updated
      updateTime: '2025-09-26T11:01:44.661916334Z'
name: projects/cloud-dev-112233/locations/global/features/multiclusterservicediscovery
resourceState:
  state: ACTIVE
spec: {}
updateTime: '2024-05-16T08:02:36.718079209Z'
```

Next steps

Deploy the Main site

8.3.3.3 Enable multi-cluster Services on EKS

The AWS Cloud Map MCS Controller is an open-source Kubernetes controller that implements the Multi-Cluster Services API using AWS Cloud Map as the backend. It allows services exported from one cluster to be discovered and consumed in another using the DNS format:

```
SERVICE_NAME.NAMESPACE.svc.clusterset.local
```

Read more about how AWS Cloud Map MCS Controller works in the AWS blog post.

Also, learn more about AWS Cloud Map pricing

Prerequisites

Before you get started with MCS on EKS, ensure you have the following:

1. Two EKS clusters that can communicate with each other over the Virtual Private Cloud (Amazon VPC) peering. See Create the EKS cluster guide for the cluster creation steps.
2. Each EKS cluster has permissions to communicate with AWS Cloud Map. A service account must have the IAM policy assigned that grants access to the AWS Cloud Map. For testing purposes, you can use the *AWSCloudMapFullAccess* policy. In production, apply least privilege permissions.

   For more information about IAM policies for AWS Cloud Map, see the Identity and Access Management for AWS Cloud Map documentation.

Configuration

Follow the steps from the AWS Cloud Map MCS Controller for K8s guide to configure multi-cloud Services.

Before you enable MCS on the clusters, create the ClusterProperty objects on each cluster:

```
apiVersion: about.k8s.io/v1alpha1
kind: ClusterProperty
metadata:
  name: cluster.clusterset.k8s.io
spec:
  value: [Your Cluster identifier]
---
apiVersion: about.k8s.io/v1alpha1
kind: ClusterProperty
metadata:
  name: clusterset.k8s.io
spec:
  value: [Your ClusterSet identifier]
```

Check the AWS MCS controller repository for more information.

Next steps

Deploy the Main site

8.3.3.4 Apply MCS to an existing cluster

You can turn on MCS for the **already-existing non-MCS cluster**. To do this:

- Restart the Operator after editing the `multiCluster` subsection keys and applying `deploy/cr.yaml`. Find the Operator's Pod name in the output of the `kubectl get pods` command (it will be something like `percona-server-mongodb-operator-d859b69b6-t44vk`) and delete it as follows:

```
kubectl delete percona-server-mongodb-operator-d859b69b6-t44vk
```

- If you are enabling MCS for a running cluster after upgrading from the Operator version `1.11.0` or below, you need rotating multi-domain (SAN) certificates. Do this by pausing the cluster and deleting TLS Secrets.

8.3.4 Deployment

8.3.4.1 Configure the Main site

This guide shows you how to set up the Main Percona Server for MongoDB site for a multi-cluster deployment. The steps focus on Kubernetes with [multi-cluster Services enabled on Google Kubernetes Engine](), but you can also use them for a standard Kubernetes environment.

Before you start

Clone the repository with all manifests and source code. You'll need it to edit configuration files for the database clusters, Secrets, backups and restores. Run the following command:

```
git clone -b v1.22.0 https://github.com/percona/percona-server-mongodb-operator
```

Make sure to clone the correct branch. The branch name is the same as the Operator release version.

Initial preparation

When you manage multiple clusters, creating a separate kubeconfig file for each one helps you avoid accidentally running commands on the wrong cluster. This keeps your environments organized and reduces the risk of making changes in the wrong place.

1. Create a `kubeconfig` file and export it as an environment variable
   - For the `main` cluster:

   ```
   export KUBECONFIG=./main_config gcloud container clusters get-credentials main-cluster --zone us-central1-a
   ```

   - For the `replica` cluster:

   ```
   export KUBECONFIG=./replica_config gcloud container clusters get-credentials replica-cluster --zone us-central1-a
   ```

2. Set the context for the clusters from their respective kubeconfig files.
   - On the `main` cluster:

   ```
   kubectl --kubeconfig main_config config set-context $(kubectl config current-context)
   ```

   - On the `replica` cluster:

   ```
   kubectl --kubeconfig replica_config config set-context $(kubectl config current-context)
   ```

3. Grant your Google Cloud user permissions to manage clusters. To do this, create a ClusterRoleBinding binding of the `cluster-admin` ClusterRole to your account for each cluster. Specify different names for each cluster to avoid naming collision:
   - On the `main` cluster:

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

   - On the `replica` cluster:

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

4. Create the same namespace on both clusters and set the context to point to this namespace. The namespace must be the same because it is a part of the shared DNS used to identify and resolve services across clusters.

   Run this command on both clusters to create the `example` namespace. Use your own value:

   ```
   kubectl create namespace example
   kubectl config set-context --current --namespace=example
   ```

Install the Operator and Percona Server for MongoDB

1. Install the Operator deployment:

```
kubectl apply --server-side -f https://raw.githubusercontent.com/percona/percona-server-mongodb-
operator/v1.22.0/deploy/bundle.yaml -n <namespace>
```

Expected output

```
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbs.psmdb.percona.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbbackups.psmdb.percona.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbrestores.psmdb.percona.com serverside-applied
role.rbac.authorization.k8s.io/percona-server-mongodb-operator serverside-applied
serviceaccount/percona-server-mongodb-operator serverside-applied
rolebinding.rbac.authorization.k8s.io/service-account-percona-server-mongodb-operator serverside-applied
deployment.apps/percona-server-mongodb-operator serverside-applied
```

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

2. Prepare Percona Server for MongoDB configuration on the main cluster to include the following:

- Name your cluster to differentiate the main and replica one. For example, name it `main-cluster`.
- Replica set, config server replica set and mongos Pods are exposed with the ClusterIP Service type. This type is required by the multi-cluster services as the nodes will communicate internally
- Multi-cluster services are enabled. Learn more about preparing the Operator for multi-services

The sample configuration looks like this:

cr-main.yaml

```yaml
apiVersion: psmdb.percona.com/v1
kind: PerconaServerMongoDB
metadata:
  name: main-cluster
updateStrategy: SmartUpdate
multiCluster:
  enabled: true
  DNSSuffix: svc.clusterset.local
upgradeOptions:
  versionServiceEndpoint: https://check.percona.com
  apply: disabled
  schedule: "0 2 * * *"
  setFCV: false
secrets:
  users: my-cluster-name-secrets
  encryptionKey: my-cluster-name-mongodb-encryption-key
...
replsets:
- name: rs0
  size: 3
  expose:
    enabled: true
    type: ClusterIP
....
sharding:
  enabled: true
  configsvrReplSet:
    size: 3
    expose:
      enabled: true
      type: ClusterIP
    volumeSpec:
      persistentVolumeClaim:
        resources:
          requests:
            storage: 3Gi

mongos:
  size: 3
  expose:
    type: ClusterIP
```

3. Apply the configuration to deploy Percona Server for MongoDB:

```
kubectl apply -f cr-main.yaml
```

Export the cluster secrets and certificates to be copied from Main to Replica

The *Main* and *Replica* sites must have the same same users credentials and TLS certificates to be able to communicate with each other. To do this, export the Secrets from the `main` cluster and recreate them on the `replica` cluster.

1. List the Secrets objects:

   ```
   kubectl get secrets
   ```

   The Secrets you are interested in are the following:

   - `main-cluster-name-ssl` - SSL certificates for client connections,
   - `main-cluster-name-ssl-internal` - SSL certificates for replication,
   - `my-cluster-name-secrets` - user credentials,
   - `my-cluster-name-mongodb-encryption-key` - encryption key file.

2. Export each Secret to a file:

   ```
   kubectl get secret my-cluster-name-secrets -o yaml > my-cluster-secrets.yml
   kubectl get secret main-cluster-ssl  -o yaml > main-cluster-ssl.yml
   kubectl get secret main-cluster-ssl-internal -o yaml > main-cluster-ssl-internal.yml
   kubectl get secret my-cluster-name-mongodb-encryption-key -o yaml > my-cluster-name-mongodb-encryption-key.yml
   ```

3. Remove the `annotations`, `creationTimestamp`, `resourceVersion`, `selfLink`, and `uid` metadata fields from the resulting file to make it ready for the `replica` site.

   Use the following scripts:

   ```
   yq eval 'del(.metadata.ownerReferences, .metadata.annotations, .metadata.creationTimestamp, .metadata.resourceVersion,
   .metadata.selfLink, .metadata.uid)' my-cluster-secrets.yml > my-cluster-secrets-replica.yaml
   sed -i '' 's/main-cluster/replica-cluster/g' my-cluster-secrets-replica.yaml

   yq eval 'del(.metadata.ownerReferences, .metadata.annotations, .metadata.creationTimestamp, .metadata.resourceVersion,
   .metadata.selfLink, .metadata.uid)' main-cluster-ssl.yml > replica-cluster-ssl.yml
   sed -i '' 's/main-cluster/replica-cluster/g' replica-cluster-ssl.yml

   yq eval 'del(.metadata.ownerReferences, .metadata.annotations, .metadata.creationTimestamp, .metadata.resourceVersion,
   .metadata.selfLink, .metadata.uid)' main-cluster-ssl-internal.yml > replica-cluster-ssl-internal.yml
   sed -i '' 's/main-cluster/replica-cluster/g' replica-cluster-ssl-internal.yml


   yq eval 'del(.metadata.ownerReferences, .metadata.annotations, .metadata.creationTimestamp, .metadata.resourceVersion,
   .metadata.selfLink, .metadata.uid)' my-cluster-name-mongodb-encryption-key.yml > my-cluster-name-mongodb-encryption-
   key2.yml
   sed -i '' 's/main-cluster/replica-cluster/g' my-cluster-name-mongodb-encryption-key2.yml
   ```

   The commands do the following for each file:

   - Remove metadata fields that are unique to the original cluster (like annotations, timestamps, and IDs) from the Secret YAML file, making it suitable for use in the replica cluster
   - Update the Secret file by replacing all instances of “main-cluster” with “replica-cluster”, so the secret matches the replica cluster’s naming.

   You will need to further apply these secrets on Replica.

Next steps

Configure Replica site

8.3.4.2 Configure the Replica site

When the Operator creates a new cluster, a lot of things are happening, such as electing the Primary, generating certificates, and picking specific names. This should not happen on the *Replica* site. Therefore, you deploy the Replica site in an unmanaged mode.

> **Note**
>
> Setting `unmanaged` to true will not only prevent the Operator from controlling the Replica Set configuration, but it will also result in not generating certificates and users credentials for new clusters.

For the Main and Replica sites to communicate, they must have the same the user and TLS Secrets.

1. Ensure you have created the same namespace as on the main site and set the context to it so that subsequent commands are executed in that namespace.

   ```
   kubectl get namespaces
   kubectl config get-contexts
   ```

   > **Sample output**
   >
   > ```{.text .no-copy} CURRENT NAME CLUSTER AUTHINFO NAMESPACE * gke_*us-central1-a-replica-cluster gkeus-central1-a-replica-cluster gke*_us-central1-a-replica-cluster example

2. Create the Secrets from the secrets files you prepared from the main cluster.

   ```
   kubectl apply -f my-cluster-secrets-replica.yaml
   kubectl apply -f replica-cluster-ssl.yml
   kubectl apply -f replica-cluster-ssl-internal.yml
   kubectl apply -f my-cluster-name-mongodb-encryption-key2.yml
   ```

   > **Sample output**
   >
   > ```
   > secret/my-cluster-name-secrets created
   > secret/replica-cluster-ssl created
   > secret/replica-cluster-ssl-internal created
   > secret/my-cluster-name-mongodb-encryption-key created
   > ```

   Replica will not start if the TLS secrets and the encryption key are not copied. If users are not copied, the replica will join the replica set, but it will be restarting due to failed liveness checks.

3. Prepare the Replica site configuration:

   - Name your cluster. The name must match the names of the Secrets objects you created. For example, `replica-site`
   - Set the `spec.unmanaged` to `true`
   - Enable multi-cluster services in the `spec.multiCluster` subsection.
   - Set the `updateStrategy` key to `RollingUpdate`, because Smart Updates are not allowed on unmanaged clusters.
   - Reference the Secrets you created in the `spec.Secrets` section
   - Expose the Replica set, config server replica set and mongos Pods with the `ClusterIP` type.

   Also, the Operator versions prior to 1.19.0 did not support backups on unmanaged clusters, so set `backup.enabled` to `false` for the Operator 1.18.0 and older.

   Here is an example:

```yaml
apiVersion: psmdb.percona.com/v1
kind: PerconaServerMongoDB
metadata:
    name: replica-cluster
spec:
    unmanaged: true
    multiCluster:
        enabled: true
        DNSSuffix: svc.clusterset.local
    updateStrategy: RollingUpdate
    upgradeOptions:
        apply: disabled
        schedule: "0 2 * * *"
    secrets:
        users: my-cluster-name-secrets
        encryptionKey: my-cluster-name-mongodb-encryption-key
        ssl: replica-cluster-ssl
        sslInternal: replica-cluster-ssl-internal
    replsets:
    - name: rs0
        size: 3
        expose:
            enabled: true
            type: ClusterIP
        volumeSpec:
            persistentVolumeClaim:
                resources:
                    requests:
                        storage: 3Gi

    sharding:
        enabled: true
        configsvrReplSet:
            size: 3
            expose:
                enabled: true
                type: ClusterIP
            volumeSpec:
                persistentVolumeClaim:
                    resources:
                        requests:
                            storage: 3Gi

        mongos:
            size: 1
            expose:
                type: ClusterIP
```

4. Apply the configuration to deploy the Replica site

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

Next steps

Interconnect sites for replication

8.3.4.3 Interconnect sites for replication

At this step, you should let the clusters know about each other and interconnect them for replication. To do so, you need to add the Replica site's nodes as external nodes for the Main site. In the same way, you add the Main's site nodes as external ones for the Replica site.

Every site has three replica set members and three config server replica set members. But you add only two of them as voting members, while the third member is added as a non-voting one. In doing so, you avoid split-brain situations and prevent the primary elections if the Replica site is down or there is a network disruption between the sites.

In this way, the `main` cluster managed by the Operator is able to reach the the `replica` nodes.

List the services

Ensure that the nodes are exposed by listing the services. Run the following command on both clusters:

```
kubectl get services
```

Sample output

This is the sample output for the `main` cluster

```
gke-mcs-6i19n16iou    ClusterIP     34.118.226.213    <none>          27017/TCP         30m
gke-mcs-7kc05t4an8    ClusterIP     34.118.232.25     <none>          27017/TCP         34m
gke-mcs-8rnph5kn83    ClusterIP     34.118.232.15     <none>          27017/TCP         30m
gke-mcs-c2rifb19i2    ClusterIP     34.118.227.18     <none>          27017/TCP         37m
gke-mcs-d2a5n9jthp    ClusterIP     34.118.238.169    <none>          27017/TCP         39m
gke-mcs-eu3ms5pd46    ClusterIP     34.118.233.180    <none>          27017/TCP         25m
gke-mcs-fvvrfrc8lj    ClusterIP     34.118.229.203    <none>          27017/TCP         30m
gke-mcs-gqv8a8hvj0    ClusterIP     34.118.239.68     <none>          27017/TCP         39m
gke-mcs-i7esf6en63    ClusterIP     34.118.231.126    <none>          27017/TCP         27m
gke-mcs-jrvipn8erd    ClusterIP     34.118.235.246    <none>          27017/TCP         34m
gke-mcs-mefti0gh66    ClusterIP     34.118.236.218    <none>          27017/TCP         37m
gke-mcs-q1jve1uq67    ClusterIP     34.118.225.234    <none>          27017/TCP         25m
gke-mcs-rp77hijaj0    ClusterIP     34.118.229.123    <none>          27017/TCP         39m
gke-mcs-tjbtnp3ica    ClusterIP     34.118.236.161    <none>          27017/TCP         27m
main-cluster-cfg      ClusterIP     None              <none>          27017/TCP         44m
main-cluster-cfg-0    ClusterIP     34.118.227.52     34.42.135.122   27017:31233/TCP   44m
main-cluster-cfg-1    ClusterIP     34.118.233.187    34.135.82.96    27017:31941/TCP   43m
main-cluster-cfg-2    ClusterIP     34.118.227.173    35.202.26.197   27017:30961/TCP   42m
main-cluster-mongos   ClusterIP     34.118.233.114    <none>          27017/TCP         44m
main-cluster-rs0      ClusterIP     None              <none>          27017/TCP         44m
main-cluster-rs0-0    ClusterIP     34.118.231.58     <none>          27017/TCP         44m
main-cluster-rs0-1    ClusterIP     34.118.239.165    <none>          27017/TCP         43m
main-cluster-rs0-2    ClusterIP     34.118.225.3      <none>          27017/TCP         42m
```

Add the Replica site nodes to the Main site

1. Modify the `deploy/cr-main.yaml` file of the Main site and define the exposed nodes of the Replica site in the `replsets.externalNodes` and `sharding.configsvrReplset.externalNodes` subsections. For each node, specify the following:

   - set `host` to the URL of the external replset instance. When exposed, a node has its own service, recognized by the domain name `<service-name>.<namespace>.svc.clusterset.local`
   - set `port` to the port number of the external node. If not set, the default `27017` port is used,
   - set `priority` to define the [priority] of the external node. The default priority for local members of the cluster is `2`. When you add external nodes, set the lower priority to avoid an unmanaged node being elected as a primary. A zero `0` priority adds the node as a [non-voting member].
   - set `votes` to the number of [votes] an external node can cast in a replica set election (`0` is default and should be used for non-voting members of the cluster).

   Here is an example:

```
apiVersion: psmdb.percona.com/v1
kind: PerconaServerMongoDB
metadata:
  name: main-cluster
spec:
...
  updateStrategy: SmartUpdate
  multiCluster:
    enabled: true
    DNSSuffix: svc.clusterset.local
  upgradeOptions:
    apply: disabled
    schedule: "0 2 * * *"
  secrets:
    users: my-cluster-name-secrets
    encryptionKey: my-cluster-name-mongodb-encryption-key
  replsets:
  - name: rs0
    size: 3
    externalNodes:
    - host: replica-cluster-rs0-0.psmdb.svc.clusterset.local
      votes: 1
      priority: 1
    - host: replica-cluster-rs0-1.psmdb.svc.clusterset.local
      votes: 1
      priority: 1
    - host: replica-cluster-rs0-2.psmdb.svc.clusterset.local
      votes: 0
      priority: 0
      expose:
      enabled: true
      type: ClusterIP

  sharding:
    enabled: true
    configsvrReplSet:
      size: 3
      externalNodes:
      - host: replica-cluster-cfg-0.psmdb.svc.clusterset.local
        votes: 1
        priority: 1
      - host: replica-cluster-cfg-1.psmdb.svc.clusterset.local
        votes: 1
        priority: 1
      - host: replica-cluster-cfg-2.psmdb.svc.clusterset.local
        votes: 0
        priority: 0
      expose:
        enabled: true
        type: ClusterIP

    mongos:
      size: 3
      expose:
        type: ClusterIP
```

2. Apply the changes:

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

Add Main site nodes to the Replica site

1. Modify the `deploy/cr-main.yaml` file of the Replica site and define the exposed nodes of the Main site in the `replsets.externalNodes` and `sharding.configsvrReplset.externalNodes` subsections. For each node, specify the following:

```yaml
apiVersion: psmdb.percona.com/v1
kind: PerconaServerMongoDB
metadata:
  name: replica-cluster
spec:
...
  replsets:
  - name: rs0
    size: 3
    externalNodes:
    - host: main-cluster-rs0-0.psmdb.svc.clusterset.local
        votes: 1
        priority: 1
    - host: main-cluster-rs0-1.psmdb.svc.clusterset.local
        votes: 1
        priority: 1
    - host: main-cluster-rs0-2.psmdb.svc.clusterset.local
        votes: 0
        priority: 0

  sharding:
    configsvrReplSet:
        externalNodes:
        - host: main-cluster-cfg-0.psmdb.svc.clusterset.local
          votes: 1
          priority: 1
        - host: main-cluster-cfg-1.psmdb.svc.clusterset.local
          votes: 1
          priority: 1
        - host: main-cluster-cfg-2.psmdb.svc.clusterset.local
          votes: 0
          priority: 0
```

2. Apply the configuration:

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

Check cluster connectivity

Verify that the clusters are interconnected by connecting to one of them.

1. Connect to one of the Pods directly using the credentials of the database admin user. Refer to the Connect to Percona Server for MongoDB tutorial how to retrieve user credentials:

```
kubectl exec -it main-cluster-rs0-0 -- mongosh -u databaseAdmin -p <dbAdminPassword>
```

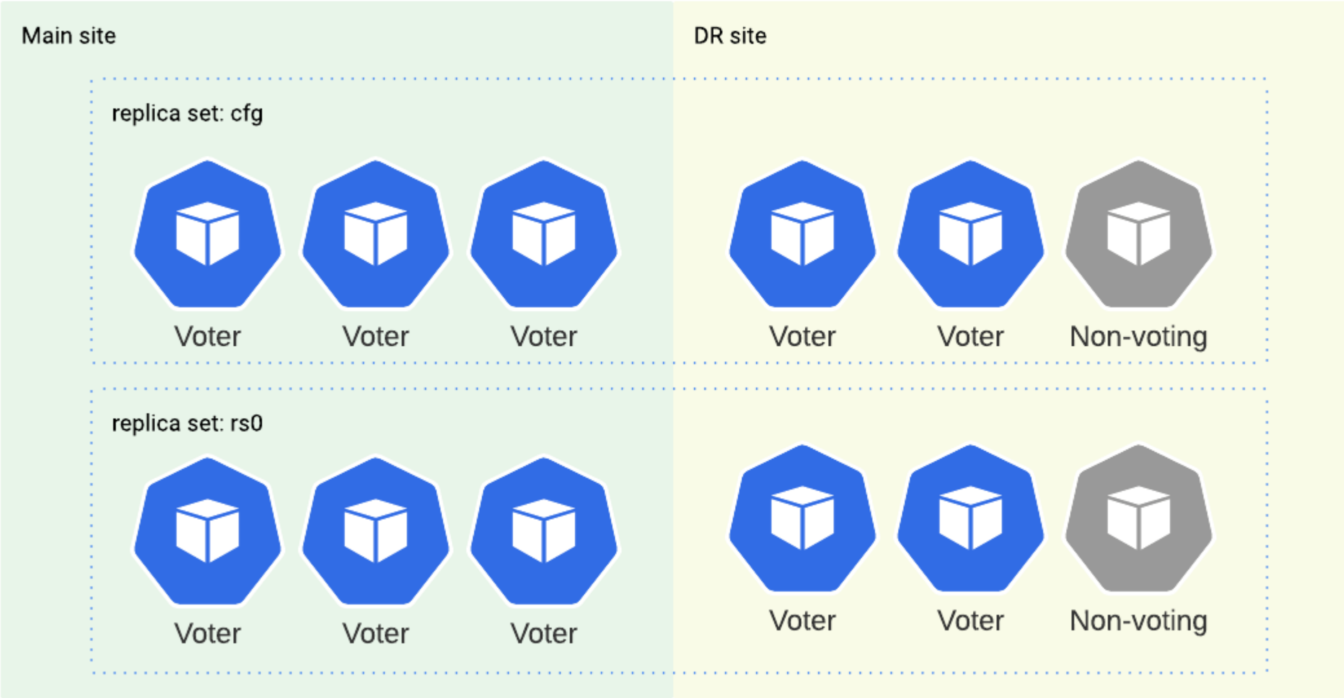
2. List replica set members:

```
rs.status().members
```

Sample output

```
{
    _id: 0,
    name: 'main-cluster-rs0-0.psmdb.svc.clusterset.local:27017',
    health: 1,
    state: 1,
    stateStr: 'PRIMARY',
    .....
},
{
    _id: 1,
    name: 'main-cluster-rs0-1.psmdb.svc.clusterset.local:27017',
    health: 1,
    state: 2,
    stateStr: 'SECONDARY',
    .....
},
{
    _id: 2,
    name: 'main-cluster-rs0-2.psmdb.svc.clusterset.local:27017',
    health: 1,
    state: 2,
    stateStr: 'SECONDARY',
    .....
},
{
    _id: 3,
    name: 'replica-cluster-rs0-0.psmdb.svc.clusterset.local:27017',
    health: 1,
    state: 2,
    stateStr: 'SECONDARY',
    ......
},
{
    _id: 4,
    name: 'replica-cluster-rs0-1.psmdb.svc.clusterset.local:27017',
    health: 1,
    state: 2,
    stateStr: 'SECONDARY',
    ....
},
{
    _id: 5,
    name: 'replica-cluster-rs0-2.psmdb.svc.clusterset.local:27017',
    health: 1,
    state: 2,
    stateStr: 'SECONDARY',
    ....
}
```

The members from both clusters appear in the list and the configuration looks like this:



Next steps

Test services failover

8.3.4.4 Fail over services to the Replica site

Failing over services to the Replica site ensures your applications remain available if the Main site needs maintenance or becomes unavailable. You might need to do this during planned maintenance windows or in response to unexpected outages. The following sections explain how to handle both planned and unplanned failover scenarios.

Planned services switchover

You can switch over services to the Replica site while doing some planned maintenance on the Main site.

Here's how to do it:

1. Set the Main site to the unmanaged mode and change the Update strategy to RollingUpdate. Modify the `deploy/cr-main.yaml` file:

```
spec:
  unmanaged: true
  updateStrategy: RollingUpdate
```

2. Apply the configuration:

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

3. Put the Replica site in the managed mode:

```
spec:
  unmanaged: false
  updateStrategy: SmartUpdate
```

4. Apply the configuration:

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

5. Connect to one of the Replica site Pods and check the replica set status. You should see that it has re-elected the new primary.

Fail over services in a disaster recovery scenario

A disaster can strike at any moment and the Main site may be down or unavailable. In this case, you must fail over the services to the Replica site.

Here's how to do it:

1. Connect to one of the replica set Pods on the Replica site. Since you will be reconfiguring the replica set, you must connect as the MongoDB `clusterAdmin` user:

```
kubectl exec -it replica-cluster-rs0-0 -- mongosh -u clusterAdmin -p <clusterAdminPassword>
```

2. Check the current replica set status:

```
rs.status().members
```

Sample output

```
{
    _id: 0,
    name: 'main-cluster-rs0-0.psmdb.svc.clusterset.local:27017',
    health: 0,
    state: 8,
    stateStr: '(not reachable/healthy)',
    uptime: 0,
    ...
  },
```

3. Retrieve and store the current configuration in the variable:

```
cfg = rs.config()
```

4. Override the member array to include the surviving members - the ones from the Replica site. For the following command replace the member indexes with the ones you got from the `rs.config()` output:

```
cfg.members = [cfg.members[3], cfg.members[4], cfg.members[5]]
```

5. Reconfigure the replica set passing the updated member list:

```
rs.reconfig(cfg, {force: true})
```

6. Check the updated configuration:

```
rs.status().members
```

Sample output

```
{
    _id: 3,
    name: 'replica-cluster-rs0-0.psmdb.svc.clusterset.local:27017',
    health: 1,
    state: 1,
    stateStr: 'PRIMARY'
  },
  {
    _id: 4,
    name: 'replica-cluster-rs0-1.psmdb.svc.clusterset.local:27017',
    health: 1,
    state: 2,
    stateStr: 'SECONDARY'
  },
  {
    _id: 5,
    name: 'replica-cluster-rs0-2.psmdb.svc.clusterset.local:27017',
    health: 1,
    state: 2,
    stateStr: 'SECONDARY'
  }
```

7. Repeat steps 1-6 for every shard’s replica set in your sharded cluster.

8. Connect to the config server replica set Pod and repeat steps 1-6.

9. Connect to the Replica site and check the replica set configuration

10. Reconfigure your MongoDB clients to connect to the Replica site.

8.3.4.5 Backups with cross-site replication

Before the Operator version 1.19.0 Backups were supported for the primary (managed) cluster only. Now backups can be taken on primary and replica clusters.

Still, backups on cross-site configurations have some specifics.

- Even though you can run backups in unmanaged clusters, you can't run restores on them.
- Even if the backup is started in primary (managed) cluster, most likely it will be taken from a secondary instance, even if such instance is on a separate cluster, because Percona Backup for MongoDB (PBM) automatically assigns lower priority to primary member to avoid affecting the write performance. This can be overwritten with custom PBM configuration.
- PBM configuration is shared across all clusters. The Operator will reconfigure PBM every time it runs a backup, and setting PBM configuration in one cluster will affect other clusters too. For example, setting `backup.configuration.backupOptions.oplogSpanMin` to 2 in a secondary cluster will be applied to primary cluster as well.

8.3.4.6 Splitting replica set across multiple data centers

Splitting the replica set of the database cluster over multiple Kubernetes clusters can be useful to get a fault-tolerant system in which all replicas are in different data centers.

The Operator cannot deploy MongoDB replicas to other data centers, but this solution can be achieved with a number of Operator deployments, equal to the size of your replica set. So, you will need at least 3 Operator instances: one Operator to control the replica set via cross-site replication, and at least two Operators to bootstrap the unmanaged clusters. Each cluster will contain replica set with only one member, and the *Main* site will manage instances from other sites as external nodes. All configuration of the replica set is done manually.

The solution has the following limitations to consider:

- setting it up involves a number of manual operations, and the same applies to scaling such a manually configured replica,
- backups are supported on the *Main* site only, not on the *Replica* sites.

Configuring the *Main* site

You will use the externally reachable URI for each of your replica set instances, manually overwiriting its default local fully-qualified domain name (FQDN) in the Custom Resource manifest. Also you will need including all these host names into TLS certificates. So the first thing needed is the list of these externally reachable names. In the above example we will use the following ones:

- `r1.percona.local:443` URI for the `cluster-name-rs0-0` (1st replica set instance),
- `r2.percona.local:443` for the 2nd replica set instance,
- `r3.percona.local:443` for the 3rd replica set instance.

Following steps will allow you to prepare the *Main* site for cross-site replication, keeping in mind the multiple data centers deployment:

TLS certificates generated by the Operator are not suitable and it's required to [generate certificates manually](#) on the *Main* site before creating a database cluster, with all names from `replsetOverrides` and `externalNodes`.

1. Use [TLS ceritficates manual generation instruction](#) to prepare TLS certificates with the host names from your prepared list.
2. Deploy your [Main site](#) as usual, with these manually generated certificates. Don't forget to turn on [Pods exposure on your Main cluster](#).
3. Now override hostname of the first replica in the replica set configuration by using the `replsets.replsetOverrides` subsection in the Custom Resource options manifest with the externally reachable endpoint from your externally reachable URI list:

   ```
   ...
   unsafeFlags:
     replsetSize: true
   replsets:
   - name: rs0
      size: 1
      replsetOverrides:
        cluster-name-rs0-0:
          host: r1.percona.local:443
      ...
   ```

   The `unsafeFlags.replsetSize` option in the above example is needed to create replica set with less than 3 instances.

   The actual approach to make the URI reachable from the outside of your Kubernetes cluster depends on the exposure type. It is different in case of the [NodePort exposure](#), [Load balancer of the cloud provider](#), etc. Operator won't perform any validation for hostnames. It's user's responsibility to ensure connectivity.

   > **Note**
   >
   > You can also add custom tags to the replset members, just to make their identification easier:
   >
   > ```
   > ...
   > replsetOverrides:
   >   cluster-name-rs0-0:
   >     host: r1.percona.local:443
   >     tags:
   >       team: cloud
   > ...
   > ```

Configuring *Replica* sites

To configure *Replica* sites, you should [deploy your Relica sites](), repeating the following steps for each Kubernetes cluster you are adding:

1. Copy secrets from the *Main* site, rename them according to the cluster name you use on the *Replica* site (if needed), and apply.

    - `cluster1-ssl` (SSL certificates for client connections),
    - `cluster1-ssl-internal` (SSL certificates for replication),
    - `cluster1-secrets` (user credentials),
    - `cluster1-mongodb-encryption-key` (encryption key).

2. Deploy the database cluster on the *Replica* site. Don't forgetting the following:

    a. All *Replica* sites must be deployed with the `unmanaged: true` Custom Resource option. This will stop the Operator in the *Replica* cluster from touching the MongoDB replset configuration. Starting from this moment, only the Operator of the *Main* cluster will be able to modify it.

    b. Backups must be disabled with the `backup.enabaled: false` Custom Resource option.

    c. The `updateStrategy` Custom Resource option must be set to `RollingUpdate` or `OnDelete`.

    d. In order to create a single-instance replica set, you will need to the `unsafeFlags.replsetSize` option to `true` as you did on the *Main* site.

3. Now add the new *Replica* site's Pod **to your *Main* site's** `externalNodes` subsection of the Custom Resource options manifest:

```
replsets:
- name: rs0
  size: 1
  replsetOverrides:
    cluster1-rs0-0:
      host: r1.percona.local:443
  externalNodes:
  - host: r2.percona.local:443
    votes: 1
    priority: 1
```

8.4 Monitor database with Percona Monitoring and Management (PMM)

In this section you will learn how to monitor the health of Percona Server for MongoDB with Percona Monitoring and Management (PMM).

The Operator supports both PMM version 2 and PMM version 3.

It determines which PMM server version you are using based on the authentication method you provide. For PMM 2, the Operator uses API keys for authentication. For PMM 3, it uses service account tokens.

We recommend using the latest PMM 3.

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 needed metrics and sends gathered data to the PMM 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. If you are using PMM server version 2, use a PMM client image compatible with PMM 2. If you are using PMM server version 3, use a PMM client image compatible with PMM 3. Check Percona certified images for the right one.
2. If you specified both authentication methods for PMM server configuration and they have non-empty values, priority goes to PMM 3.
3. For migration from PMM2 to PMM3, see PMM upgrade documentation. Also check the Automatic migration of API keys page.

Install PMM Server

You must have PMM server up and running. You can run PMM Server as a *Docker image*, a *virtual appliance*, or in Kubernetes. Please refer to the official PMM documentation for the installation instructions.

Install PMM Client

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

Configure authentication

PMM3

PMM3 uses service accounts to control access to PMM server components and resources. To authenticate in PMM server, you need a service account token. [Generate a service account and token](). Specify the Admin role for the service account.

> **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.

PMM2

Get the PMM API key from PMM Server. The API key must have the role "Admin". You need this key to authorize PMM Client within PMM Server.

From PMM UI

Generate the PMM API key

From command line

You can query your PMM Server installation for the API Key using `curl` and `jq` utilities. Replace `<login>:<password>@<server_host>` placeholders with your real PMM Server login, password, and hostname in the following command:

```
PMM_SERVER_API_KEY=$(curl --insecure -X POST -H "Content-Type: application/json" -d '{"name":"operator", "role": "Admin"}'
"https://<login>:<password>@<server_host>/graph/api/auth/keys" | jq .key)
```

> **Warning**
>
> The API key is not rotated automatically when it expires. You must manually recreate it and reconfigure the PMM Client.

Create a secret

Now you must pass the credentials to the Operator. To do so, create a Secret object.

1. Create a Secret configuration file. You can use the deploy/secrets.yaml secrets file.

PMM 3

Specify the service account token as the `PMM_SERVER_TOKEN` value in the secrets file:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-cluster-name-secrets
type: Opaque
stringData:
  ....
  PMM_SERVER_TOKEN: ""
```

PMM 2

Specify the API key as the `PMM_SERVER_API_KEY` value in the secrets file:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-cluster-name-secrets
type: Opaque
stringData:
  ....
  PMM_SERVER_API_KEY: ""
```

2. Create the Secrets object using the `deploy/secrets.yaml` file.

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

Expected output

```
secret/my-cluster-name-secrets created
```

Deploy the PMM Client

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

- Set `pmm.enabled`=`true`.
- Specify your PMM Server hostname / an IP address for the `pmm.serverHost` option. The PMM Server IP address should be resolvable and reachable from within your cluster.
- Check that the name of the Secret object that you created earlier is specified in the `secrets.users` field.

```
secrets:
  users: my-cluster-name-secrets
pmm:
  enabled: true
  image: percona/pmm-client:2.44.1-1
  serverHost: monitoring-service
```

2. Apply the changes:

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

3. 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>-rs0-0 -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 MongoDB from the left-hand navigation menu. You land on the **Instances Overview** page.
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 **MongoDB** → **Other dashboards** to see the list of available dashboards that allow you to drill down to the metrics you are interested in.

Enable profiling

Starting from the Operator version 1.12.0, MongoDB operation profiling is disabled by default. To analyze query execution on the PMM Query Analytics dashboard, you should enable profiling explicitly. You can pass options to MongoDB in several ways.

This example shows how to pass the configuration via the `configuration` subsection of the `deploy/cr.yaml` manifest.

```
spec:
  ...
  replsets:
    - name: rs0
      size: 3
      configuration: |
        operationProfiling:
          slowOpThresholdMs: 200
          mode: slowOp
          rateLimit: 100
```

Optionally, you can specify additional parameters for the `pmm-admin add mongodb` command in the `pmm.mongodParams` and `pmm.mongosParams` keys for `mongod` and `mongos` Pods respectively.

Info: Note that the Operator automatically manages common MongoDB Service Monitoring parameters such as username, password, service-name, host, etc. Assigning values to these parameters is not recommended and can negatively affect the functionality of the PMM setup carried out by the Operator.

When done, apply the edited `deploy/cr.yaml` file:

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

Update the secrets file

The `deploy/secrets.yaml` file contains all values for each key/value pair in a convenient plain text format. But the resulting Secrets Objects contains passwords stored as base64-encoded strings. If you want to *update* the password field, you need to encode the new password into the base64 format and pass it to the Secrets Object.

To encode a password or any other parameter, run the following command:

on Linux

```
echo -n "password" | base64 --wrap=0
```

on macOS

```
echo -n "password" | base64
```

For example, to set the new PMM Server token in the `my-cluster-name-secrets` object, do the following:

on Linux

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

on macOS

```
kubectl patch secret/my-cluster-name-secrets -p '{"data":{"PMM_SERVER_TOKEN": '$(echo -n <new-token> | base64)'}}'
```

8.5 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 Distribution for MongoDB Replica Set, Config Servers, and mongos Pods. Just use `sidecars` subsection in the `replsets`, `sharding.configsvrReplSet`, and `sharding.mongos` 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:
  replsets:
    ....
    sidecars:
    - image: busybox
      command: ["/bin/sh"]
      args: ["-c", "while true; do echo echo $(date -u) 'test' >> /dev/null; sleep 5; done"]
      name: rs-sidecar-0
    ....
```

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 my-cluster-name-rs0-0
```

Expected output

```
....
Containers:
....
rs-sidecar-0:
  Container ID:  docker://f0c3437295d0ec819753c581aae174a0b8d062337f80897144eb8148249ba742
  Image:         busybox
  Image ID:      docker-pullable://busybox@sha256:139abcf41943b8bcd4bc5c42ee71ddc9402c7ad69ad9e177b0a9bc4541f14924
  Port:          <none>
  Host Port:     <none>
  Command:
    /bin/sh
  Args:
    -c
    while true; do echo echo $(date -u) 'test' >> /dev/null; sleep 5; done
  State:          Running
    Started:      Thu, 11 Nov 2021 10:38:15 +0300
  Ready:          True
  Restart Count:  0
  Environment:    <none>
  Mounts:
    /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-fbrbn (ro)
....
```

Getting shell access to a sidecar container

You can login to your sidecar container as follows:

```
kubectl exec -it my-cluster-name-rs0-0 -c rs-sidecar-0 -- 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. They allow either dynamically provisioning volumes for sidecar containers or mounting existing volumes.

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.

The following example requests 1G storage with `sidecar-volume-claim` PersistentVolumeClaim, and mounts the correspondent Persistent Volume to the `rs-sidecar-0` container's filesystem under the `/volume0` directory:

```
...
sidecars:
- image: busybox
  command: ["/bin/sh"]
  args: ["-c", "while true; do echo echo $(date -u) 'test' >> /dev/null; sleep 5; done"]
  name: rs-sidecar-0
  volumeMounts:
  - mountPath: /volume0
    name: sidecar-volume-claim
sidecarPVCs:
- apiVersion: v1
  kind: PersistentVolumeClaim
  metadata:
    name: sidecar-volume-claim
  spec:
    resources:
      requests:
        storage: 1Gi
    volumeMode: Filesystem
    accessModes:
      - ReadWriteOnce
```

> **Note**
>
> Sidecar containers for *mongos* Pods have limited Persistent volumes support: `sharding.mongos.sidecarPVCs` option can be used if there is a single mongos in deployment or when `ReadWriteMany/ReadOnlyMany` access modes are used (but these modes are available not in every storage).

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 via the `sidecarVolumes` subsection as follows:

```
...
sidecars:
- image: busybox
  command: ["/bin/sh"]
  args: ["-c", "while true; do echo echo $(date -u) 'test' >> /dev/null; sleep 5; done"]
  name: rs-sidecar-0
  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 `rs-sidecar-0` 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. Secrets are stored with the Kubernetes API and mounted to the container as RAM-stored files.

You can mount a configMap volume via the `sidecarVolumes` subsection as follows:

```
...
sidecars:
- image: busybox
  command: ["/bin/sh"]
  args: ["-c", "while true; do echo echo $(date -u) 'test' >> /dev/null; sleep 5; done"]
  name: rs-sidecar-0
  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 `rs-sidecar-0` container's filesystem under the `/config` directory.

> **Note**
>
> Don't forget you need to create a configMap Object before you can use it.

8.6 Pause/resume Percona Server for MongoDB

There may be external situations when it is needed to shutdown the 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
```

To start the cluster after it was shut down just revert the `spec.pause` key to `false`.

8.7 Configure VolumeAttributesClass for Persistent Volumes

A `VolumeAttributesClass` is a Kubernetes resource that enables you to modify the attributes of an existing Persistent Volume (PV) without recreating it.

It defines a group of storage attributes that you can change on an existing volume. These settings act as a “profile” that you attach to a PersistentVolumeClaim (PVC). To use this feature, the underlying CSI driver must support the Kubernetes `ModifyVolume` API.

Percona Operator for MongoDB supports VolumeAttributesClass through references in the PVC templates you define in the cluster’s Custom Resource (CR).

This page explains when to use VolumeAttributesClass and how to configure it for Percona Operator deployments.

When to use a VolumeAttributesClass?

You can use a VolumeAttributesClass to adjust the storage performance of an existing volume. Typical use cases include:

- Increasing provisioned IOPS
- Adjusting throughput
- Switching performance tiers
- Enabling storage features supported by the CSI backend

Unlike a `StorageClass`, which defines how a volume is **created**, a `VolumeAttributesClass` defines how a volume is **modified** after creation.

Read more about `VolumeAttributesClass` in the [Kubernetes documentation]().

How to configure a VolumeAttributesClass

Requirements

To use `VolumeAttributesClass`, your environment must meet the following requirements:

- Kubernetes version 1.34 and above that supports the `VolumeAttributesClass` API.
- The `VolumeAttributesClass` feature gate is enabled for your Kubernetes cluster.
- A CSI driver that supports `ModifyVolume`. The example of such driver on GKE is `pd.csi.storage.gke.io`. You can check the list of CSI drivers with the following command:

  ```
  kubectl get storageclasses
  ```

  Expected output

  ```
  NAME                    PROVISIONER             RECLAIMPOLICY   VOLUMEBINDINGMODE      ALLOWVOLUMEEXPANSION   AGE
  premium-rwo             pd.csi.storage.gke.io   Delete          WaitForFirstConsumer   true                   5h18m
  standard                kubernetes.io/gce-pd    Delete          Immediate              true                   5h18m
  standard-rwo (default)  pd.csi.storage.gke.io   Delete          WaitForFirstConsumer   true                   5h18m
  ```

- `PersistentVolumes` provisioned through that CSI driver.

Steps

To configure a VolumeAttributesClass, you need to create a `VolumeAttributesClass` resource. The resource defines the storage attributes that you can change on an existing volume.

1. Create a `VolumeAttributesClass` configuration file. For example, `volume-attributes-class.yaml`:

   volume-attributes-class.yaml

```
apiVersion: storage.k8s.io/v1
kind: VolumeAttributesClass
metadata:
  name: silver
driverName: pd.csi.storage.gke.io
parameters:
  provisioned-iops: "3000"
  provisioned-throughput: "50"
```

2. Apply the configuration file to create the `VolumeAttributesClass` resource:

```
kubectl apply -f volume-attributes-class.yaml -n <namespace>
```

3. Check that the `VolumeAttributesClass` is created:

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

Expected output

```
NAME     DRIVERNAME              AGE
silver   pd.csi.storage.gke.io   9m32s
```

4. Reference the `VolumeAttributesClass` in the PVC template in your cluster's Custom Resource (CR). For example, edit the `deploy/cr.yaml` file and add the following:

```
replsets:
- name: rs0
  ...
  volumeSpec:
    persistentVolumeClaim:
      volumeAttributesClassName: silver
      resources:
        requests:
          storage: 3Gi
```

5. Apply the changes to your cluster:

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

Behavior

The Operator applies `VolumeAttributesClass` only to new PVCs when they are created. It does not update existing PVCs automatically. To apply a `VolumeAttributesClass` to an existing PVC, either patch the PVC manually or trigger the Operator to recreate the PVC.

6. Verify that the `VolumeAttributesClass` was applied to the PVC:

```
kubectl get pvc my-cluster-name-rs0-0 -n <namespace> -o yaml
```

Expected output

```
{.text .no-copy}
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-cluster-name-rs0-0
  ...
spec:
 accessModes:
 - ReadWriteOnce
 resources:
   requests:
     storage: 3Gi
 storageClassName: standard-rwo
 volumeAttributesClassName: silver
 volumeMode: Filesystem
```

9 Troubleshooting

9.1 Percona Operator troubleshooting

This section provides information on how to troubleshoot issues when you install Percona Operator for MongoDB.

Make sure you have CLI tool `kubectl` installed to interact with Kubernetes API.

Check connection to Kubernetes cluster

It may happen that `kubectl` you installed locally is not connected to your Kubernetes cluster.

To check connectivity to your Kubernetes API, run the following command:

```
kubectl cluster-info
```

If you see the output similar to the following, it means that `kubectl` is connected to your Kubernetes cluster:

Sample output

```
Kubernetes control plane is running at https://<control-plane-ip>:49475
CoreDNS is running at https://<control-plane-ip>:49475/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
```

If multiple Kubernetes configurations are present in `kubeconfig`, check if you have set the correct context. If the context is wrong, switch it.

Here's how:

1. Check the current context:

```
kubectl config current-context # Get the current Context
```

2. Switch the context :

```
kubectl config use-context <Context-To-Be-Used>
```

3. Run the `kubectl cluster-info` command again to verify that `kubectl` is connected to your Kubernetes cluster.

If you are still running into issues, check with your Kubernetes cluster administrator to resolve the connectivity or configuration issues.

Troubleshoot Operator installation issues

1. Check the Operator logs

```
kubectl logs deploy/<operator-deployment-name>
```

2. Installing the Operator requires specific privileges, such as the ability to create custom resource definitions and other Kubernetes objects.

   To verify that you have the necessary privileges, download and run the following script:

```
bash <(curl -s
https://gist.githubusercontent.com/cshiv/6048bdd0174275b48f633549c69d0844/raw/fd547b783a30b827362ee9f9ec03436f9bc79524/chec
k_priviliges.sh)
```

Sample output

```
Checking privileges to install Percona Operators in kubernetes cluster...
Warning: Unable to check the privileges for resource 'issuers', check if the resource 'issuers' is present in the cluster
Warning: Unable to check the privileges for resource 'certificates', check if the resource 'certificates' is present in the cluster

Warning: Some resources are not found in the kubernetes cluster.Check the Warning messages before you proceed
------------------------------------------------------------------------------------------
GOOD TO INSTALL: Percona Operator for PostgreSQL
https://docs.percona.com/percona-operator-for-postgresql/index.html
------------------------------------------------------------------------------------------
GOOD TO INSTALL: Percona Operator for MySQL based on Percona XtraDB Cluster
https://docs.percona.com/percona-operator-for-mysql/pxc/index.html
------------------------------------------------------------------------------------------
GOOD TO INSTALL: Percona Operator for MongoDB
https://docs.percona.com/percona-operator-for-mongodb/index.html
```

If you have insufficient permissions, the script will show you which ones are missing for installing a particular Operator. In this case, contact the Kubernetes cluster administrator.

3. If you have the necessary privileges but the installation is still failing, review the Kubernetes Events for more details. Keep in mind that Kubernetes Events are retained for only 60 minutes.

```
kubectl get events --sort-by=".lastTimestamp"
```

Events provide good information about affinity issues, resource issues etc.

Troubleshooting database cluster issues

1. The Operator deployment must be in the `Running` state for the database cluster to function properly. Check the Operator Pod for restarts to identify potential issues.

```
kubectl get pod <operator-pod-name>
```

2. Check the status of the database cluster

```
kubectl get psmdb <database-cluster-name>
```

The cluster should typically be in the `Running` state. It may briefly enter the `initializing` state while reconciling changes. If the cluster remains in the `initializing` state for an extended period, investigate further to identify any underlying issues.

Additionally, you can describe the database cluster and search for the information in the `State` and `State Description` fields:

```
kubectl describe psmdb <database-cluster-name>
```

3. Check the Operator logs

```
kubectl logs deploy/<operator-deployment-name>
```

4. Check the events

```
kubectl get events --sort-by=".lastTimestamp"
```

Events can provide information like storage class issues, PVC binding issues etc

5. Check for the PVC, PV. Both of them should be in `Bound` status

```
kubectl get pvc
```

```
kubectl get pv
```

6. Check for logs of database pods / `mongos` pods

```
kubectl logs <database-pod-name>
```

```
kubectl logs <mongos-pod-name>
```

To check logs of `init` containers or other sidecar containers, use the option `-c` with the container name:

```
kubectl logs <mongos-pod-name> -c mongo-init
```

7. Check for error details. Run the `kubectl describe` command:

`bash`
`   kubectl describe <database-pod-name>`

````
```bash
kubectl describe <mongos-pod-name>
```

Check the information in the `Status` section. The `State` and `State Description` fields explain why the Pod reports errors.
````

1. To run commands inside a container, use the `kubectl exec` command:

   ```
   kubectl exec <pod-name> -- <command>
   ```

   If you need an interactive shell to run multiple commands, use the `-it` flag for an interactive terminal:

   ```
   kubectl exec -it <pod-name> -- sh
   ```

2. If the pods are not running, it may not be possible to execute commands or open an interactive shell. In such cases, consider using a `sleep-forever` script to prevent the containers from restarting repeatedly.

See the [Avoid the restart-on-fail loop for Percona Server for MongoDB containers]() section for steps.

9.2 Initial troubleshooting

Percona Operator for MongoDB uses Custom Resources to manage options for the various components of the cluster.

- `PerconaServerMongoDB` Custom Resource with Percona Server for MongoDB options (it has handy `psmdb` shortname also),
- `PerconaServerMongoDBBackup` and `PerconaServerMongoDBRestore` Custom Resources contain options for Percona Backup for MongoDB used to backup Percona Server for MongoDB and to restore it from backups (`psmdb-backup` and `psmdb-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 psmdb
```

Expected output

```
NAME              ENDPOINT                                       STATUS   AGE
my-cluster-name   my-cluster-name-mongos.default.svc.cluster.local   ready    5m26s
```

The Custom Resource should have `Ready` status.

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

```
perconaservermongodbbackups      psmdb-backup    psmdb.percona.com/v1                true         PerconaServerMongoDBBackup
perconaservermongodbrestores     psmdb-restore   psmdb.percona.com/v1                true         PerconaServerMongoDBRestore
perconaservermongodbs            psmdb           psmdb.percona.com/v1                true         PerconaServerMongoDB
```

Check the Pods

If Custom Resource is not getting `Ready` status, it makes sense to check individual Pods. You can do it as follows:

```
kubectl get pods
```

Expected output

```
NAME                                                READY   STATUS    RESTARTS   AGE
my-cluster-name-cfg-0                               2/2     Running   0          11m
my-cluster-name-cfg-1                               2/2     Running   1          10m
my-cluster-name-cfg-2                               2/2     Running   1          9m
my-cluster-name-mongos-0                            1/1     Running   0          11m
my-cluster-name-mongos-1                            1/1     Running   0          11m
my-cluster-name-mongos-2                            1/1     Running   0          11m
my-cluster-name-rs0-0                               2/2     Running   0          11m
my-cluster-name-rs0-1                               2/2     Running   0          10m
my-cluster-name-rs0-2                               2/2     Running   0          9m
percona-server-mongodb-operator-665cd69f9b-xg5dl   1/1     Running   0          37m
```

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, `my-cluster-name-rs0-0` Pod 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 my-cluster-name-rs0-0
```

Expected output

```
...
Name:         my-cluster-name-rs0-0
Namespace:    default
...
Controlled By:  StatefulSet/my-cluster-name-rs0
Init Containers:
 mongo-init:
...
Containers:
 mongod:
...
   Restart Count:  0
   Limits:
     cpu:     300m
     memory:  500M
   Requests:
     cpu:      300m
     memory:   500M
   Liveness:   exec [/opt/percona/mongodb-healthcheck k8s liveness --ssl --sslInsecure --sslCAFile /etc/mongodb-ssl/ca.crt --sslPEMKeyFile
/tmp/tls.pem --startupDelaySeconds 7200] delay=60s timeout=10s period=30s #success=1 #failure=4
   Readiness:  tcp-socket :27017 delay=10s timeout=2s period=3s #success=1 #failure=8
   Environment Variables from:
     internal-my-cluster-name-users  Secret  Optional: false
   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.

Check logs

Logs help you pinpoint startup failures, crash loops, configuration issues, and performance problems. They include timestamps, error messages, and component-specific context that you won’t see in Pod status alone. See the Check logs section for more information about logs and how to check them.

9.3 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 `mongod` of the `my-cluster-name-rs0-0` Pod.

- Run `date` command:

  ```
  kubectl exec -ti my-cluster-name-rs0-0 -c mongod -- date
  ```

  Expected output

  ```
  Thu Nov 24 10:01:17 UTC 2022
  ```

  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 my-cluster-name-rs0-0 -c mongod -- 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/mongo/mongod.log` file to a terminal:

  ```
  kubectl exec -ti my-cluster-name-rs0-0 -c mongod -- cat /var/log/mongo/mongod.log
  ```

- Similarly, you can open an Interactive terminal, execute a pair of commands in the container, and exit:

  ```
  kubectl exec -ti my-cluster-name-rs0-0 -c mongod -- bash
  ```

  Expected output

  ```
  [mongodb@my-cluster-name-rs0-0 db]$
  ```

  ```
  cat /etc/hostname
  ```

  Expected output

  ```
  my-cluster-name-rs0-0
  ```

  ```
  ls /var/log/mongo/mongod.log
  ls /var/log/mongo/mongod.log
  ```

  ```
  exit
  ```

Avoid the restart-on-fail loop for Percona Server for MongoDB containers

The restart-on-fail loop takes place when the container entry point fails (e.g. `mongod` 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 MongoDB containers into the “infinite sleep” *without* starting mongod. This behavior of the container entry point is triggered by the presence of the `/data/db/sleep-forever` file. The feature is available for both replica set and confg server Pods.

For example, you can do it for the `mongod` container of an appropriate Percona Server for MongoDB Pod as follows:

```
kubectl exec -it my-cluster-name-cfg-0 -c mongod -- sh -c 'touch /data/db/sleep-forever'
```

If `mongod` container can't start, you can use `backup-agent` container instead:

```
kubectl exec -it my-cluster-name-cfg-0 -c backup-agent -- sh -c 'touch /data/db/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).

9.4 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 `my-cluster-name-rs0-0` Pod.

- Check logs of the `mongod` container:

```
kubectl logs my-cluster-name-rs0-0 -c mongod
```

- Check logs of the `pmm-client` container:

```
kubectl logs my-cluster-name-rs0-0 -c pmm-client
```

- Filter logs of the `mongod` container which are not older than 600 seconds:

```
kubectl logs my-cluster-name-rs0-0 -c mongod --since=600s
```

- Check logs of a previous instantiation of the `mongod` container, if any:

```
kubectl logs my-cluster-name-rs0-0 -c mongod --previous
```

- Check logs of the `mongod` container, parsing the output with jq JSON processor:

```
kubectl logs my-cluster-name-rs0-0 -c mongod -f | jq -R 'fromjson?'
```

Changing logs representation

You can also change the representation of logs: either use structured representation, which produces a parsing-friendly JSON, or use traditional console-friendly logging with specific level. Changing representation of logs is possible by editing the `deploy/operator.yml` file, which sets the following environment variables with self-speaking names and values:

```
env:
    ...
    name: LOG_STRUCTURED
    value: 'false'
    name: LOG_LEVEL
    value: INFO
    ...
```

9.5 Special debug images

For the cases when Pods are failing for some reason or just show abnormal behavior, the Operator can be used with a special *debug image* of the Percona Server for MongoDB, which has the following specifics:

- it avoids restarting on fail,
- it contains additional tools useful for debugging (sudo, telnet, gdb, mongodb-debuginfo package, etc.),
- extra verbosity is added to the mongodb daemon.

Images are available for Percona server for MongoDB versions 5.0 and 6.0, not for 7.0.

Particularly, using this image is useful if the container entry point fails (`mongod` 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.

To use the debug image instead of the normal one, set the following image name for the `image` key in the `deploy/cr.yaml` configuration file:

`percona/percona-server-mongodb:6.0.27-21-debug`

The Pod should be restarted to get the new image.

> **Note**
>
> When the Pod is continuously restarting, you may have to delete it to apply image changes.

9.6 Troubleshooting storage

Storage resizing

Use this guidance if storage resizing does not behave as expected.

Storage does not scale when you expect it

- Check namespace quotas; they can block PVC expansion even if the storage class has capacity.
- Check quotas or limits set by your cloud provider.
- Look for storage or CSI-level errors (missing storage, driver issues). Fix the issue and let the Operator reconcile.

Max size is not respected

The Operator always grows storage by the full `growthStep` value. If the current size plus `growthStep` exceeds the `maxSize` value, the Operator caps the resize at `maxSize` instead. To avoid large jumps or unexpected caps, either set the `growthStep` value to the increment you want the storage to resize or raise the `maxSize` limit.

Scaling doesn’t happen despite it hasn’t reached the maximum.

If storage is not scaling, check if any PVC has reached the max size. Thresholds and limits apply to all PVCs, so when one hits the limit, scaling stops for all. Adjust PVC sizes if needed.

PVC resize is one-way

PVC resizing in Kubernetes is one-way. You cannot shrink a PVC to roll back to the old size without recreating it. Recreating PVCs can cause data loss, so make sure you have a backup strategy in place. See About backups.

10 HOWTOs

10.1 Install Percona Server for MongoDB with customized parameters

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

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

`kubectl`

To customize the configuration, do the following:

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

```
git clone -b v1.22.0 https://github.com/percona/percona-server-mongodb-operator
```

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

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

Helm

To install Percona Server for MongoDB with custom parameters, use the following command:

```
helm install --set key=value
```

You can pass any of the Operator's Custom Resource options as a `--set key=value[,key=value]` argument.

The following example deploys a Percona Server for MongoDB Cluster in the `psmdb` namespace, with disabled backups and 20 Gi storage:

Command line

```
helm install my-db percona/psmdb-db --version 1.22.0 --namespace psmdb \
  --set "replsets.rs0.name=rs0" --set "replsets.rs0.size=3" \
  --set "replsets.rs0.volumeSpec.pvc.resources.requests.storage=20Gi" \
  --set backup.enabled=false --set sharding.enabled=false
```

YAML file

You can specify customized options in a YAML file instead of using separate command line parameters. The resulting file similar to the following example looks as follows:

values.yaml

```
allowUnsafeConfigurations: true
sharding:
  enabled: false
replsets:
- name: rs0
  size: 3
  volumeSpec:
    pvc:
      resources:
        requests:
          storage: 2Gi
backup:
  enabled: false
```

Apply the resulting YAML file as follows:

```
helm install my-db percona/psmdb-db --namespace psmdb -f values.yaml
```

Configure ports for MongoDB cluster components

By default, the Operator starts Percona Server for MongoDB with the default port `27017` for all cluster components: `mongod`, `mongos` and `configsvrReplSet` Pods. Starting with version 1.20.0, you can start a new cluster with custom ports for all components or for a specific one.

Here's how to do it.

kubectl

1. Edit the `deploy/cr.yaml` file and specify the following configuration:

```
spec:
  ...
  replsets:
    - name: rs0
      configuration: |
        net:
          port: 27018
  sharding:
    configsvrReplSet:
      configuration: |
        net:
          port: 27019
    mongos:
      configuration: |
        net:
          port: 27017
```

2. Apply the `deploy/cr.yaml` to deploy Percona Server for MongoDB:

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

Helm

1. Create a yaml file with the desired configuration. For example, `values.yaml`:

values.yaml

```
replsets:
  rs0:
    name: rs0
    configuration: |
      net:
        port: 27018
sharding:
  configsvrReplSet:
    configuration: |
      net:
        port: 27019
  mongos:
    configuration: |
      net:
        port: 27017
```

2. Install Percona Server for MongoDB with the specified configuration:

```
helm install my-db percona/psmdb-db --namespace psmdb -f values.yaml
```

10.2 How to integrate Percona Operator for MongoDB with OpenLDAP

LDAP services provided by software like OpenLDAP, Microsoft Active Directory, etc. are widely used by enterprises to control information about users, systems, networks, services and applications and the corresponding access rights for the authentication/authorization process in a centralized way.

The following guide covers a simple integration of the already-installed OpenLDAP server with Percona Distribution for MongoDB and the Operator. You can know more about LDAP concepts and LDIF files used to configure it, and find how to install and configure OpenLDAP in the official OpenLDAP and Percona Server for MongoDB documentation.

The OpenLDAP side

You can add needed OpenLDAP settings will the following LDIF portions:

```
0-percona-ous.ldif: |-
  dn: ou=perconadba,dc=ldap,dc=local
  objectClass: organizationalUnit
  ou: perconadba
1-percona-users.ldif: |-
  dn: uid=percona,ou=perconadba,dc=ldap,dc=local
  objectClass: top
  objectClass: account
  objectClass: posixAccount
  objectClass: shadowAccount
  cn: percona
  uid: percona
  uidNumber: 1100
  gidNumber: 100
  homeDirectory: /home/percona
  loginShell: /bin/bash
  gecos: percona
  userPassword: {crypt}x
  shadowLastChange: -1
  shadowMax: -1
  shadowWarning: -1
2-group-cn.ldif: |-
  dn: cn=admin,ou=perconadba,dc=ldap,dc=local
  cn: admin
  objectClass: groupOfUniqueNames
  objectClass: top
  ou: perconadba
  uniqueMember: uid=percona,ou=perconadba,dc=ldap,dc=local
```

Also a read-only user should be created for the database-issued user lookups. If everything is done correctly, the following command should work, resetting the percona user password:

```
ldappasswd -s percona -D "cn=admin,dc=ldap,dc=local" -w password -x "uid=percona,ou=perconadba,dc=ldap,dc=local"
```

> **Note**
>
> If you are not sure about the approach to make references between user and group objects, OpenDAP overlays provide one of the possible ways to go.

The MongoDB and Operator side

The following steps will look different depending on whether sharding is on (the default behavior) or off.

If sharding is off

In order to get MongoDB connected with OpenLDAP in case of a a non-sharded (ReplicaSet) MongoDB cluster we need to configure two things:

- Mongod
- Internal mongodb role

Create configuration Secrets for mongod:

my_mongod.conf

```
security:
  authorization: "enabled"
  ldap:
    authz:
      queryTemplate: '{USER}?memberOf?base'
    servers: "openldap"
    transportSecurity: none
    bind:
      queryUser: "cn=readonly,dc=ldap,dc=local"
      queryPassword: "password"
    userToDNMapping:
      '[
          {
            match : "(.+)",
            ldapQuery: "OU=perconadba,DC=ldap,DC=local??sub?(uid={0})"
          }
    ]'
setParameter:
  authenticationMechanisms: 'PLAIN,SCRAM-SHA-1'
```

Note

This fragment provides mongod with LDAP-specific parameters, such as FQDN of the LDAP server (`server`), explicit lookup user, domain rules, etc.

Put the snippet on you local machine and create a Kubernetes Secret object named based on your MongoDB cluster name:

```
kubectl create secret generic <your_cluster_name>-rs0-mongod --from-file=mongod.conf=my_mongod.conf
```

Next step is to start the MongoDB cluster up as it’s described in Install Percona server for MongoDB on Kubernetes. On successful completion of the steps from this doc, we are to proceed with setting the roles for the ‘external’ (managed by LDAP) user inside the MongoDB. For this, log into MongoDB as administrator:

```
mongo "mongodb+srv://userAdmin:<userAdmin_password>@<your_cluster_name>-rs0.<your_namespace>.svc.cluster.local/admin?
replicaSet=rs0&ssl=false"
```

When logged in, execute the following:

```
mongos> db.getSiblingDB("admin").createRole(
{
 role: "cn=admin,ou=perconadba,dc=ldap,dc=local",
 privileges: [],
 roles : [
   {
     "role" : "readAnyDatabase",
     "db" : "admin"
   },
   {
     "role" : "dbAdminAnyDatabase",
     "db" : "admin"
   },
   {
     "role" : "clusterMonitor",
     "db" : "admin"
   },
   {
     "role" : "readWriteAnyDatabase",
     "db" : "admin"
   },
   {
     "role" : "restore",
     "db" : "admin"
   },
   {
     "role" : "backup",
     "db" : "admin"
   }
 ],
}
)
```

> **Note**
>
> Extra roles listed in the above example are just to show more than one possible variant.

Now the new `percona` user created inside OpenLDAP is able to login to MongoDB as administrator. Verify whether the user role has been identified correctly with the following command:

```
mongo --username percona --password 'percona' --authenticationMechanism 'PLAIN' --authenticationDatabase '$external' --host
<mongodb-rs-endpoint> --port 27017
```

When logged in, execute the following:

```
mongos> db.runCommand({connectionStatus:1})
```

The output should be like follows:

```
{
 "authInfo" : {
   "authenticatedUsers" : [
     {
       "user" : "percona",
       "db" : "$external"
     }
   ],
   "authenticatedUserRoles" : [
     {
       "role" : "restore",
       "db" : "admin"
     },
     {
       "role" : "readAnyDatabase",
       "db" : "admin"
     },
     {
       "role" : "clusterMonitor",
       "db" : "admin"
     },
     {
       "role" : "dbAdminAnyDatabase",
       "db" : "admin"
     },
     {
       "role" : "backup",
       "db" : "admin"
     },
     {
       "role" : "cn=admin,ou=perconadba,dc=ldap,dc=local",
       "db" : "admin"
     },
     {
       "role" : "readWriteAnyDatabase",
       "db" : "admin"
     }
   ]
 },
 "ok" : 1,
 "$clusterTime" : {
   "clusterTime" : Timestamp(1663067287, 4),
   "signature" : {
     "hash" : BinData(0,"ZaLGSVj4ZwZrngXZSOqXB5rx+oo="),
     "keyId" : NumberLong("7142816031004688408")
   }
 },
 "operationTime" : Timestamp(1663067287, 4)
}
mongos>
```

If sharding is on

In order to get MongoDB connected with OpenLDAP in this case we need to configure three things:

- Mongod
- Internal mongodb role
- Mongos

Both the routing interface (mongos) and the configuration ReplicaSet (mongod) have to be configured to make the LDAP server a part of the Authentication/Authorization chain.

> **Note**
>
> mongos is just a router between shards and underlying database instances, and configuration ReplicaSet is responsible for keeping information about database users and roles. Thus, the router can perform only authentication, while authorization is the responsibility of the configuration ReplicaSet.

Create configuration Secrets for the router and the configuration ReplicaSet respectively.

Secret for the router should look as follows:

my_mongos.conf

```
security:
  ldap:
    servers: "openldap"
    transportSecurity: none
    bind:
      queryUser: "cn=readonly,dc=ldap,dc=local"
      queryPassword: "password"
    userToDNMapping:
      '[
          {
            match : "(.+)",
            ldapQuery: "OU=perconadba,DC=ldap,DC=local??sub?(uid={0})"
          }
    ]'
setParameter:
  authenticationMechanisms: 'PLAIN,SCRAM-SHA-1'
```

Put the snippet on you local machine and create a Kubernetes Secret object named based on [your MongoDB cluster name](#):

```
kubectl create secret generic <your_cluster_name>-mongos --from-file=mongos.conf=my_mongos.conf
```

Secret for the configuration ReplicaSet should look as follows:

my_mongod.conf

```
security:
  authorization: "enabled"
  ldap:
    authz:
      queryTemplate: '{USER}?memberOf?base'
    servers: "openldap"
    transportSecurity: none
    bind:
      queryUser: "cn=readonly,dc=ldap,dc=local"
      queryPassword: "password"
    userToDNMapping:
      '[
          {
            match : "(.+)",
            ldapQuery: "OU=perconadba,DC=ldap,DC=local??sub?(uid={0})"
          }
    ]'
setParameter:
  authenticationMechanisms: 'PLAIN,SCRAM-SHA-1'
```

Put the snippet on you local machine and create a Kubernetes Secret object named based on [your MongoDB cluster name](#):

```
kubectl create secret generic <your_cluster_name>-cfg-mongod --from-file=mongod.conf=my_mongod.conf
```

Both files are pretty much the same except the `authz` subsection, which is only present for the configuration ReplicaSet.

Next step is to start the MongoDB cluster up as it's described in [Install Percona server for MongoDB on Kubernetes](). On successful completion of the steps from this doc, we are to proceed with setting the roles for the 'external' (managed by LDAP) user inside the MongoDB. For this, log into MongoDB as administrator:

```
mongo "mongodb://userAdmin:<userAdmin_password>@<your_cluster_name>-mongos.<your_namespace>.svc.cluster.local/admin?ssl=false"
```

When logged in, execute the following:

```
mongos> db.getSiblingDB("admin").createRole(
{
 role: "cn=admin,ou=perconadba,dc=ldap,dc=local",
 privileges: [],
 roles : [
   {
     "role" : "readAnyDatabase",
     "db" : "admin"
   },
   {
     "role" : "dbAdminAnyDatabase",
     "db" : "admin"
   },
   {
     "role" : "clusterMonitor",
     "db" : "admin"
   },
   {
     "role" : "readWriteAnyDatabase",
     "db" : "admin"
   },
   {
     "role" : "restore",
     "db" : "admin"
   },
   {
     "role" : "backup",
     "db" : "admin"
   }
 ],
}
)
```

> **Note**
>
> Extra roles listed in the above example are just to show more than one possible variant.

Now the new `percona` user created inside OpenLDAP is able to login to MongoDB as administrator. Verify whether the user role has been identified correctly with the following command:

```
mongo --username percona --password 'percona' --authenticationMechanism 'PLAIN' --authenticationDatabase '$external' --host
<your_cluster_name>-mongos --port 27017
```

When logged in, execute the following:

```
mongos> db.runCommand({connectionStatus:1})
```

The output should be like follows:

```
{
  "authInfo" : {
    "authenticatedUsers" : [
      {
        "user" : "percona",
        "db" : "$external"
      }
    ],
    "authenticatedUserRoles" : [
      {
        "role" : "restore",
        "db" : "admin"
      },
      {
        "role" : "readAnyDatabase",
        "db" : "admin"
      },
      {
        "role" : "clusterMonitor",
        "db" : "admin"
      },
      {
        "role" : "dbAdminAnyDatabase",
        "db" : "admin"
      },
      {
        "role" : "backup",
        "db" : "admin"
      },
      {
        "role" : "cn=admin,ou=perconadba,dc=ldap,dc=local",
        "db" : "admin"
      },
      {
        "role" : "readWriteAnyDatabase",
        "db" : "admin"
      }
    ]
  },
  "ok" : 1,
  "$clusterTime" : {
    "clusterTime" : Timestamp(1663067287, 4),
    "signature" : {
      "hash" : BinData(0,"ZaLGSVj4ZwZrngXZSOqXB5rx+oo="),
      "keyId" : NumberLong("7142816031004688408")
    }
  },
  "operationTime" : Timestamp(1663067287, 4)
}
mongos>
```

Using LDAP over TLS connection

LDAP over TLS allows you to use Transport Layer Security, encrypting your communication between MongoDB and OpenLDAP server.

Here are the needed modifications to The MongoDB and Operator side subsection which will enable it:

1. First, create a secret that contains the SSL certificate to connect to LDAP. The following example creates it from the file with CA certificate (the one you use in `/etc/openldap/ldap.conf`), naming the new secret `my-ldap-secret`:

   ```
   kubectl create secret generic my-ldap-secret --from-file=ca.crt=ldap-ca.pem
   ```

2. Set the `secrets.ldapSecret` Custom Resource option to the name of your newly created secret. Your modified `deploy/cr.yaml` may look as follows:

   ```
   ...
     secrets:
       ...
       ldapSecret: my-ldap-secret
   ```

3. It is also necessary to change the value of transportSecurity to `tls` in mongod and mongos configurations. The configuration is similar to one described at the The MongoDB and Operator side subsection:

Changed mongod configuration should look as follows:

````
``` yaml title="my_mongod.conf"  hl_lines="7"
security:
  authorization: "enabled"
  ldap:
    authz:
      queryTemplate: '{USER}?memberOf?base'
    servers: "openldap"
    transportSecurity: tls
    bind:
      queryUser: "cn=readonly,dc=ldap,dc=local"
      queryPassword: "password"
    userToDNMapping:
      '[
          {
            match : "(.+)",
            ldapQuery: "OU=perconadba,DC=ldap,DC=local??sub?(uid={0})"
          }
   ]'
setParameter:
  authenticationMechanisms: 'PLAIN,SCRAM-SHA-1'
```

If **sharding is on**, you will also need to change mongos configuration:

```yaml title="my_mongos.conf" hl_lines="4"
security:
  ldap:
    servers: "openldap"
    transportSecurity: tls
    bind:
      queryUser: "cn=readonly,dc=ldap,dc=local"
      queryPassword: "password"
    userToDNMapping:
      '[
          {
            match : "(.+)",
            ldapQuery: "OU=perconadba,DC=ldap,DC=local??sub?(uid={0})"
          }
    ]'
setParameter:
  authenticationMechanisms: 'PLAIN,SCRAM-SHA-1'
```
````

10.3 Use Docker images from a custom registry

Using images from a private Docker registry may required for privacy, security or other reasons. In these cases, Percona Operator for MongoDB allows the use of a custom registry This following example of the Operator deployed in the OpenShift environment demonstrates the process:

1. Log into the OpenShift and create a project.

```
oc login
```

Expected output

```
Authentication required for https://192.168.1.100:8443 (openshift)
Username: admin
Password:
Login successful.
```

```
oc new-project psmdb
```

Expected output

```
Now using project "psmdb" on server "https://192.168.1.100:8443".
```

2. You need obtain the following objects to configure your custom registry access:

- A user token
- the registry IP address

You can view the token with the following command:

```
oc whoami -t
```

Expected output

```
ADO8CqCDappWR4hxjfDqwijEHei31yXAvWg61Jg210s
```

The following command returns the registry IP address:

```
kubectl get services/docker-registry -n default
```

Expected output

```
NAME              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
docker-registry   ClusterIP   172.30.162.173   <none>        5000/TCP   1d
```

3. Use the user token and the registry IP address to login to the registry:

```
docker login -u admin -p ADO8CqCDappWR4hxjfDqwijEHei31yXAvWg61Jg210s 172.30.162.173:5000
```

Expected output

```
Login Succeeded
```

4. Use the Docker commands to pull the needed image by its SHA digest:

```
docker pull docker.io/perconalab/percona-server-
mongodb@sha256:991d6049059e5eb1a74981290d829a5fb4ab0554993748fde1e67b2f46f26bf0
```

Expected output

```
Trying to pull repository docker.io/perconalab/percona-server-mongodb ...
sha256:991d6049059e5eb1a74981290d829a5fb4ab0554993748fde1e67b2f46f26bf0: Pulling from docker.io/perconalab/percona-server-mongodb
Digest: sha256:991d6049059e5eb1a74981290d829a5fb4ab0554993748fde1e67b2f46f26bf0
Status: Image is up to date for docker.io/perconalab/percona-server-
mongodb@sha256:991d6049059e5eb1a74981290d829a5fb4ab0554993748fde1e67b2f46f26bf0
```

You can find correct names and SHA digests in the current list of the Operator-related images officially certified by Percona.

5. The following method can push an image to the custom registry for the example OpenShift `psmdb` project:

```
docker tag \
    docker.io/perconalab/percona-server-mongodb@sha256:991d6049059e5eb1a74981290d829a5fb4ab0554993748fde1e67b2f46f26bf0 \
    172.30.162.173:5000/psmdb/percona-server-mongodb:7.0.30-16
docker push 172.30.162.173:5000/psmdb/percona-server-mongodb:7.0.30-16
```

6. Verify the image is available in the OpenShift registry with the following command:

```
oc get is
```

Expected output

```
NAME                          DOCKER REPO                                                    TAGS            UPDATED
percona-server-mongodb        docker-registry.default.svc:5000/psmdb/percona-server-mongodb  7.0.30-16  2 hours ago
```

7. When the custom registry image is available, edit the the `image:` option in `deploy/operator.yaml` configuration file with a Docker Repo + Tag string (it should look like `docker-registry.default.svc:5000/psmdb/percona-server-mongodb:7.0.30-16`)

Note

If the registry requires authentication, you can specify the `imagePullSecrets` option for all images.

8. Repeat steps 3-5 for other images, and update corresponding options in the `deploy/cr.yaml` file.

Note

Don't forget to set upgradeoptions.apply option to `Disabled`. Otherwise Smart Upgrade functionality will try using the image recommended by the Version Service instead of the custom one.

9. Now follow the standard Percona Operator for MongoDB installation instruction.

10.4 Creating a private S3-compatible cloud for backups

As it is mentioned in backups, any cloud storage which implements the S3 API can be used for backups. The one way to setup and implement the S3 API storage on Kubernetes or OpenShift is Minio - the S3-compatible object storage server deployed via Docker on your own infrastructure.

Setting up Minio to be used with Percona Operator for MongoDB backups involves the following steps:

1. Install Minio in your Kubernetes or OpenShift environment and create the correspondent Kubernetes Service as follows:

   ```
   helm install \
     --name minio-service \
     --version 8.0.5 \
     --set accessKey=some-access-key \
     --set secretKey=some-secret-key \
     --set service.type=ClusterIP \
     --set configPath=/tmp/.minio/ \
     --set persistence.size=2G \
     --set environment.MINIO_REGION=us-east-1 \
     stable/minio
   ```

   Don't forget to substitute default `some-access-key` and `some-secret-key` strings in this command with actual unique key values. The values can be used later for access control. The `storageClass` option is needed if you are using the special Kubernetes Storage Class for backups. Otherwise, this setting may be omitted. You may also notice the `MINIO_REGION` value which is may not be used within a private cloud. Use the same region value here and on later steps (`us-east-1` is a good default choice).

2. Create an S3 bucket for backups:

   ```
   kubectl run -i --rm aws-cli --image=perconalab/awscli --restart=Never -- \
     bash -c 'AWS_ACCESS_KEY_ID=some-access-key \
     AWS_SECRET_ACCESS_KEY=some-secret-key \
     AWS_DEFAULT_REGION=us-east-1 \
     /usr/bin/aws \
     --endpoint-url http://minio-service:9000 \
     s3 mb s3://operator-testing'
   ```

   This command creates the bucket named `operator-testing` with the selected access and secret keys (substitute `some-access-key` and `some-secret-key` with the values used on the previous step).

3. Now edit the backup section of the deploy/cr.yaml file to set proper values for the `bucket` (the S3 bucket for backups created on the previous step), `region`, `credentialsSecret` and the `endpointUrl` (which should point to the previously created Minio Service).

   ```
   ...
   backup:
     enabled: true
     version: 0.3.0
     ...
     storages:
       minio:
         type: s3
         s3:
           bucket: operator-testing
           region: us-east-1
           credentialsSecret: my-cluster-name-backup-minio
           endpointUrl: http://minio-service:9000
     ...
   ```

   The option which should be specially mentioned is `credentialsSecret` which is a Kubernetes secret for backups. Sample backup-s3.yaml can be used to create this secret object. Check that the object contains the proper `name` value and is equal to the one specified for `credentialsSecret`, i.e. `my-cluster-name-backup-minio` in the backup to Minio example, and also contains the proper `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` keys. After you have finished editing the file, the secrets object are created or updated when you run the following command:

   ```
   kubectl apply -f deploy/backup-s3.yaml
   ```

4. When the setup process is completed, making the backup is based on a script. Following example illustrates how to make an on-demand backup:

```
kubectl run -it --rm pbmctl --image=percona/percona-server-mongodb-operator:0.3.0-backup-pbmctl --restart=Never -- \
   run backup \
   --server-address=<cluster-name>-backup-coordinator:10001 \
   --storage <storage> \
   --compression-algorithm=gzip \
   --description=my-backup
```

Don't forget to specify the name of your cluster instead of the `<cluster-name>` part of the Backup Coordinator URL (the cluster name is specified in the deploy/cr.yaml file). Also substitute `<storage>` with the actual storage name located in a subsection inside of the `backups` in the deploy/cr.yaml file. In the earlier example this value is `minio`.

5. To restore a previously saved backup you must specify the backup name. With the proper Backup Coordinator URL and storage name, you can obtain a list of the available backups:

```
kubectl run -it --rm pbmctl --image=percona/percona-server-mongodb-operator:0.3.0-backup-pbmctl --restart=Never -- list
backups --server-address=<cluster-name>-backup-coordinator:10001
```

Now, restore the backup, using backup name instead of the `backup-name` parameter:

```
kubectl run -it --rm pbmctl --image=percona/percona-server-mongodb-operator:0.3.0-backup-pbmctl --restart=Never -- \
   run restore \
   --server-address=<cluster-name>-backup-coordinator:10001 \
   --storage <storage> \
   backup-name
```

10.5 How to use backups to move the external database to Kubernetes

The Operator allows restoring a backup not only on the Kubernetes cluster where it was made, but also on any Kubernetes-based environment with the installed Operator, and the backup/restore tool actually used by the Operator is the Percona Backup for MongoDB. That makes it possible to *move* external MongoDB Cluster to Kubernetes with Percona Backup for MongoDB.

> **Note**
>
> There are other scenarios for migrating MongoDB database to Kubernetes as well. For example, this blogpost covers migration based on the regular MongoDB replication capabilities.

Backups can be stored either locally, or remotely (on Amazon S3 or S3-compatible storage, or on Azure Blob Storage). S3-compatible storage to be used for backups.

1. Make sure the following prerequisite requirements are satisfied within your setup:
   - Percona Backup for MongoDB packages are installed on the replica set nodes of the source cluster following the official installation instructions, and the authentication of the pbm-agent is configured to allow it accessing your database.
   - The Operator and the *destination* cluster should be installed in the Kubernetes-based environment. For simplicity, it's reasonable to have the same topology of the *source* and *destination* clusters, although Percona Backup for MongoDB allows replset-remapping as well.
2. Configure the cloud storage for backups on your *source* cluster following the official guide. For example, using the Amazon S3 storage can be configured with the following YAML file:

   pbm_config.yaml

   ```
   type: s3
   s3:
     region: us-west-2
     bucket: pbm-test-bucket
     credentials:
       access-key-id: <your-access-key-id-here>
       secret-access-key: <your-secret-key-here>
   ```

   After putting all needed details into the file (`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, the S3 bucket and region in the above example), provide the config file to the pbm-agent on all nodes as follows:

   ```
   pbm config --file pbm_config.yaml
   ```

3. Start the pbm-agent:

   ```
   sudo systemctl start pbm-agent
   ```

4. Now you can make backup as follows:

   ```
   pbm backup --wait
   ```

   The command output will contain the *backup name*, which you will further use to restore the backup:

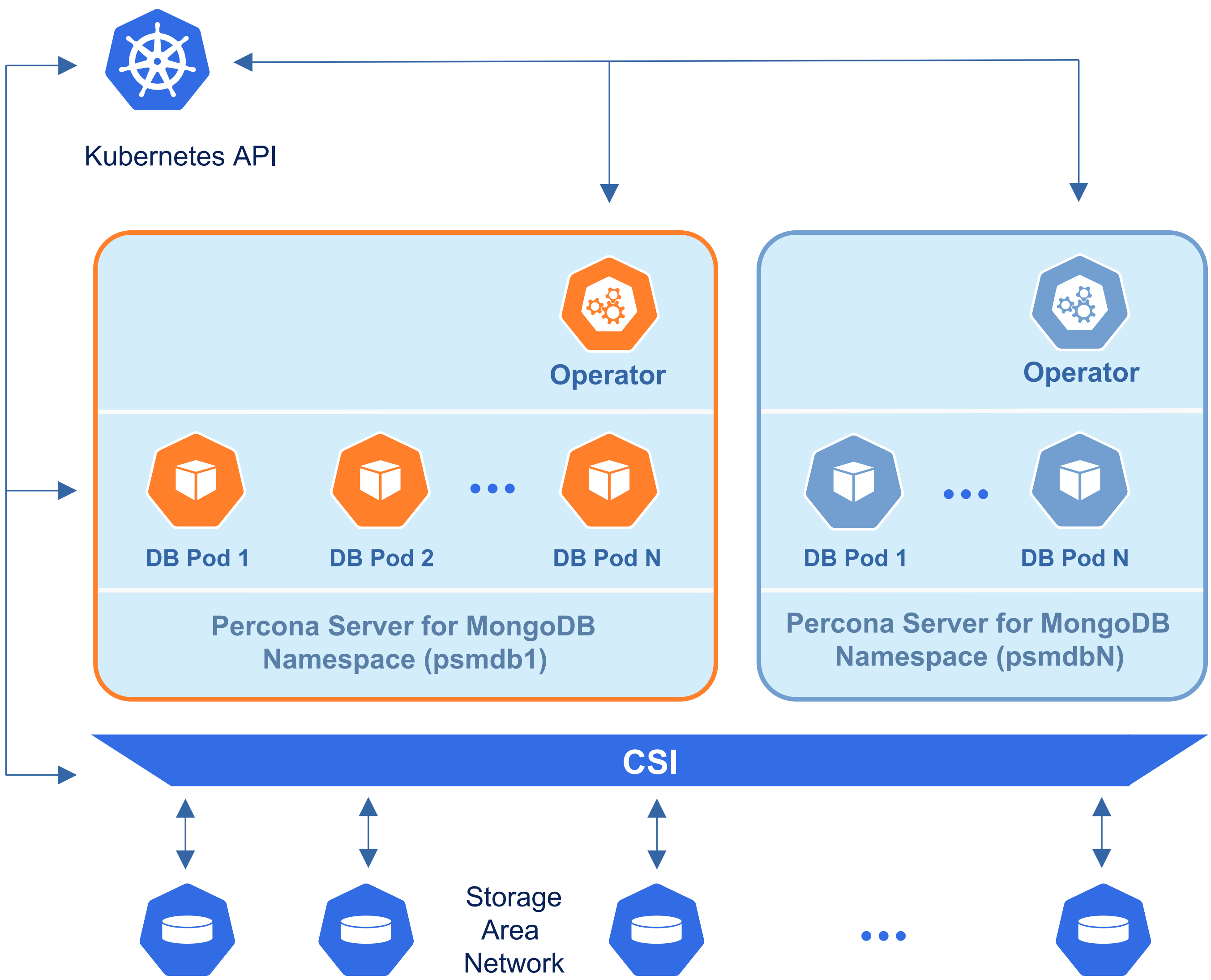
   ```
   Starting backup '2022-06-15T08:18:44Z'....
   Waiting for '2022-06-15T08:18:44Z' backup.......... done

   pbm-conf> pbm status -s backups

   Backups:
   ========
   FS  /data/pbm
     Snapshots:
       2022-06-15T08:18:44Z 28.23KB <logical> [complete: 2022-06-15T08:18:49Z]
   ```

5. The rest of operations will be carried out on your *destination* cluster in a Kubernetes-based environment of your choice. These actions are described in the How to restore backup to a new Kubernetes-based environment guide. Just use the proper name of the backup (`2022-06-15T08:18:44Z`) in the above example, and proper parameters specific to your cloud storage (e.g. the `pbm-test-bucket` bucket name we used above).

10.6 Install Percona Operator for MongoDB in multi-namespace (cluster-wide) mode

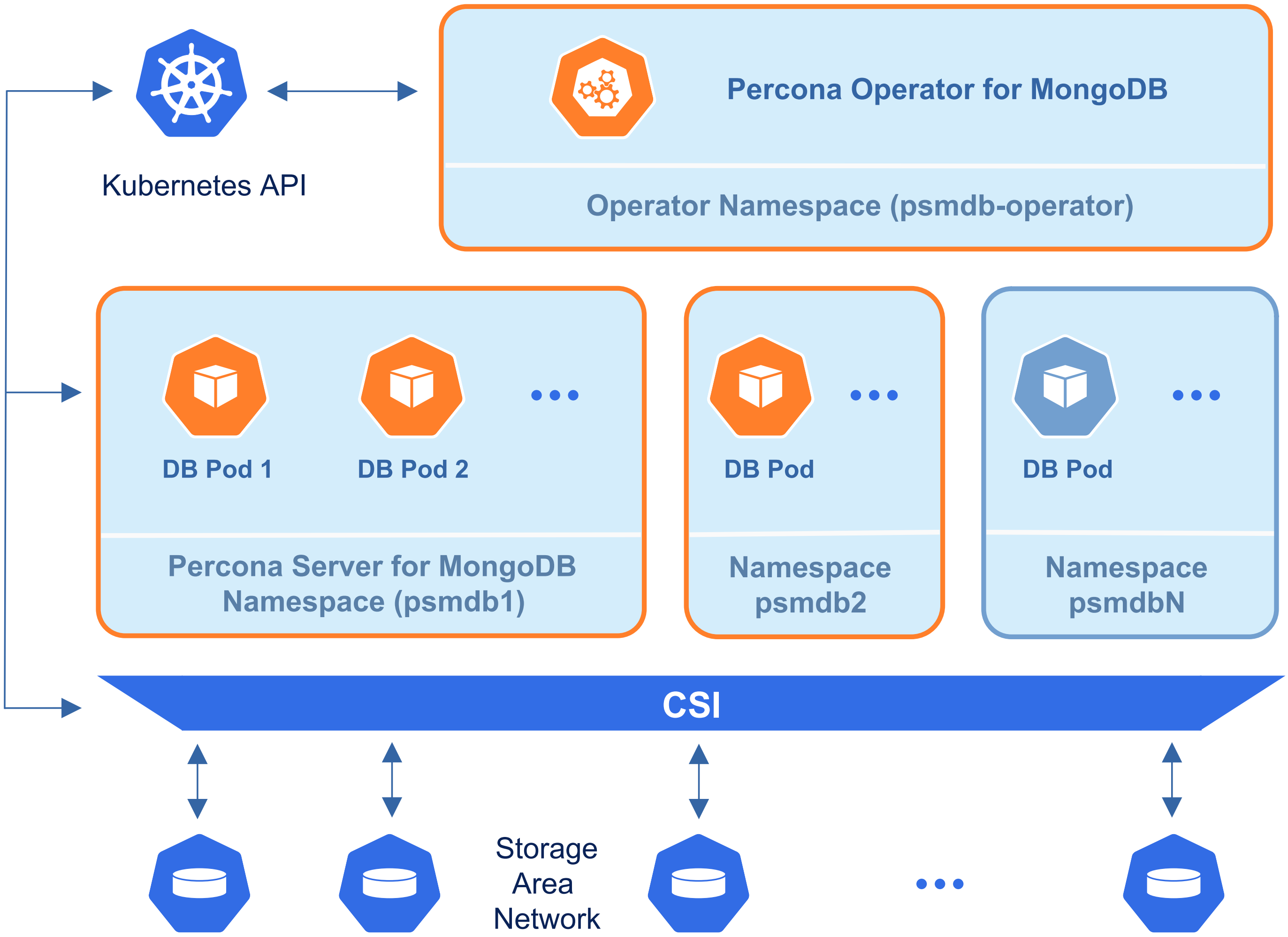
Difference between single-namespace and multi-namespace Operator deployment

By default, Percona Operator for MongoDB functions in a specific Kubernetes namespace. You can create one during 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:



Still, sometimes it is more convenient to have one Operator watching for Percona Server for MongoDB Custom Resources in several namespaces.

We recommend running Percona Operator for MongoDB in a traditional way, limited to a specific namespace. But it is possible to run it in so-called *cluster-wide* mode, one Operator watching several namespaces, if needed:



> **Note**
>
> Please take into account that if several Operators are configured to watch the same namespace, it is entirely unpredictable which one will get ownership of the Custom Resource in it, so this situation should be avoided.

Installing the Operator in cluster-wide mode

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, *and* the namespace in which the Operator resides (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. First of all, clone the percona-server-mongodb-operator repository:

```
git clone -b v1.22.0 https://github.com/percona/percona-server-mongodb-operator
cd percona-server-mongodb-operator
```

2. Let's suppose that Operator's namespace should be the `psmdb-operator` one. Create it as follows:

```
kubectl create namespace psmdb-operator
```

Namespaces to be watched by the Operator should be created in the same way if not exist. Let's say the Operator should watch the `psmdb` namespace:

```
kubectl create namespace psmdb
```

3. Edit the `deploy/cw-bundle.yaml` configuration file to set proper namespaces:

```
...
subjects:
- kind: ServiceAccount
  name: percona-server-mongodb-operator
  namespace: "psmdb-operator"
...
env:
         - name: WATCH_NAMESPACE
           value: "psmdb"
...
```

4. [Apply](#) the `deploy/cw-bundle.yaml` file with the following command:

```
kubectl apply -f deploy/cw-bundle.yaml --server-side -n psmdb-operator
```

5. After the Operator is started, Percona Server for MongoDB can be created at any time by applying the `deploy/cr.yaml` configuration file, like in the case of normal installation:

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

The creation process may take some time. When the process is over your cluster will obtain the `ready` status. You can check it by quering the `PerconaServerMongoDB` Custom Resource (it has handy `psmdb` shortname also) with the following command:

```
kubectl get psmdb -n psmdb
```

Expected output

```
NAME             ENDPOINT                                         STATUS   AGE
my-cluster-name  my-cluster-name-mongos.psmdb.svc.cluster.local   ready    5m26s
```

Verifying the cluster operation

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

1. You will need the login and password for the admin user to access the cluster. Use `kubectl get secrets` command to see the list of Secrets objects (by default the Secrets object you are interested in has `my-cluster-name-secrets` name). Then `kubectl get secret my-cluster-name-secrets -o yaml` command will return the YAML file with generated Secrets, including the `MONGODB_DATABASE_ADMIN` and `MONGODB_DATABASE_ADMIN_PASSWORD` strings, which should look as follows:

```
...
data:
  ...
  MONGODB_DATABASE_ADMIN_PASSWORD: aDAzQ0pCY3NSWEZ2ZUIzS1I=
  MONGODB_DATABASE_ADMIN_USER: ZGF0YWJhc2VBZG1pbg==
```

Here the actual login name and password are base64-encoded. Use `echo 'aDAzQ0pCY3NSWEZ2ZUIzS1I=' | base64 --decode` command to bring it back to a human-readable form.

2. Run a container with a MongoDB client and connect its console output to your terminal. The following command will do this, naming the new Pod `percona-client`:

```
kubectl run -i --rm --tty percona-client --image=percona/percona-server-mongodb:6.0.27-21 --restart=Never --
env="POD_NAMESPACE=psmdb" -- bash -il
```

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

3. Now run `mongo` tool in the percona-client command shell using the login (which is normally `databaseAdmin`) and a proper password obtained from the Secret. The command will look different depending on whether sharding is on (the default behavior) or off:

   if sharding is on

   ```
   mongosh "mongodb://databaseAdmin:databaseAdminPassword@my-cluster-name-mongos.psmdb.svc.cluster.local/admin?ssl=false"
   ```

   if sharding is off

   ```
   mongosh "mongodb+srv://databaseAdmin:databaseAdminPassword@my-cluster-name-rs0.psmdb.svc.cluster.local/admin?
   replicaSet=rs0&ssl=false"
   ```

10.7 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 PerconaServerMongoDB.

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-mongodb-operator` deployment controls the number of concurrent workers that can reconcile resources in Percona Server for MongoDB 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 MongoDB clusters. When the number of workers is greater, the excessive workers will remain idle.

Set the number of reconciliation workers

1. Check the index of the `MAX_CONCURRENT_RECONCILES` environment variable using the following command:

```
kubectl get deployment percona-server-mongodb-operator -o jsonpath='{.spec.template.spec.containers[0].env[?
(@.name=="MAX_CONCURRENT_RECONCILES")].value}'
```

Sample output

```
1
```

2. To set a new value and verify it's been updated, run the following command:

```
kubectl patch deployment percona-server-mongodb-operator \
--type='strategic' \
-o yaml \
-p='{
    "spec": {
        "template": {
            "spec": {
                "containers": [
                    {
                        "name": "percona-server-mongodb-operator",
                        "env": [
                            {
                                "name": "MAX_CONCURRENT_RECONCILES",
                                "value": "2"
                            }
                        ]
                    }
                ]
            }
        }
    }
}'\
-o jsonpath='{.spec.template.spec.containers[0].env[?(@.name=="MAX_CONCURRENT_RECONCILES")].value}'
```

The command does the following:

- Patches the deployment to update the `MAX_CONCURRENT_RECONCILES` environment variable
- Sets the value to `2`.
- Outputs the result

You can set the value to any number greater than 0.

Sample output

```
2
```

10.8 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 Kubernetes monitoring stack as the metrics collector. These steps may also apply if you use another Prometheus-compatible storage.

The Operator is compatible with both PMM versions 2 and 3. PMM2 has reached end-of-life. Therefore, we recommend using the latest PMM version 3 for optimal monitoring capabilities.

The steps in this tutorial are for PMM 3.

Pre-requisites

To set up monitoring of Kubernetes, you need the following:

1. PMM Server up and running. You can run PMM Server 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 v3.
3. kubectl.
4. PMM 3 Service account token (or PMM2 API key).

Configure authentication

PMM3 (recommended)

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. [Generate a service account and token](). Specify the Admin role for the service account.

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.

PMM2

[Get the PMM API key from PMM Server](). The API key must have the role "Admin". You need this key to authorize PMM Client within PMM Server.

From PMM UI

[Generate the PMM API key]()

From command line

You can query your PMM Server installation for the API Key using `curl` and `jq` utilities. Replace `<login>:<password>@<server_host>` placeholders with your real PMM Server login, password, and hostname in the following command:

```
API_KEY=$(curl --insecure -X POST -H "Content-Type: application/json" -d '{"name":"operator", "role": "Admin"}'
"https://<login>:<password>@<server_host>/graph/api/auth/keys" | jq .key)
```

> **Warning**
>
> The API key is not rotated.

Install the Victoria Metrics Kubernetes monitoring stack

Quick install

1. To install the Victoria Metrics Kubernetes monitoring stack with the default parameters, use the quick install command. 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`.

   ```
   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>
   ```

> **Note**
>
> 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, 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.

Configure authentication in PMM

To access the PMM Server resources and perform actions on the server, configure authentication.

1. Encode the PMM Server token key with base64.

   Linux

   ```
   $ echo -n <API-key> | base64 --wrap=0
   ```

   macOS

   ```
   echo -n <API-key> | base64
   ```

2. 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 latter steps, specify your namespace instead of the `<namespace>` placeholder.

   ```
   kubectl create namespace monitoring-system
   ```

3. Create the YAML file for the [Kubernetes Secrets]() and specify the base64-encoded API key value within. 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: default
   type: Opaque
   ```

4. Create the Secrets object using the YAML file you created previously. Replace the `<filename>` placeholder with your value.

   ```
   kubectl apply -f pmm-api-vmoperator.yaml -n <namespace>
   ```

5. 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>` placeholder with your value. 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

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

Sample output

```
vm-k8s-stack-kube-state-metrics-d9d85978d-9pzbs                1/1     Running   0          28m
vm-k8s-stack-victoria-metrics-operator-844d558455-gvg4n        1/1     Running   0          28m
vmagent-vm-k8s-stack-victoria-metrics-k8s-stack-55fd8fc4fbcxwhx  2/2     Running   0          28m
```

What Pods are running depends on the configuration chosen in values used while installing `victoria-metrics-k8s-stack` chart.

Validate the successful installation

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

Sample output

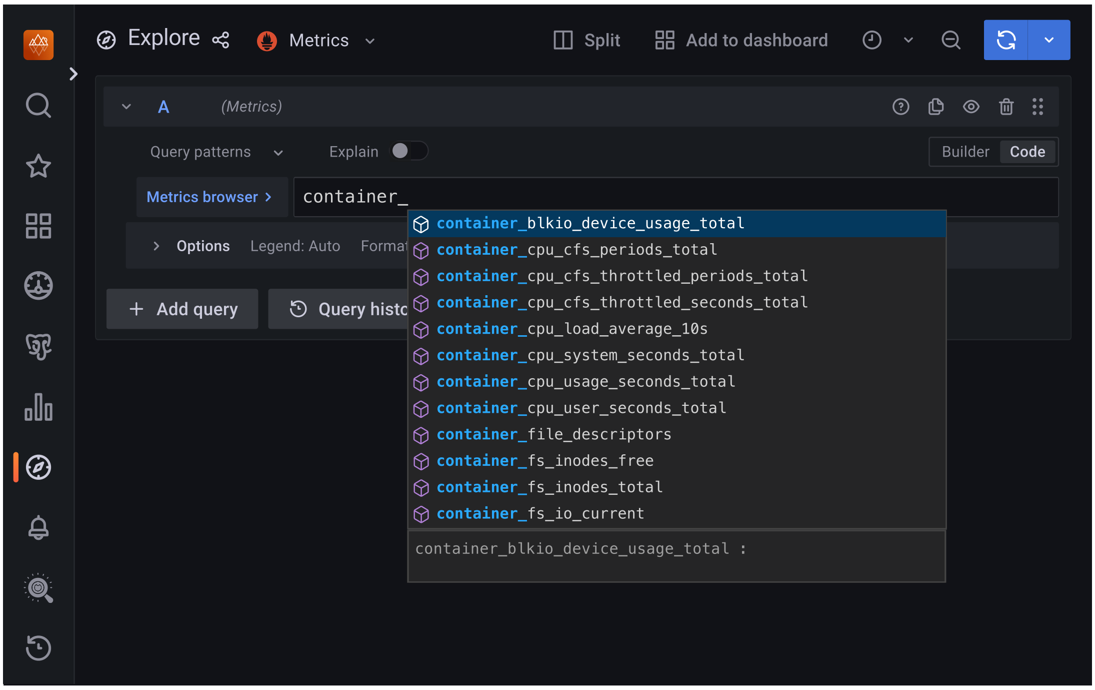
```
vm-k8s-stack-kube-state-metrics-d9d85978d-9pzbs                   1/1     Running   0          28m
vm-k8s-stack-victoria-metrics-operator-844d558455-gvg4n           1/1     Running   0          28m
vmagent-vm-k8s-stack-victoria-metrics-k8s-stack-55fd8fc4fbcxwhx   2/2     Running   0          28m
```

What Pods are running depends on the configuration chosen in values used while installing `victoria-metrics-k8s-stack` chart.

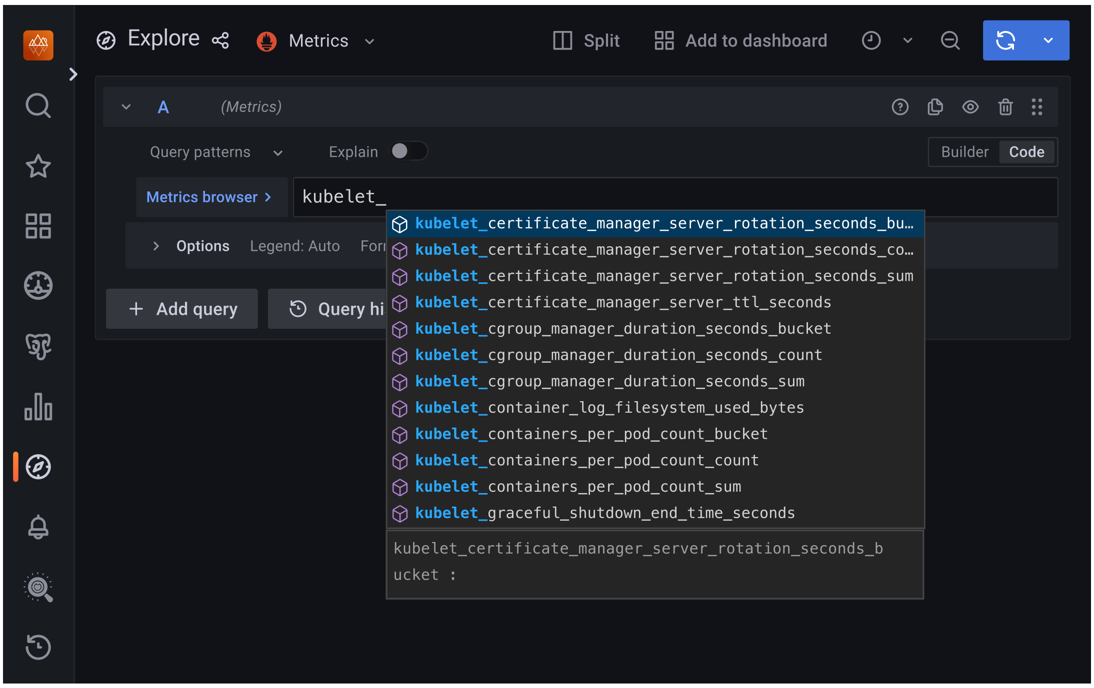
Verify metrics capture

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:

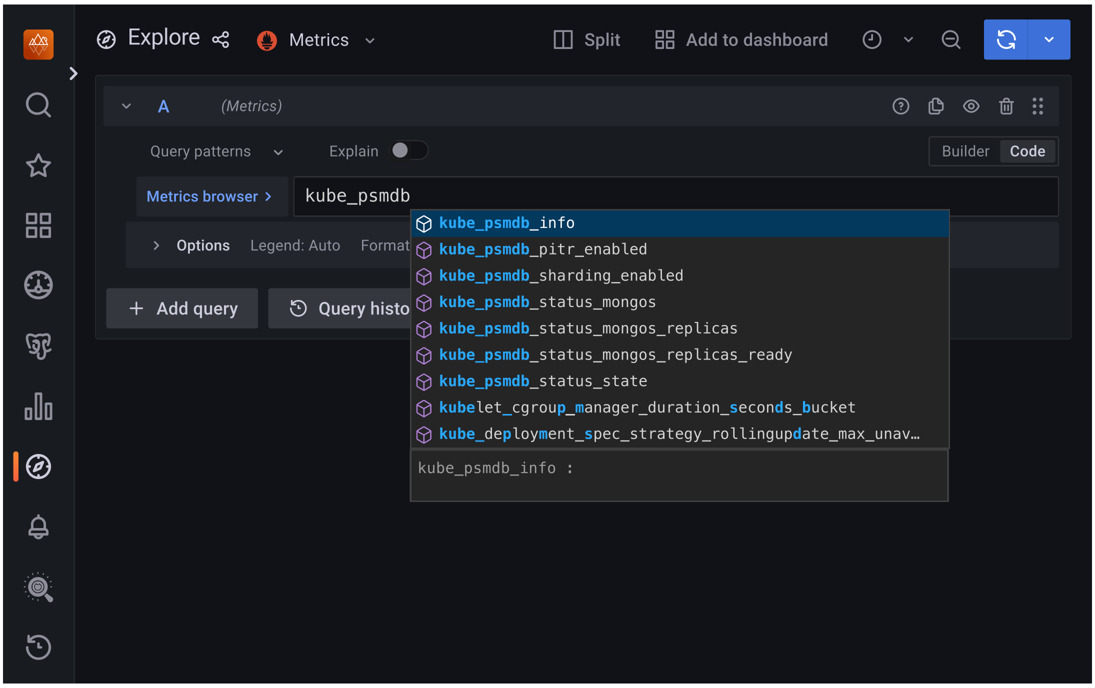
- [cadvisor](#):



- 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>
```

10.9 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/psmdb-operator/<operator-version>/<psmdb-version> | jq -r '.versions[].matrix'
```

where:

- **`<operator-version>`** is the version of the Percona Operator for MongoDB you are using.
- **`<psmdb-version>`** is the version of Percona Server for MongoDB you want to get images for. This part is optional and helps filter the results. It can be a specific Percona Server for MongoDB version (e.g. 8.0.8-3), a recommended version (e.g. 7.0-recommended), or the latest available version (e.g. 8.0-latest).

For example, to retrieve the list of images for the Operator version `1.20.0` and the latest version of Percona Server for MongoDB 8.0, use the following command:

```
curl https://check.percona.com/versions/v1/psmdb-operator/1.20.0/8.0-latest |jq -r '.versions[].matrix'
```

Sample output


```
{
"mongod": {
    "8.0.8-3": {
        "imagePath": "percona/percona-server-mongodb:8.0.8-3",
        "imageHash": "e4580ca292f07fd7800e139121aea4b2c1dfa6aa34f3657d25a861883fd3de41",
        "imageHashArm64": "96cfee2102499aba05e63ca7862102c2b1da1cf9f4eea0cbea3793a07c183925",
        "status": "available",
        "critical": false
    }
},
"pmm": {
    "2.44.1": {
        "imagePath": "percona/pmm-client:2.44.1",
        "imageHash": "8b2eaddffd626f02a2d5318ffebc0c277fe8457da6083b8cfcada9b6e6168616",
        "imageHashArm64": "337fecd4afdb3f6daf2caa2b341b9fe41d0418a0e4ec76980c7f29be9d08b5ea",
        "status": "recommended",
        "critical": false
    }
},
"backup": {
    "2.9.1": {
        "imagePath": "percona/percona-backup-mongodb:2.9.1",
        "imageHash": "976bfbaa548eb70dd90bf0bd2dcfe40b2994d749ef644af3a0590f4856e4d7e2",
        "imageHashArm64": "ebc6e5c5aa3ed97991d3fd90e9201597b485ddc0eae8d7ee4311ecb785c03bf0",
        "status": "recommended",
        "critical": false
    }
},
"operator": {
    "1.20.0": {
        "imagePath": "percona/percona-server-mongodb-operator:1.20.0",
        "imageHash": "01da3139b0f7f64a27f3642ca06581ea065a02891b13ce2375d61471011d6dd4",
        "imageHashArm64": "26d885398af42d18928f51f070aff770df900eb5ddf46e3e0bc2570720089bb1",
        "status": "recommended",
        "critical": false
    }
},
```


To narrow down the search and check the Percona Server for MySQL images available for a specific Operator version (`1.20.1` in the following example), use the following command:

```
curl -s https://check.percona.com/versions/v1/psmdb-operator/1.20.1 | jq -r '.versions[0].matrix.mongod | to_entries[] | "\(.key)\t\(.value.imagePath)\t\(.value.status)"'
```

Sample output

```
6.0.15-12   percona/percona-server-mongodb:6.0.15-12    available
6.0.16-13   percona/percona-server-mongodb:6.0.16-13    available
6.0.18-15   percona/percona-server-mongodb:6.0.18-15-multi  available
6.0.19-16   percona/percona-server-mongodb:6.0.19-16-multi  available
6.0.21-18   percona/percona-server-mongodb:6.0.21-18    recommended
7.0.12-7    percona/percona-server-mongodb:7.0.12-7 available
7.0.14-8    percona/percona-server-mongodb:7.0.14-8-multi   available
7.0.15-9    percona/percona-server-mongodb:7.0.15-9-multi   available
7.0.18-11   percona/percona-server-mongodb:7.0.18-11    recommended
8.0.4-1 percona/percona-server-mongodb:8.0.4-1-multi    available
8.0.8-3 percona/percona-server-mongodb:8.0.8-3  available
```

10.10 Delete Percona Operator for MongoDB

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 Distribution for MongoDB cluster managed by the Operator
- Percona Operator for MongoDB itself
- Custom Resource Definitions 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.

There are two finalizers defined in the Custom Resource, which are related to cluster deletion:

- `percona.com/delete-psmdb-pods-in-order`: it is enabled by default and it ensures the Pods are deleted in order on cluster deletion. PVCs are not deleted.
- `percona.com/delete-psmdb-pvc`: if present, Persistent Volume Claims for the database cluster Pods are deleted along with the cluster deletion.

  This finalizer is off by default in the `deploy/cr.yaml` configuration file, allowing you to recreate the cluster without losing data. If you need, you can delete TLS-related objects and PVCs manually.

The steps are the following:

1. List the Custom Resources. Replace the `<namespace>` placeholder with your value

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

2. Delete the Custom Resource with the name of your cluster

```
kubectl delete psmdb <cluster_name> -n <namespace>
```

It may take a while to stop and delete the cluster.

Sample output

```
perconaservermongodb.psmdb.percona.com "my-cluster-name" deleted
```

3. Check that the cluster is deleted by listing the Custom Resources again:

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

Sample output

```
No resources found in <namespace> namespace.
```

Delete the Operator

Choose the instructions relevant to the way you installed the Operator.

Use 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>
```

2. Delete the `percona-*` deployment

```
kubectl delete deploy percona-server-mongodb-operator -n <namespace>
```

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 MongoDB cluster 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
```

5. Delete the `percona*.psmdb.percona.com` CRDs

```
kubectl delete crd perconaservermongodbbackups.psmdb.percona.com perconaservermongodbrestores.psmdb.percona.com
perconaservermongodbs.psmdb.percona.com
```

Sample output

```
customresourcedefinition.apiextensions.k8s.io "perconaservermongodbbackups.psmdb.percona.com" deleted
customresourcedefinition.apiextensions.k8s.io "perconaservermongodbrestores.psmdb.percona.com" deleted
customresourcedefinition.apiextensions.k8s.io "perconaservermongodbs.psmdb.percona.com" deleted
```

Use Helm

To delete the Operator, do the following:

1. List the Helm charts:

```
helm list -n <namespace>
```

Sample output

```
cluster1    <namespace>         1           2023-10-31 10:18:10.763049 +0100 CET    deployed    psmdb-db-1.14.4         1.22.0
my-op       <namespace>         1           2023-10-31 10:15:18.41444 +0100 CET     deployed    psmdb-operator-1.14.3   1.22.0
```

2. Delete the release object for Percona Server for MongoDB

```
helm uninstall cluster1 --namespace <namespace>
```

3. Delete the release object for the Operator

```
helm uninstall my-op --namespace <namespace>
```

This command deletes only the Operator deployment. The CRDs remain in the Kubernetes environment, allowing other clusters in other namespaces to run.

4. Delete Custom Resource Definitions:

!!! warning

```
 Although you install the CRD Helm chart (`psmdb-operator-crds`) in a specific namespace, the CRDs themselves are cluster-
scoped resources and apply to *all* namespaces in your Kubernetes environment. If you uninstall or delete this CRD chart,
it will remove the CRDs from the Kubernetes environment. This will irreversibly delete *all* Percona Server for MongoDB
clusters managed by these CRDs across *all* namespaces where you deployed them.

 If you run other Percona Server for MongoDB clusters in other namespaces, skip this step.
```

```
helm uninstall psmdb-operator-crds --namespace <namespace>
```

Clean up resources

By default, TLS-related objects 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-psmdb-pvc` finalizer. Note that in this case user Secrets will also be deleted.

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
   mongod-data-my-cluster-name-cfg-0   Bound    pvc-245641fe-b172-439b-8c9c-cba5ea4ccd80   3Gi        RWO            standard-rwo   10m
   mongod-data-my-cluster-name-cfg-1   Bound    pvc-4ff7c3c4-b91c-4938-a52e-591fd559f4a4   3Gi        RWO            standard-rwo   9m19s
   mongod-data-my-cluster-name-cfg-2   Bound    pvc-acbff4a3-784a-48e7-ad4b-8b00239982d3   3Gi        RWO            standard-rwo   8m36s
   mongod-data-my-cluster-name-rs0-0   Bound    pvc-0a56e9ab-e22b-47ce-95de-a55f2676456a   3Gi        RWO            standard-rwo   10m
   mongod-data-my-cluster-name-rs0-1   Bound    pvc-cd075679-a7f5-4182-a8ce-341db1fb12d3   3Gi        RWO            standard-rwo   9m19s
   mongod-data-my-cluster-name-rs0-2   Bound    pvc-9ff0d41d-c739-494d-a45c-576f3a1fb590   3Gi        RWO            standard-rwo   8m26s
   ```

   2. Delete PVCs related to your cluster. The following command deletes PVCs for the `my-cluster-name` cluster:

   ```
   kubectl delete pvc mongod-data-my-cluster-name-cfg-0 mongod-data-my-cluster-name-cfg-1 mongod-data-my-cluster-name-
   cfg-2 mongod-data-my-cluster-name-rs0-0 mongod-data-my-cluster-name-rs0-1 mongod-data-my-cluster-name-rs0-2 -n
   <namespace>
   ```

   Sample output

   ```
   persistentvolumeclaim "mongod-data-my-cluster-name-cfg-0" deleted persistentvolumeclaim "mongod-data-my-cluster-name-cfg-1" deleted
   persistentvolumeclaim "mongod-data-my-cluster-name-cfg-2" deleted
   persistentvolumeclaim "mongod-data-my-cluster-name-rs0-0" deleted
   persistentvolumeclaim "mongod-data-my-cluster-name-rs0-1" deleted
   persistentvolumeclaim "mongod-data-my-cluster-name-rs0-2" deleted
   ```

   Note that if your Custom Resource manifest includes the `percona.com/delete-psmdb-pvc` finalizer, all Secrets will be automatically deleted when you delete the PVCs. To prevent this from happening, disable the finalizer.

2. Delete the Secrets

   1. List Secrets:

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

2. Delete the Secret:

```
kubectl delete secret <secret_name> -n <namespace>
```

11 Reference

11.1 Custom Resource options

The Operator uses Custom Resources (CRs) to manage the database cluster and related objects such as backups and restores. A CR defines the desired state of an object. The Operator reads the CR and reconciles that object to bring it to the state you define.

There are the following Custom Resources:

- `PerconaServerMongoDB` contains options to configure Percona Server for MongoDB.
- `PerconaServerMongoDBBackup` contains options for PBM to make backups.
- `PerconaServerMongoDBRestore` contains options to restore Percona Server for MongoDB from backups.

PerconaServerMongoDB Custom Resource options

To customize PerconaServerMongoDB Custom Resource, edit the `spec` section in the deploy/cr.yaml.

This document explains every section of the `deploy/cr.yaml` Custom Resource manifest and describes available options.

`apiVersion`

Specifies the API version of the Custom Resource. `psmdb.percona.com` indicates the group, and `v1` is the version of the API.

This tells Kubernetes which version of the custom resource definition (CRD) to use.

`kind`

Defines the type of resource being created.

`metadata`

The metadata part of the `deploy/cr.yaml` contains metadata about the resource, such as its name and other attributes. It includes the following keys:

- `name` sets the name of your Percona Server for MongoDB Cluster. The name must follow these rules:
  - include only URL-compatible characters,
  - not exceed 22 characters,
  - start and end with an alphanumeric character

  The default name is `my-cluster-name`.
- `finalizers` ensure safe deletion of resources in Kubernetes under certain conditions. This subsection includes the following finalizers:
  - `percona.com/delete-psmdb-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-psmdb-pvc` if present, activates the Finalizer which deletes appropriate Persistent Volume Claims after the cluster deletion event (off by default). It also deletes Secrets.
  - `percona.com/delete-pitr-chunks` if present, activates the Finalizer which deletes all point-in-time recovery chunks from the cloud storage on cluster deletion (off by default)

Toplevel `spec` elements

The spec part of the deploy/cr.yaml file contains the following keys and sections:

`platform`

Override/set the Kubernetes platform: `kubernetes` or `openshift`.

Value type	Example
S string	`kubernetes`

`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 scaling / volume expansion with Volume Expansion capability.

Value type	Example
boolean	`true`
This option is deprecated and will be removed in version 1.25.0. Use the	
`storageScaling.enableVolumeScaling` option instead.	

`enableExternalVolumeAutoscaling`

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`

This option is deprecated and will be removed in version 1.25.0. Use the `storageScaling.enableExternalAutoscaling` option instead.

`unmanaged`

Setting it to `true` instructs the Operator to run the cluster in unmanaged state - the Operator does not form replica sets, and does not generate TLS certificates or user credentials. This can be useful for migration scenarios and for cross-site replication.

Value type	Example
boolean	`false`

`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. Read more about it in Storage resizing with Volume Expansion capability

Value type	Example
boolean	`false`

`storageScaling.autoscaling.enabled`

Enables or disables 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. Minimum value is 50, maximum is 95.

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.

Value type	Example
string	`10Gi`

`crVersion`

Version of the Operator the Custom Resource belongs to.

Value type	Example
string	`1.22.0`

`image`

The Docker image of Percona Server for MongoDB to deploy (actual image names can be found in the list of certified images).

Value type	Example
string	`percona/percona-server-mongodb:6.0.27-21`

`imagePullPolicy`

The policy used to update images.

Value type	Example
string	`Always`

`imagePullSecrets.name`

The [Kubernetes ImagePullSecret](#) to access the [custom registry](#).

Value type	Example
string	`private-registry-credentials`

`initImage`

An alternative image for the initial Operator installation.

Value type	Example
string	`percona/percona-server-mongodb-operator:1.22.0`

`initContainerSecurityContext`

A custom [Kubernetes Security Context for a Container](#) for the initImage (image, which can be used instead of the default one while the initial Operator installation).

Value type	Example
subdoc	`{}`

`ClusterServiceDNSSuffix`

The (non-standard) cluster domain to be used as a suffix of the Service name.

Value type	Example
string	`svc.cluster.local`

`clusterServiceDNSMode`

Can be `internal` (local fully-qualified domain names will be used in replset configuration even if the replset is exposed - the default value), `external` (exposed MongoDB instances will use ClusterIP addresses, [should be applied with caution](#)) or `ServiceMesh` (use a [special FQDN based on the Pod name](#)). Being set, `ServiceMesh` value suprecedes multiCluster settings, and therefore these two modes cannot be combined together.

Value type	Example
string	`Internal`

`allowUnsafeConfigurations`

Prevents users from configuring a cluster with unsafe parameters: starting it with less than 3 replica set instances, with an [even number of replica set instances without additional arbiter](#), or without TLS/SSL certificates, or running a sharded cluster with less than 3 config server Pods or less than 2 mongos Pods (if `false`, the Operator will automatically change unsafe parameters to safe defaults). *After switching to unsafe configurations permissive mode you will not be able to switch the cluster back by setting `spec.allowUnsafeConfigurations` key to `false`, the flag will be ignored*. **This option is deprecated and will be removed in future releases**. Use `unsafeFlags` subsection instead

Value type	Example
boolean	`false`

`updateStrategy`

A strategy the Operator uses for upgrades. Possible values are SmartUpdate, RollingUpdate and OnDelete.

Value type	Example
string	`SmartUpdate`

`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`

`multiCluster.enabled`

Multi-cluster Services (MCS): setting it to `true` enables MCS cluster mode.

Value type	Example
boolean	`false`

`multiCluster.DNSSuffix`

The cluster domain to be used as a suffix for multi-cluster Services used by Kubernetes (`svc.clusterset.local` by default).

Value type	Example
string	`svc.clusterset.local`

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.

Once you enable permissive mode with unsafe settings, you cannot disable it by simply turning the same settings back (e.g. by setting a configuration option to `false`). These settings will be ignored if you try to revert them. Reverting the cluster to a secure state may require additional steps or reinitialization.

`unsafeFlags.tls`

Prevents users from configuring a cluster without TLS/SSL certificates (if `false`, the Operator will automatically change unsafe parameters to safe defaults).

Value type	Example
boolean	`false`

`unsafeFlags.replsetSize`

Prevents users from configuring a cluster with unsafe parameters: starting it with less than 3 replica set instances or with an even number of replica set instances without additional arbiter (if `false`, the Operator will automatically change unsafe parameters to safe defaults).

Value type	Example
boolean	`false`

`unsafeFlags.mongosSize`

Prevents users from configuring a sharded cluster with less than 3 config server Pods or less than 2 mongos Pods (if `false`, the Operator will automatically change unsafe parameters to safe defaults).

Value type	Example
boolean	`false`

`unsafeFlags.terminationGracePeriod`

Prevents users from configuring a sharded cluster without termination grace period for replica set, config servers and mongos Pods.

Value type	Example
boolean	`false`

`unsafeFlags.backupIfUnhealthy`

Prevents running backup on a cluster with failed health checks.

Value type	Example
boolean	`false`

TLS (extended cert-manager configuration section)

The `tls` section in the deploy/cr.yaml file contains various configuration options for additional customization of the Transport Layer Security.

`tls.mode`

Controls if the TLS encryption should be used and/or enforced. Can be `disabled`, `allowTLS`, `preferTLS`, or `requireTLS`. If set to `disabled`, it also requires setting `unsafeFlags.tls option to`true`.

Value type	Example
string	`preferTLS`

`tls.certValidityDuration`

The validity duration of the external certificate for cert manager (90 days by default). This value is used only at cluster creation time and can't be changed for existing clusters.

Value type	Example
string	`2160h`

`tls.allowInvalidCertificates`

If `true`, the mongo shell will not attempt to validate the server certificates. **Should be true (default variant) to use self-signed certificates generated by the Operator when there is no cert-manager.**

Value type	Example
boolean	`true`

`tls.issuerConf.name`

A [cert-manager issuer name](https://cert-manager.io/docs/concepts/issuer/).

Value type	Example
string	`special-selfsigned-issuer`

`tls.issuerConf.kind`

A [cert-manager issuer type](https://cert-manager.io/docs/configuration/).

`tls.issuerConf.group`

A [cert-manager issuer group](https://cert-manager.io/docs/configuration/). Should be `cert-manager.io` for built-in cert-manager certificate issuers.

Value type	Example
string	`cert-manager.io`

Upgrade Options Section

The `upgradeOptions` section in the [deploy/cr.yaml](https://github.com/percona/percona-server-mongodb-operator/blob/main/deploy/cr.yaml) file contains various configuration options to control Percona Server for MongoDB upgrades.

`upgradeOptions.versionServiceEndpoint`

The Version Service URL used to check versions compatibility for upgrade.

Value type	Example
string	`https://check.percona.com`

`upgradeOptions.apply`

Specifies how [updates are processed](update.md#automated-upgrade) by the Operator. `Never` or `Disabled` will completely disable automatic upgrades, otherwise it can be set to `Latest` or `Recommended` or to a specific version string of Percona Server for MongoDB (e.g. `6.0.27-21`) that is wished to be version-locked (so that the user can control the version running, but use automatic upgrades to move between them).

Value type	Example
string	`disabled`

`upgradeOptions.schedule`

Scheduled time to check for updates, specified in the [crontab format](https://en.wikipedia.org/wiki/Cron).

Value type	Example
string	`0 2 \* \* \*`

`upgradeOptions.setFCV`

If enabled, [FeatureCompatibilityVersion (FCV)](#) will be set to match the version during major version upgrade.

Value type	Example
boolean	`false`

Secrets section

Each spec in its turn may contain some key-value pairs. The secrets one has only two of them:

`secrets.keyFile`

The secret name for the MongoDB Internal Auth Key file. This secret is auto-created by the operator if it doesn't exist.

Value type	Example
string	`my-cluster-name-mongodb-keyfile`

`secrets.users`

The name of the Secrets object for the MongoDB users required to run the operator.

Value type	Example
string	`my-cluster-name-secrets`

`secrets.sse`

The name of the Secrets object for server side encryption credentials

Value type	Example
string	`my-cluster-name-sse`

`secrets.ssl`

A secret with TLS certificate generated for *external* communications. When generated by the Operator, the name defaults to `<cluster-name>-ssl`. See Transport Layer Security (TLS) for details.

Value type	Example
string	`my-custom-ssl`

`secrets.sslInternal`

A secret with TLS certificate generated for *internal* communications. When generated by the Operator, the name defaults to `<cluster-name>-sslInternal`. See Transport Layer Security (TLS) for details.

Value type	Example
string	`my-custom-ssl-internal`

`secrets.encryptionKey`

Specifies a secret object with the encryption key.

Value type	Example
string	`my-cluster-name-mongodb-encryption-key`

`secrets.vault`

Specifies a secret object to provide integration with HashiCorp Vault.

Value type	Example
string	`my-cluster-name-vault`

`secrets.ldapSecret`

Specifies a secret object for LDAP over TLS connection between MongoDB and OpenLDAP server.

Value type	Example
string	`my-ldap-secret`

Vault section

The Vault section defines how the Operator connects to HashiCorp Vault to sync system user credentials. Read more about this feature in Manage system users with Vault

`vault.endpointUrl`

The Vault server address.

Value type	Example
string	`https://vault-service:8200`

`vault.tlsSecret`

The Kubernetes Secret with Vault TLS certificates. If set, the Operator uses these certificates for TLS connections to Vault.

Value type	Example
string	`my-tls-vault-secret`

`vault.syncUsers.role`

The Vault Kubernetes auth role name.

Value type	Example
string	`operator`

`vault.syncUsers.mountPath`

The root path where a specific secrets engine is enabled inside Vault.

Value type	Example
S string	`secret`

`vault.syncUsers.keyPath`

The Vault path where system user credentials are stored.

Value type	Example
S string	`psmdb/operator/{namespace}/my-cluster-name/users`

`vault.syncUsers.tokenSecret`

The Kubernetes Secret that contains a Vault token for token-based authentication. If this value is set, the Operator uses the provided token to authenticate with Vault. If this value is not set and authentication with Kubernetes service account is also enabled, the Operator will use its service account with Vault Kubernetes authentication.

Value type	Example
S string	`vault-token-operator`

Replsets Section

The replsets section controls the MongoDB Replica Set.

`replsets.name`

The name of the [MongoDB Replica Set](https://docs.mongodb.com/manual/replication/).

Value type	Example
S string	`rs 0`

`replsets.size`

The size of the MongoDB Replica Set, must be >= 3 for High-Availability.

Value type	Example
I int	`3`

`replsets.terminationGracePeriodSeconds`

The amount of seconds Kubernetes will wait for a clean replica set Pods termination.

Value type	Example
I int	`300`

`replsets.serviceAccountName`

Name of the separate privileged service account for Replica Set Pods.

Value type	Example

string	`default`

`replsets.topologySpreadConstraints.labelSelector.matchLabels`

The label selector for the [Kubernetes Pod Topology Spread Constraints]().

Value type	Example
label	`app.kubernetes.io/name: percona-server-mongodb`

`replsets.topologySpreadConstraints.maxSkew`

The degree to which Pods may be unevenly distributed under the Kubernetes Pod Topology Spread Constraints.

Value type	Example
int	`1`

`replsets.topologySpreadConstraints.topologyKey`

The key of node labels for the Kubernetes Pod Topology Spread Constraints.

Value type	Example
string	`kubernetes.io/hostname`

`replsets.topologySpreadConstraints.whenUnsatisfiable`

What to do with a Pod if it doesn't satisfy the Kubernetes Pod Topology Spread Constraints.

Value type	Example
string	`DoNotSchedule`

`replsets.replsetOverrides.MEMBER-NAME.host`

Use if you need to override the replica set members FQDNs with custom host names. Each key (`MEMBER-NAME`) under `replsetOverrides` should be name of a Pod. The Operator won't perform any validation for hostnames, so it's the user's responsibility to ensure connectivity.

Value type	Example
string	`my-cluster-name-rs0-0.example.net:27017`

`replsets.replsetOverrides.MEMBER-NAME.priority`

Use if you need to override the replica set members priorities.

Value type	Example
int	`3`

`replsets.replsetOverrides.MEMBER-NAME.tags`

Optional custom tags which can be added to the replset members to make their identification easier.

Value type	Example
label	`key: value-0`

`replsets.externalNodes.host`

The URL or IP address of the external replica set instance.

Value type	Example
string	`34.124.76.90`

`replsets.externalNodes.port`

The port number of the external replset instance.

Value type	Example
string	`27017`

`replsets.externalNodes.votes`

The number of votes of the external replset instance.

Value type	Example
string	`0`

`replsets.externalNodes.priority`

The priority of the external replset instance.

Value type	Example
string	`0`

`replsets.configuration`

Custom configuration options for mongod. Please refer to the official manual for the full list of options, and specific Percona Server for MongoDB docs.

Value type	Example

subdoc	<pre>|
	operationProfiling:
	  mode: slowOp
	systemLog:
	  verbosity: 1
	storage:
	  engine: wiredTiger
	  wiredTiger:
	    engineConfig:
	      directoryForIndexes: false
	      journalCompressor: snappy
	    collectionConfig:
	      blockCompressor: snappy
	    indexConfig:
	      prefixCompression: true</pre>

`replsets.env.name`

The name of a custom environment variable for mongod containers in this replica set (available since Operator 1.22.0). See Define custom environment variables.

Value type	Example
S string	`MY_ENV`

`replsets.env.value`

The vallue for a custom environment variable for mongod containers in this replica set (available since Operator 1.22.0). See Define custom environment variables.

Value type	Example
S string	`MY_VALUE`

`replsets.envFrom.configMapRef.name`

The name of a ConfigMap from where environment variables will be loaded for mongod containers in this replica set (available since Operator 1.22.0). See Define custom environment variables.

Value type	Example
S	`my-configmap`

`replsets.envFrom.secretRef.name`

The name of a Sexcret object from where environment variables will be loaded for mongod containers in this replica set (available since Operator 1.22.0). See Define custom environment variables.

Value type	Example
S	`custom-secret`

`replsets.affinity.antiAffinityTopologyKey`

The Kubernetes topologyKey node affinity constraint for the Replica Set nodes.

Value type	Example
S string	`kubernetes.io/hostname`

`replsets.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	

`replsets.tolerations.key`

The Kubernetes Pod tolerations key for the Replica Set nodes.

Value type	Example

string	`node.alpha.kubernetes.io/unreachable`

`replsets.tolerations.operator`

The [Kubernetes Pod tolerations](#) operator for the Replica Set nodes.

Value type	Example
string	`Exists`

`replsets.tolerations.effect`

The [Kubernetes Pod tolerations](#) effect for the Replica Set nodes.

Value type	Example
string	`NoExecute`

`replsets.tolerations.tolerationSeconds`

The [Kubernetes Pod tolerations](#) time limit for the Replica Set nodes.

Value type	Example
int	`6000`

`replsets.primaryPreferTagSelector.region`

Ensures the MongoDB instance is selected as Primary based on specified region

Value type	Example
string	`us-west-2`

`replsets.primaryPreferTagSelector.zone`

Ensures the MongoDB instance is selected as Primary based on specified zone

Value type	Example
string	`us-west-2c`

`replsets.priorityClassName`

The [Kuberentes Pod priority class](#) for the Replica Set nodes.

Value type	Example
string	`high priority`

`replsets.annotations`

The [Kubernetes annotations](#) metadata for the Replica Set nodes.

Value type	Example

string	`iam.amazonaws.com/role: role-arn`

`replsets.labels`

The [Kubernetes affinity labels](#) for the Replica Set nodes.

Value type	Example
label	`rack: rack-22`

`replsets.nodeSelector`

The [Kubernetes nodeSelector](#) affinity constraint for the Replica Set nodes.

Value type	Example
label	`disktype: ssd`

`replsets.storage.engine`

Sets the storage.engine option https://docs.mongodb.com/manual/reference/configuration-options/#storage.engine`_ for the Replica Set nodes.

Value type	Example
string	`wiredTiger`

`replsets.storage.wiredTiger.engineConfig.cacheSizeRatio`

The ratio used to compute the [storage.wiredTiger.engineConfig.cacheSizeGB option](#) for the Replica Set nodes.

Value type	Example
float	`0.5`

`replsets.storage.wiredTiger.engineConfig.directoryForIndexes`

Sets the [storage.wiredTiger.engineConfig.directoryForIndexes option](#) for the Replica Set nodes.

Value type	Example
boolean	`false`

`replsets.storage.wiredTiger.engineConfig.journalCompressor`

Sets the [storage.wiredTiger.engineConfig.journalCompressor option](#) for the Replica Set nodes.

Value type	Example
string	`snappy`

`replsets.storage.wiredTiger.collectionConfig.blockCompressor`

Sets the [storage.wiredTiger.collectionConfig.blockCompressor option](#) for the Replica Set nodes.

Value type	Example
string	`snappy`

`replsets.storage.wiredTiger.indexConfig.prefixCompression`

Sets the [storage.wiredTiger.indexConfig.prefixCompression option](#) for the Replica Set nodes.

Value type	Example
boolean	`true`

`replsets.storage.inMemory.engineConfig.inMemorySizeRatio`

The ratio used to compute the [storage.engine.inMemory.inMemorySizeGb option](#) for the Replica Set nodes.

Value type	Example
float	`0.9`

`replsets.livenessProbe.failureThreshold`

Number of consecutive unsuccessful tries of the [liveness probe](#) to be undertaken before giving up.

Value type	Example
int	`4`

`replsets.livenessProbe.initialDelaySeconds`

Number of seconds to wait after the container start before initiating the [liveness probe](#).

Value type	Example
int	`60`

`replsets.livenessProbe.periodSeconds`

How often to perform a [liveness probe](#) (in seconds).

Value type	Example
int	`30`

`replsets.livenessProbe.timeoutSeconds`

Number of seconds after which the [liveness probe](#) times out.

Value type	Example
int	`10`

`replsets.livenessProbe.startupDelaySeconds`

Time after which the liveness probe is failed if the MongoDB instance didn't finish its full startup yet.

Value type	Example
int	`7200`

`replsets.readinessProbe.failureThreshold`

Number of consecutive unsuccessful tries of the readiness probe to be undertaken before giving up.

Value type	Example
int	`8`

`replsets.readinessProbe.initialDelaySeconds`

Number of seconds to wait after the container start before initiating the readiness probe.

Value type	Example
int	`10`

`replsets.readinessProbe.periodSeconds`

How often to perform a readiness probe (in seconds).

Value type	Example
int	`3`

`replsets.readinessProbe.successThreshold`

Minimum consecutive successes for the readiness probe to be considered successful after having failed.

Value type	Example
int	`1`

`replsets.readinessProbe.timeoutSeconds`

Number of seconds after which the readiness probe times out.

Value type	Example
int	`2`

`replsets.containerSecurityContext`

A custom Kubernetes Security Context for a Container to be used instead of the default one.

Value type	Example
subdoc	`privileged: false`

`replsets.podSecurityContext`

A custom Kubernetes Security Context for a Pod to be used instead of the default one.

Value type	Example

subdoc	`runAsUser: 1001`
	`runAsGroup: 1001`
	`supplementalGroups: [1001]`

`replsets.runtimeClassName`

Name of the Kubernetes Runtime Class for Replica Set Pods.

Value type	Example
string	`image-rc`

`replsets.hookScript.configMapRef.name`

The name of the ConfigMap that contains a custom hook script that you can apply to the replica set Pods before the main process.

Value type	Example
string	`hookscript-configmap`

`replsets.hookScript.script`

The inline hook script that you can apply to the replica set Pods before the main process.

Value type	Example
string	`#!/usr/bin/env bash   echo "Hookscript started"`

`replsets.sidecars.image`

Image for the custom sidecar container for Replica Set Pods.

Value type	Example
string	`busybox`

`replsets.sidecars.command`

Command for the custom sidecar container for Replica Set Pods.

Value type	Example
array	`["/bin/sh"]`

`replsets.sidecars.args`

Command arguments for the custom sidecar container for Replica Set Pods.

Value type	Example
array	`["-c", "while true; do echo echo $(date -u) 'test' >> /dev/null; sleep 5;done"]`

`replsets.sidecars.name`

Name of the custom sidecar container for Replica Set Pods.

Value type	Example
string	`rs-sidecar-1`

`replsets.sidecars.volumeMounts.mountPath`

Mount path of the custom sidecar container volume for Replica Set Pods.

Value type	Example
string	`/volume1`

`replsets.sidecars.volumeMounts.name`

Name of the custom sidecar container volume for Replica Set Pods.

Value type	Example
string	`sidecar-volume-claim`

`replsets.sidecarVolumes.name`

Name of the custom sidecar container volume for Replica Set Pods.

Value type	Example
string	`sidecar-config`

`replsets.sidecarVolumes.configMap.name`

Name of the ConfigMap for a custom sidecar container volume for Replica Set Pods.

Value type	Example
string	`myconfigmap`

`replsets.sidecarVolumes.secret.secretName`

Name of the Secret for a custom sidecar container volume for Replica Set Pods.

Value type	Example
string	`sidecar-secret`

`replsets.sidecarVolumes.nfs.server`

The hostname of the NFS server that will provide remote filesystem to the custom sidecar container volume for Replica Set Pods.

Value type	Example
string	`nfs-service.storage.svc.cluster.local`

`replsets.sidecarVolumes.nfs.path`

The path on the NFS server that will be provided as a remote filesystem to the custom sidecar container volume for Replica Set Pods.

Value type	Example
string	`/psmdb-some-name-rs0`

`replsets.sidecarPVCs`

Persistent Volume Claim for the custom sidecar container volume for Replica Set Pods.

Value type	Example
subdoc	

`replsets.podDisruptionBudget.maxUnavailable`

The Kubernetes Pod distribution budget limit specifying the maximum value for unavailable Pods.

Value type	Example
int	`1`

`replsets.podDisruptionBudget.minAvailable`

The Kubernetes Pod distribution budget limit specifying the minimum value for available Pods.

Value type	Example
int	`1`

`replsets.splitHorizons.REPLICASET-POD-NAME.external`

External URI for Split-horizon for replica set Pods of the exposed cluster.

Value type	Example
string	`rs0-0.mycluster.xyz`

`replsets.splitHorizons.REPLICASET-POD-NAME.external-2`

External URI for Split-horizon for replica set Pods of the exposed cluster.

Value type	Example
string	`rs0-0.mycluster2.xyz`

`replsets.expose.enabled`

Enable or disable exposing MongoDB Replica Set nodes with dedicated IP addresses.

Value type	Example
boolean	`false`

`replsets.expose.type`

The [IP address type](#) to be exposed.

Value type	Example
string	`ClusterIP`

`replsets.expose.loadBalancerClass`

Define the implementation of the load balancer you want to use. This setting enables you to select a custom or specific load balancer class instead of the default one provided by the cloud provider.

Value type	Example
string	`eks.amazonaws.com/nlb`

`replsets.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`

`replsets.expose.annotations`

The [Kubernetes annotations](#) metadata for the MongoDB mongod daemon.

Value type	Example
string	`service.beta.kubernetes.io/aws-load-balancer-backend-protocol: http`

`replsets.expose.labels`

The [Kubernetes labels](#) for the MongoDB Replica Set Service.

Value type	Example
string	`rack: rack-22`

`replsets.expose.internalTrafficPolicy`

Specifies whether Service for MongoDB instances should [route internal traffic to cluster-wide or to node-local endpoints](#) (it can influence the load balancing effectiveness).

Value type	Example
boolean	`Local`

`replsets.expose.externalTrafficPolicy`

Specifies whether Service for MongoDB instances [should route external traffic](#) to cluster-wide (`Cluster`) or to node-local (`Local`) endpoints. It [can influence the load balancing effectiveness](#).

Value type	Example
string	`Local`

Service protocol configuration

Starting from Operator version 1.22.0, the Operator automatically sets `appProtocol: mongo` on Service ports for replica sets. This is required for service mesh implementations (such as Istio) because MongoDB uses a server-first protocol, which breaks mTLS without explicit protocol specification. The `appProtocol` field is set automatically and cannot be configured manually.

`replsets.nonvoting.enabled`

Enable or disable creation of [Replica Set non-voting instances] within the cluster.

Value type	Example
boolean	`false`

`replsets.nonvoting.size`

The number of [Replica Set non-voting instances] within the cluster.

Value type	Example
int	`1`

`replsets.nonvoting.podSecurityContext`

A custom [Kubernetes Security Context for a Pod] to be used instead of the default one.

Value type	Example
subdoc	`{}`

`replsets.nonvoting.containerSecurityContext`

A custom [Kubernetes Security Context for a Container] to be used instead of the default one.

Value type	Example
subdoc	`{}`

`replsets.nonvoting.affinity.antiAffinityTopologyKey`

The [Kubernetes topologyKey] node affinity constraint for the non-voting nodes.

Value type	Example
string	`kubernetes.io/hostname`

`replsets.nonvoting.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	

`replsets.nonvoting.tolerations.key`

The Kubernetes Pod tolerations key for the non-voting nodes.

Value type	Example
string	`node.alpha.kubernetes.io/unreachable`

`replsets.nonvoting.tolerations.operator`

The Kubernetes Pod tolerations operator for the non-voting nodes.

Value type	Example
string	`Exists`

`replsets.nonvoting.tolerations.effect`

The Kubernetes Pod tolerations effect for the non-voting nodes.

Value type	Example
string	`NoExecute`

`replsets.nonvoting.tolerations.tolerationSeconds`

The Kubernetes Pod tolerations time limit for the non-voting nodes.

Value type	Example
int	`6000`

`replsets.nonvoting.priorityClassName`

The Kuberentes Pod priority class for the non-voting nodes.

Value type	Example
string	`high priority`

`replsets.nonvoting.hookScript.configMapRef.name`

The name of the ConfigMap that contains a custom hook script that you can apply to the non-voting replica set members before the main process.

Value type	Example
string	`hookscript-configmap`

`replsets.nonvoting.hookScript.script`

The inline hook script that you can apply to the non-voting replica set members before the main process.

Value type	Example
string	`#!/usr/bin/env bash   echo "Hookscript started"`

`replsets.nonvoting.annotations`

The [Kubernetes annotations](#) metadata for the non-voting nodes.

Value type	Example
string	`iam.amazonaws.com/role: role-arn`

`replsets.nonvoting.labels`

The [Kubernetes affinity labels](#) for the non-voting nodes.

Value type	Example
label	`rack: rack-22`

`replsets.nonvoting.nodeSelector`

The [Kubernetes nodeSelector](#) affinity constraint for the non-voting nodes.

Value type	Example
label	`disktype: ssd`

`replsets.nonvoting.podDisruptionBudget.maxUnavailable`

The [Kubernetes Pod distribution budget](#) limit specifying the maximum value for unavailable Pods among non-voting nodes.

Value type	Example
int	`1`

`replsets.nonvoting.podDisruptionBudget.minAvailable`

The [Kubernetes Pod distribution budget](#) limit specifying the minimum value for available Pods among non-voting nodes.

Value type	Example
int	`1`

`replsets.nonvoting.resources.limits.cpu`

[Kubernetes CPU limit](#) for MongoDB container.

Value type	Example
string	`300m`

`replsets.nonvoting.resources.limits.memory`

[Kubernetes Memory limit](#) for MongoDB container.

Value type	Example
string	`0.5G`

`replsets.nonvoting.resources.requests.cpu`

The Kubernetes CPU requests for MongoDB container.

Value type	Example
string	`300m`

`replsets.nonvoting.resources.requests.memory`

The Kubernetes Memory requests for MongoDB container.

Value type	Example
string	`0.5G`

`replsets.nonvoting.volumeSpec.emptyDir`

The Kubernetes emptyDir volume, i.e. the directory which will be created on a node, and will be accessible to the MongoDB Pod containers.

Value type	Example
string	`{}`

`replsets.nonvoting.volumeSpec.hostPath.path`

Kubernetes hostPath volume, i.e. the file or directory of a node that will be accessible to the MongoDB Pod containers.

Value type	Example
string	`/data`

`replsets.nonvoting.volumeSpec.hostPath.type`

The Kubernetes hostPath volume type.

Value type	Example
string	`Directory`

`replsets.nonvoting.volumeSpec.persistentVolumeClaim.annotations`

The Kubernetes annotations metadata for Persistent Volume Claim.

Value type	Example
string	`service.beta.kubernetes.io/aws-load-balancer-backend-protocol: http`

`replsets.nonvoting.volumeSpec.persistentVolumeClaim.labels`

The Kubernetes labels metadata for Persistent Volume Claim.

Value type	Example
string	`rack: rack-22`

`replsets.nonvoting.volumeSpec.persistentVolumeClaim.storageClassName`

The [Kubernetes Storage Class](#) to use with the MongoDB container [Persistent Volume Claim](#) for the non-voting nodes. Use Storage Class with XFS as the default filesystem if possible, [for better MongoDB performance](https://dba.stackexchange.com/questions/190578/is-xfs-still-the-best-choice-for-mongodb.

Value type	Example
string	`standard`

`replsets.nonvoting.volumeSpec.persistentVolumeClaim.volumeAttributesClassName`

The name of the [volumeAttributesClassName](#) used to modify the provisioned PVCs bound to MongoDB Pods. See [How to configure VolumeAttributesClass for Persistent Volumes](#) to learn more.

Value type	Example
string	`silver`

`replsets.nonvoting.volumeSpec.persistentVolumeClaim.accessModes`

The [Kubernetes Persistent Volume](#) access modes for the MongoDB container for the non-voting nodes.

Value type	Example
array	`[ "ReadWriteOnce" ]`

`replsets.nonvoting.volumeSpec.persistentVolumeClaim.resources.requests.storage`

The [Kubernetes Persistent Volume](#) size for the MongoDB container for the non-voting nodes.

Value type	Example
string	`3Gi`

`replsets.hidden.enabled`

Enable or disable creation of [Replica Set hidden instances](#) within the cluster.

Value type	Example
boolean	`false`

`replsets.hidden.size`

The number of [Replica Set hidden instances](#) within the cluster.

Value type	Example
int	`1`

`replsets.hidden.podSecurityContext`

A custom [Kubernetes Security Context for a Pod](#) to be used instead of the default one.

Value type	Example

subdoc	`{}`

`replsets.hidden.containerSecurityContext`

A custom [Kubernetes Security Context for a Container] to be used instead of the default one.

Value type	Example
subdoc	`{}`

`replsets.hidden.affinity.antiAffinityTopologyKey`

The [Kubernetes topologyKey] node affinity constraint for the hidden nodes.

Value type	Example
string	`kubernetes.io/hostname`

`replsets.hidden.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	

`replsets.hidden.tolerations.key`

The [Kubernetes Pod tolerations] key for the hidden nodes.

Value type	Example
string	`node.alpha.kubernetes.io/unreachable`

`replsets.hidden.tolerations.operator`

The [Kubernetes Pod tolerations] operator for the hidden nodes.

Value type	Example
string	`Exists`

`replsets.hidden.tolerations.effect`

The [Kubernetes Pod tolerations] effect for the hidden nodes.

Value type	Example
string	`NoExecute`

`replsets.hidden.tolerations.tolerationSeconds`

The [Kubernetes Pod tolerations] time limit for the hidden nodes.

Value type	Example
int	`6000`

`replsets.hidden.priorityClassName`

The [Kuberentes Pod priority class](#) for the hidden nodes.

Value type	Example
string	`high priority`

`replsets.hidden.hookScript.configMapRef.name`

The name of the ConfigMap that contains a custom hook script that you can apply to the hidden replica set members before the main process.

Value type	Example
string	`hookscript-configmap`

`replsets.hidden. hookScript.script`

The inline hook script that you can apply to the hidden replica set members before the main process.

Value type	Example
string	`#!/usr/bin/env bash   echo "Hookscript started"`

`replsets.hidden.annotations`

The [Kubernetes annotations](#) metadata for the hidden nodes.

Value type	Example
string	`iam.amazonaws.com/role: role-arn`

`replsets.hidden.labels`

The [Kubernetes affinity labels](#) for the hidden nodes.

Value type	Example
label	`rack: rack-22`

`replsets.hidden.nodeSelector`

The [Kubernetes nodeSelector](#) affinity constraint for the hidden nodes.

Value type	Example
label	`disktype: ssd`

`replsets.hidden.podDisruptionBudget.maxUnavailable`

The [Kubernetes Pod distribution budget](#) limit specifying the maximum value for unavailable Pods among hidden nodes.

Value type	Example
int	`1`

`replsets.hidden.podDisruptionBudget.minAvailable`

The [Kubernetes Pod distribution budget](#) limit specifying the minimum value for available Pods among hidden nodes.

Value type	Example
int	`1`

`replsets.hidden.resources.limits.cpu`

[Kubernetes CPU limit](#) for MongoDB container.

Value type	Example
string	`300m`

`replsets.hidden.resources.limits.memory`

[Kubernetes Memory limit](#) for MongoDB container.

Value type	Example
string	`0.5G`

`replsets.hidden.volumeSpec.emptyDir`

The [Kubernetes emptyDir volume](#), i.e. the directory which will be created on a node, and will be accessible to the MongoDB Pod containers.

Value type	Example
string	`{}`

`replsets.hidden.volumeSpec.hostPath.path`

[Kubernetes hostPath volume](#), i.e. the file or directory of a node that will be accessible to the MongoDB Pod containers.

Value type	Example
string	`/data`

`replsets.hidden.volumeSpec.hostPath.type`

The [Kubernetes hostPath volume type](#).

Value type	Example
string	`Directory`

`replsets.hidden.volumeSpec.persistentVolumeClaim.annotations`

The [Kubernetes annotations](#) metadata for [Persistent Volume Claim](#).

Value type	Example
string	`service.beta.kubernetes.io/aws-load-balancer-backend-protocol: http`

`replsets.hidden.volumeSpec.persistentVolumeClaim.labels`

The [Kubernetes labels]() metadata for [Persistent Volume Claim]().

Value type	Example
string	`rack: rack-22`

`replsets.hidden.volumeSpec.persistentVolumeClaim.storageClassName`

The [Kubernetes Storage Class]() to use with the MongoDB container [Persistent Volume Claim]() for the hidden nodes. Use Storage Class with XFS as the default filesystem if possible, [for better MongoDB performance]().

Value type	Example
string	`standard`

`replsets.hidden.volumeSpec.persistentVolumeClaim.volumeAttributesClassName`

The name of the [volumeAttributesClassName]() used to modify the provisioned PVCs bound to MongoDB Pods. See [How to configure VolumeAttributesClass for Persistent Volumes]() to learn more.

Value type	Example
string	`silver`

`replsets.hidden.volumeSpec.persistentVolumeClaim.accessModes`

The [Kubernetes Persistent Volume]() access modes for the MongoDB container for the hidden nodes.

Value type	Example
array	`[ "ReadWriteOnce" ]`

`replsets.hidden.volumeSpec.persistentVolumeClaim.resources.requests.storage`

The [Kubernetes Persistent Volume]() size for the MongoDB container for the hidden nodes.

Value type	Example
string	`3Gi`

`replsets.arbiter.enabled`

Enable or disable creation of [Replica Set Arbiter]() nodes within the cluster.

Value type	Example
boolean	`false`

`replsets.arbiter.size`

The number of [Replica Set Arbiter](#) instances within the cluster.

Value type	Example
int	`1`

`replsets.arbiter.affinity.antiAffinityTopologyKey`

The [Kubernetes topologyKey](#) node affinity constraint for the Arbiter.

Value type	Example
string	`kubernetes.io/hostname`

`replsets.arbiter.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	

`replsets.arbiter.tolerations.key`

The [Kubernetes Pod tolerations](#) key for the Arbiter nodes.

Value type	Example
string	`node.alpha.kubernetes.io/unreachable`

`replsets.arbiter.tolerations.operator`

The [Kubernetes Pod tolerations](#) operator for the Arbiter nodes.

Value type	Example
string	`Exists`

`replsets.arbiter.tolerations.effect`

The [Kubernetes Pod tolerations](#) effect for the Arbiter nodes.

Value type	Example
string	`NoExecute`

`replsets.arbiter.tolerations.tolerationSeconds`

The [Kubernetes Pod tolerations](#) time limit for the Arbiter nodes.

Value type	Example
int	`6000`

`replsets.arbiter.priorityClassName`

The [Kuberentes Pod priority class](#) for the Arbiter nodes.

Value type	Example
string	`high priority`

`replsets.arbiter.hookScript.configMapRef.name`

The name of the ConfigMap that contains a custom hook script that you can apply to the arbiter replica set members before the main process.

Value type	Example
string	`hookscript-configmap`

`replsets.arbiter.hookScript.script`

The inline hook script that you can apply to the arbiter replica set members before the main process.

Value type	Example
string	`#!/usr/bin/env bash   echo "Hookscript started"`

`replsets.arbiter.annotations`

The [Kubernetes annotations](#) metadata for the Arbiter nodes.

Value type	Example
string	`iam.amazonaws.com/role: role-arn`

`replsets.arbiter.labels`

The [Kubernetes affinity labels](#) for the Arbiter nodes.

Value type	Example
label	`rack: rack-22`

`replsets.arbiter.nodeSelector`

The [Kubernetes nodeSelector](#) affinity constraint for the Arbiter nodes.

Value type	Example
label	`disktype: ssd`

`replsets.resources.limits.cpu`

[Kubernetes CPU limit](#) for MongoDB container.

Value type	Example
string	`300m`

`replsets.resources.limits.memory`

[Kubernetes Memory limit](#) for MongoDB container.

Value type	Example
string	`0.5G`

`replsets.resources.requests.cpu`

The [Kubernetes CPU requests](#) for MongoDB container.

Value type	Example
string	`300m`

`replsets.resources.requests.memory`

The [Kubernetes Memory requests](#) for MongoDB container.

Value type	Example
string	`0.5G`

`replsets.volumeSpec.emptyDir`

The [Kubernetes emptyDir volume](#), i.e. the directory which will be created on a node, and will be accessible to the MongoDB Pod containers.

Value type	Example
string	`{}`

`replsets.volumeSpec.hostPath.path`

[Kubernetes hostPath volume](#), i.e. the file or directory of a node that will be accessible to the MongoDB Pod containers.

Value type	Example
string	`/data`

`replsets.volumeSpec.hostPath.type`

The [Kubernetes hostPath volume type](#).

Value type	Example
string	`Directory`

`replsets.volumeSpec.persistentVolumeClaim.annotations`

The [Kubernetes annotations](#) metadata for [Persistent Volume Claim](#).

Value type	Example
string	`service.beta.kubernetes.io/aws-load-balancer-backend-protocol: http`

`replsets.volumeSpec.persistentVolumeClaim.labels`

The Kubernetes labels metadata for Persistent Volume Claim.

Value type	Example
string	`rack: rack-22`

`replsets.volumeSpec.persistentVolumeClaim.storageClassName`

The Kubernetes Storage Class to use with the MongoDB container Persistent Volume Claim. Use Storage Class with XFS as the default filesystem if possible, for better MongoDB performance.

Value type	Example
string	`standard`

`replsets.volumeSpec.persistentVolumeClaim.volumeAttributesClassName`

The name of the volumeAttributesClassName used to modify the provisioned PVCs bound to MongoDB Pods. See How to configure VolumeAttributesClass for Persistent Volumes to learn more.

Value type	Example
string	`silver`

`replsets.volumeSpec.persistentVolumeClaim.accessModes`

The Kubernetes Persistent Volume access modes for the MongoDB container.

Value type	Example
array	`[ "ReadWriteOnce" ]`

`replsets.volumeSpec.persistentVolumeClaim.resources.requests.storage`

The Kubernetes Persistent Volume size for the MongoDB container.

Value type	Example
string	`3Gi`

`replsets.hostAliases.ip`

The IP address for Kubernetes host aliases for replica set Pods.

Value type	Example
string	`"10.10.0.2"`

`replsets.hostAliases.hostnames`

Hostnames for Kubernetes host aliases for replica set Pods.

Value type	Example
subdoc	

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 MongoDB with PMM.

Value type	Example
boolean	`false`

`pmm.image`

PMM Client Docker image to use.

Value type	Example
string	`percona/pmm-client:2.44.1-1`

`pmm.serverHost`

Address of the PMM Server to collect data from the Cluster.

Value type	Example
string	`monitoring-service`

`pmm.containerSecurityContext`

A custom Kubernetes Security Context for a Container to be used instead of the default one.

Value type	Example
subdoc	`{}`

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
string	`mongo-cluster`

`pmm.mongodParams`

Additional parameters which will be passed to the pmm-admin add mongodb command for `mongod` Pods.

Value type	Example

string	`--environment=DEV-ENV --custom-labels=DEV-ENV`

`pmm.mongosParams`

Additional parameters which will be passed to the pmm-admin add mongodb command for `mongos` Pods.

Value type	Example
string	`--environment=DEV-ENV --custom-labels=DEV-ENV`

`pmm.resources.requests.cpu`

The Kubernetes CPU requests for PMM Client container.

Value type	Example
string	`300m`

`pmm.resources.requests.memory`

The Kubernetes Memory requests for PMM Client container.

Value type	Example
string	`150M`

`pmm.resources.limits.cpu`

Kubernetes CPU limit for PMM Client container.

Value type	Example
string	`400m`

`pmm.resources.limits.memory`

Kubernetes Memory limit for PMM Client container.

Value type	Example
string	`256M`

Sharding Section

The `sharding` section in the deploy/cr.yaml file contains configuration options for Percona Server for MondoDB sharding.

`sharding.enabled`

Enables or disables Percona Server for MondoDB sharding.

Value type	Example
boolean	`true`

`sharding.configsvrReplSet.size`

The number of [Config Server instances](#) within the cluster.

Value type	Example
int	`3`

`sharding.configsvrReplSet.terminationGracePeriodSeconds`

The amount of seconds Kubernetes will wait for a clean config server Pods termination.

Value type	Example
int	`300`

`sharding.configsvrReplSet.serviceAccountName`

Name of the separate privileged service account for Config Server Pods.

Value type	Example
string	`default`

`sharding.configsvrReplSet.topologySpreadConstraints.labelSelector.matchLabels`

The label selector for the [Kubernetes Pod Topology Spread Constraints](#).

Value type	Example
label	`app.kubernetes.io/name: percona-server-mongodb`

`sharding.configsvrReplSet.topologySpreadConstraints.maxSkew`

The degree to which Pods may be unevenly distributed under the [Kubernetes Pod Topology Spread Constraints](#).

Value type	Example
int	`1`

`sharding.configsvrReplSet.topologySpreadConstraints.topologyKey`

The key of node labels for the [Kubernetes Pod Topology Spread Constraints](#).

Value type	Example
string	`kubernetes.io/hostname`

`sharding.configsvrReplSet.topologySpreadConstraints.whenUnsatisfiable`

What to do with a Pod if it doesn’t satisfy the [Kubernetes Pod Topology Spread Constraints](#).

Value type	Example
string	`DoNotSchedule`

`sharding.configsvrReplSet.externalNodes.host`

The URL or IP address of the external config server instance.

Value type	Example
string	`34.124.76.90`

`sharding.configsvrReplSet.externalNodes.port`

The port number of the external config server instance.

Value type	Example
string	`27017`

`sharding.configsvrReplSet.externalNodes.votes`

The number of votes of the external config server instance.

Value type	Example
string	`0`

`sharding.configsvrReplSet.externalNodes.priority`

The priority of the external config server instance.

Value type	Example
string	`0`

`sharding.configsvrReplSet.configuration`

Custom configuration options for Config Servers. Please refer to the official manual for the full list of options.

Value type	Example

string	<pre>|
	operationProfiling:
	  mode: slowOp
	systemLog:
	  verbosity: 1</pre>

`sharding.configsvrReplSet.env.name`

The name of a custom environment variable for Config Server mongod containers (available since Operator 1.22.0). See Define custom environment variables.

Value type	Example
string	`MY_ENV`

`sharding.configsvrReplSet.env.value`

The value of a custom environment variable for Config Server mongod containers (available since Operator 1.22.0). See Define custom environment variables.

Value type	Example

S string	`MY_VALUE`

`sharding.configsvrReplSet.envFrom.ConfigMapRef.name`

The name of a ConfigMap from where environment variables will be loaded for Config Server mongod containers (available since Operator 1.22.0). See Define custom environment variables.

Value type	Example
S	`my-configmap`

`sharding.configsvrReplSet.envFrom.secretRef.name`

The name of a Secret from where environment variables will be loaded for Config Server mongod containers (available since Operator 1.22.0). See Define custom environment variables.

Value type	Example
S	`my-secret`

`sharding.configsvrReplSet.livenessProbe.failureThreshold`

Number of consecutive unsuccessful tries of the liveness probe to be undertaken before giving up.

Value type	Example
1 int	`4`

`sharding.configsvrReplSet.livenessProbe.initialDelaySeconds`

Number of seconds to wait after the container start before initiating the liveness probe.

Value type	Example
1 int	`60`

`sharding.configsvrReplSet.livenessProbe.periodSeconds`

How often to perform a liveness probe (in seconds).

Value type	Example
1 int	`30`

`sharding.configsvrReplSet.livenessProbe.timeoutSeconds`

Number of seconds after which the liveness probe times out.

Value type	Example
1 int	`10`

`sharding.configsvrReplSet.livenessProbe.startupDelaySeconds`

Time after which the liveness probe is failed if the MongoDB instance didn't finish its full startup yet.

Value type	Example
int	`7200`

`sharding.configsvrReplSet.readinessProbe.failureThreshold`

Number of consecutive unsuccessful tries of the readiness probe to be undertaken before giving up.

Value type	Example
int	`3`

`sharding.configsvrReplSet.readinessProbe.initialDelaySeconds`

Number of seconds to wait after the container start before initiating the readiness probe.

Value type	Example
int	`10`

`sharding.configsvrReplSet.readinessProbe.periodSeconds`

How often to perform a readiness probe (in seconds).

Value type	Example
int	`3`

`sharding.configsvrReplSet.readinessProbe.successThreshold`

Minimum consecutive successes for the readiness probe to be considered successful after having failed.

Value type	Example
int	`1`

`sharding.configsvrReplSet.readinessProbe.timeoutSeconds`

Number of seconds after which the readiness probe times out.

Value type	Example
int	`2`

`sharding.configsvrReplSet.containerSecurityContext`

A custom Kubernetes Security Context for a Container to be used instead of the default one.

Value type	Example
subdoc	`privileged: false`

`sharding.configsvrReplSet.podSecurityContext`

A custom Kubernetes Security Context for a Pod to be used instead of the default one.

Value type	Example
subdoc	runAsUser: 1001 runAsGroup: 1001 supplementalGroups: [1001]

`sharding.configsvrReplSet.runtimeClassName`

Name of the Kubernetes Runtime Class for Config Server Pods.

Value type	Example
string	`image-rc`

`sharding.configsvrReplSet.hookScript.configMapRef.name`

The name of the ConfigMap that contains a custom hook script that you can apply to the config server replica set Pods before the main process.

Value type	Example
string	`hookscript-configmap`

`sharding.configsvrReplSet.hookScript.script`

The inline hook script that you can apply to the config server replica set Pods before the main process.

Value type	Example
string	`#!/usr/bin/env bash   echo "Hookscript started"`

`sharding.configsvrReplSet.sidecars.image`

Image for the custom sidecar container for Config Server Pods.

Value type	Example
string	`busybox`

`sharding.configsvrReplSet.sidecars.command`

Command for the custom sidecar container for Config Server Pods.

Value type	Example
array	`["/bin/sh"]`

`sharding.configsvrReplSet.sidecars.args`

Command arguments for the custom sidecar container for Config Server Pods.

Value type	Example
array	`["-c", "while true; do echo echo $(date -u) 'test' >> /dev/null; sleep 5;done"]`

`sharding.configsvrReplSet.sidecars.name`

Name of the custom sidecar container for Config Server Pods.

Value type	Example
S string	`rs-sidecar-1`

`sharding.configsvrReplSet.sidecarVolumes.name`

Name of the custom sidecar container volume for Config Server Pods.

Value type	Example
S string	`sidecar-config`

`sharding.configsvrReplSet.sidecarVolumes.nfs.server`

The hostname of the NFS server that will provide remote filesystem to the custom sidecar container volume for Config Server Pods.

Value type	Example
S string	`nfs-service.storage.svc.cluster.local`

`sharding.configsvrReplSet.sidecarVolumes.nfs.path`

The path on the NFS server that will be provided as a remote filesystem to the custom sidecar container volume for Config Server Pods.

Value type	Example
S string	`/psmdb-some-name-rs0`

`sharding.configsvrReplSet.limits.cpu`

Kubernetes CPU limit for Config Server container.

Value type	Example
S string	`300m`

`sharding.configsvrReplSet.limits.memory`

Kubernetes Memory limit for Config Server container.

Value type	Example
S string	`0.5G`

`sharding.configsvrReplSet.resources.requests.cpu`

The Kubernetes CPU requests for Config Server container.

Value type	Example
S string	`300m`

`sharding.configsvrReplSet.requests.memory`

The [Kubernetes Memory requests](#) for Config Server container.

Value type	Example
string	`0.5G`

`sharding.configsvrReplSet.expose.enabled`

Enable or disable exposing [Config Server](#) nodes with dedicated IP addresses.

Value type	Example
boolean	`false`

`sharding.configsvrReplSet.expose.type`

The [IP address type](#) to be exposed.

Value type	Example
string	`ClusterIP`

`sharding.configsvrReplSet.expose.loadBalancerClass`

Define the implementation of the load balancer you want to use. This setting enables you to select a custom or specific load balancer class instead of the default one provided by the cloud provider.

Value type	Example
string	`eks.amazonaws.com/nlb`

`sharding.configsvrReplSet.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`

`sharding.configsvrReplSet.expose.annotations`

The [Kubernetes annotations](#) metadata for the Config Server daemon.

Value type	Example
string	`service.beta.kubernetes.io/aws-load-balancer-backend-protocol: http`

`sharding.configsvrReplSet.expose.labels`

The [Kubernetes labels](#) for the Config Server Service.

Value type	Example
string	`rack: rack-22`

`sharding.configsvrReplSet.expose.internalTrafficPolicy`

Specifies whether Service for config servers should [route internal traffic to cluster-wide or to node-local endpoints](#) (it can influence the load balancing effectiveness).

Value type	Example
boolean	`Local`

`sharding.configsvrReplSet.expose.externalTrafficPolicy`

Specifies whether Service for config servers [should route external traffic](#) to cluster-wide (`Cluster`) or to node-local (`Local`) endpoints. It [can influence the load balancing effectiveness](#).

Value type	Example
string	`Local`

Service protocol configuration

Starting from Operator version 1.22.0, the Operator automatically sets `appProtocol: mongo` on Service ports for config servers. This is required for service mesh implementations (such as Istio) because MongoDB uses a server-first protocol, which breaks mTLS without explicit protocol specification. The `appProtocol` field is set automatically and cannot be configured manually.

`sharding.configsvrReplSet.volumeSpec.emptyDir`

The [Kubernetes emptyDir volume](#), i.e. the directory which will be created on a node, and will be accessible to the Config Server Pod containers.

Value type	Example
string	`{}`

`sharding.configsvrReplSet.volumeSpec.hostPath.path`

[Kubernetes hostPath volume](#), i.e. the file or directory of a node that will be accessible to the Config Server Pod containers.

Value type	Example
string	`/data`

`sharding.configsvrReplSet.volumeSpec.hostPath.type`

The [Kubernetes hostPath volume type](#).

Value type	Example
string	`Directory`

`sharding.configsvrReplSet.volumeSpec.persistentVolumeClaim.annotations`

The [Kubernetes annotations](#) metadata for [Persistent Volume Claim](#).

Value type	Example
string	`service.beta.kubernetes.io/aws-load-balancer-backend-protocol: http`

`sharding.configsvrReplSet.volumeSpec.persistentVolumeClaim.labels`

The Kubernetes labels metadata for Persistent Volume Claim.

Value type	Example
string	`rack: rack-22`

`sharding.configsvrReplSet.volumeSpec.persistentVolumeClaim.storageClassName`

The Kubernetes Storage Class to use with the Config Server container Persistent Volume Claim. Use Storage Class with XFS as the default filesystem if possible, for better MongoDB performance.

Value type	Example
string	`standard`

`sharding.configsvrReplSet.volumeSpec.persistentVolumeClaim.volumeAttributesClassName`

The name of the volumeAttributesClassName used to modify the provisioned PVCs bound to MongoDB Pods. See How to configure VolumeAttributesClass for Persistent Volumes to learn more.

Value type	Example
string	`silver`

`sharding.configsvrReplSet.volumeSpec.persistentVolumeClaim.accessModes`

The Kubernetes Persistent Volume access modes for the Config Server container.

Value type	Example
array	`[ "ReadWriteOnce" ]`

`sharding.configsvrReplSet.volumeSpec.persistentVolumeClaim.resources.requests.storage`

The Kubernetes Persistent Volume size for the Config Server container.

Value type	Example
string	`3Gi`

`sharding.configsvrReplSet.hostAliases.ip`

The IP address for Kubernetes host aliases for replica set Pods.

Value type	Example
string	`"10.10.0.2"`

`sharding.configsvrReplSet.hostAliases.hostnames`

Hostnames for Kubernetes host aliases for config server Pods.

Value type	Example

subdoc

`sharding.configsvrReplSet.splitHorizons.CFGREPLICASET-POD-NAME.external`

External URI for Split-horizon for Config Server replica set Pods of the exposed cluster.

Value type	Example
string	`cfg-0.mycluster.xyz`

`sharding.configsvrReplSet.splitHorizons.CFGREPLICASET-POD-NAME.external-2`

External URI for Split-horizon for Config Server replica set Pods of the exposed cluster.

Value type	Example
string	`cfg-0.mycluster2.xyz`

`sharding.mongos.size`

The number of mongos instances within the cluster.

Value type	Example
int	`3`

`sharding.mongos.terminationGracePeriodSeconds`

The amount of seconds Kubernetes will wait for a clean mongos Pods termination.

Value type	Example
int	`300`

`sharding.mongos.serviceAccountName`

Name of the separate privileged service account for mongos Pods.

Value type	Example
string	`default`

`sharding.mongos.topologySpreadConstraints.labelSelector.matchLabels`

The label selector for the Kubernetes Pod Topology Spread Constraints.

Value type	Example
label	`app.kubernetes.io/name: percona-server-mongodb`

`sharding.mongos.topologySpreadConstraints.maxSkew`

The degree to which Pods may be unevenly distributed under the Kubernetes Pod Topology Spread Constraints.

Value type	Example

int	`1`

`sharding.mongos.topologySpreadConstraints.topologyKey`

The key of node labels for the [Kubernetes Pod Topology Spread Constraints](#).

Value type	Example
string	`kubernetes.io/hostname`

`sharding.mongos.topologySpreadConstraints.whenUnsatisfiable`

What to do with a Pod if it doesn't satisfy the [Kubernetes Pod Topology Spread Constraints](#).

Value type	Example
string	`DoNotSchedule`

`sharding.mongos.configuration`

Custom configuration options for mongos. Please refer to the [official manual](#) for the full list of options.

Value type	Example

string	`|`
	`systemLog:`
	`  verbosity: 1`

`sharding.mongos.env.name`

The name of a custom environment variable for mongos containers (available since Operator 1.22.0). See [Define custom environment variables](#).

Value type	Example
string	`MY_ENV`

`sharding.mongos.env.value`

The value of a custom environment variable for mongos containers (available since Operator 1.22.0). See [Define custom environment variables](#).

Value type	Example
string	`MY_VALUE`

`sharding.mongos.envFrom.configMapRef.name`

The name of a ConfigMap from where environment variables will be loaded for mongos containers (available since Operator 1.22.0). See [Define custom environment variables](#).

Value type	Example
	`my-configmap`

`sharding.mongos.envFrom.secretRef.name`

The name of a Secret from where environment variables will be loaded for mongos containers (available since Operator 1.22.0). See Define custom environment variables.

Value type	Example
S	`my-secret`

`sharding.mongos.affinity.antiAffinityTopologyKey`

The Kubernetes topologyKey node affinity constraint for mongos.

Value type	Example
S string	`kubernetes.io/hostname`

`sharding.mongos.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	

`sharding.mongos.tolerations.key`

The Kubernetes Pod tolerations key for mongos instances.

Value type	Example
S string	`node.alpha.kubernetes.io/unreachable`

`sharding.mongos.tolerations.operator`

The Kubernetes Pod tolerations operator for mongos instances.

Value type	Example
S string	`Exists`

`sharding.mongos.tolerations.effect`

The Kubernetes Pod tolerations effect for mongos instances.

Value type	Example
S string	`NoExecute`

`sharding.mongos.tolerations.tolerationSeconds`

The Kubernetes Pod tolerations time limit for mongos instances.

Value type	Example
I int	`6000`

`sharding.mongos.priorityClassName`

The [Kuberentes Pod priority class](#) for mongos instances.

Value type	Example
string	`high priority`

`sharding.mongos.annotations`

The [Kubernetes annotations](#) metadata for the mongos instances.

Value type	Example
string	`iam.amazonaws.com/role: role-arn`

`sharding.mongos.labels`

The [Kubernetes affinity labels](#) for mongos instances.

Value type	Example
label	`rack: rack-22`

`sharding.mongos.nodeSelector`

The [Kubernetes nodeSelector](#) affinity constraint for mongos instances.

Value type	Example
label	`disktype: ssd`

`sharding.mongos.livenessProbe.failureThreshold`

Number of consecutive unsuccessful tries of the [liveness probe](#) to be undertaken before giving up.

Value type	Example
int	`4`

`sharding.mongos.livenessProbe.initialDelaySeconds`

Number of seconds to wait after the container start before initiating the [liveness probe](#).

Value type	Example
int	`60`

`sharding.mongos.livenessProbe.periodSeconds`

How often to perform a [liveness probe](#) (in seconds).

Value type	Example
int	`30`

`sharding.mongos.livenessProbe.timeoutSeconds`

Number of seconds after which the [liveness probe](#) times out.

Value type	Example
int	`10`

`sharding.mongos.livenessProbe.startupDelaySeconds`

Time after which the liveness probe is failed if the MongoDB instance didn't finish its full startup yet.

Value type	Example
int	`7200`

`sharding.mongos.readinessProbe.failureThreshold`

Number of consecutive unsuccessful tries of the [readiness probe](#) to be undertaken before giving up.

Value type	Example
int	`3`

`sharding.mongos.readinessProbe.initialDelaySeconds`

Number of seconds to wait after the container start before initiating the [readiness probe](#).

Value type	Example
int	`10`

`sharding.mongos.readinessProbe.periodSeconds`

How often to perform a [readiness probe](#) (in seconds).

Value type	Example
int	`3`

`sharding.mongos.readinessProbe.successThreshold`

Minimum consecutive successes for the [readiness probe](#) to be considered successful after having failed.

Value type	Example
int	`1`

`sharding.mongos.readinessProbe.timeoutSeconds`

Number of seconds after which the [readiness probe](#) times out.

Value type	Example
int	`2`

`sharding.mongos.containerSecurityContext`

A custom Kubernetes Security Context for a Container to be used instead of the default one.

Value type	Example
subdoc	`privileged: false`

`sharding.mongos.podSecurityContext`

A custom Kubernetes Security Context for a Pod to be used instead of the default one.

Value type	Example

subdoc	`runAsUser: 1001`
	`runAsGroup: 1001`
	`supplementalGroups: [1001]`

`sharding.mongos.runtimeClassName`

Name of the Kubernetes Runtime Class for mongos Pods.

Value type	Example
string	`image-rc`

`sharding.mongos.hookScript.configMapRef.name`

The name of the ConfigMap that contains a custom hook script that you can apply to the `mongos` Pods before the main process.

Value type	Example
string	`hookscript-configmap`

`sharding.mongos.hookScript.script`

The inline hook script that you can apply to the `mongos` Pods before the main process.

Value type	Example
string	`#!/usr/bin/env bash   echo "Hookscript started"`

`sharding.mongos.sidecars.image`

Image for the custom sidecar container for mongos Pods.

Value type	Example
string	`busybox`

`sharding.mongos.sidecars.command`

Command for the custom sidecar container for mongos Pods.

Value type	Example
array	`["/bin/sh"]`

`sharding.mongos.sidecars.args`

Command arguments for the custom sidecar container for mongos Pods.

Value type	Example
array	`["-c", "while true; do echo echo $(date -u) 'test' >> /dev/null; sleep 5;done"]`

`sharding.mongos.sidecars.name`

Name of the custom sidecar container for mongos Pods.

Value type	Example
string	`rs-sidecar-1`

`sharding.mongos.limits.cpu`

Kubernetes CPU limit for mongos container.

Value type	Example
string	`300m`

`sharding.mongos.limits.memory`

Kubernetes Memory limit for mongos container.

Value type	Example
string	`0.5G`

`sharding.mongos.resources.requests.cpu`

The Kubernetes CPU requests for mongos container.

Value type	Example
string	`300m`

`sharding.mongos.requests.memory`

The Kubernetes Memory requests for mongos container.

Value type	Example
string	`0.5G`

`sharding.mongos.expose.type`

The IP address type to be exposed.

Value type	Example
string	`ClusterIP`

`sharding.mongos.expose.servicePerPod`

If set to `true`, a separate ClusterIP Service is created for each mongos instance.

Value type	Example
boolean	`true`

`sharding.mongos.expose.loadBalancerClass`

Define the implementation of the load balancer you want to use. This setting enables you to select a custom or specific load balancer class instead of the default one provided by the cloud provider.

Value type	Example
string	`eks.amazonaws.com/nlb`

`sharding.mongos.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`

`sharding.mongos.expose.annotations`

The Kubernetes annotations metadata for the MongoDB mongos daemon.

Value type	Example
string	`service.beta.kubernetes.io/aws-load-balancer-backend-protocol: http`

`sharding.mongos.expose.labels`

The Kubernetes labels for the MongoDB mongos Service.

Value type	Example
string	`rack: rack-22`

`sharding.mongos.expose.nodePort`

The Node port number to be allocated for the MongoDB mongos Service when the `sharding.mongos.expose.type` is set to the `NodePort`, and `sharding.mongos.expose.servicePerPod` is not turned on.

Value type	Example
int	`32017`

`sharding.mongos.internalTrafficPolicy`

Specifies whether Services for the mongos instances should route internal traffic to cluster-wide or to node-local endpoints (it can influence the load balancing effectiveness).

Value type	Example
boolean	`Local`

`sharding.mongos.externalTrafficPolicy`

Specifies whether Service for the mongos instances [should route external traffic](#) to cluster-wide (`Cluster`) or to node-local (`Local`) endpoints. It [can influence the load balancing effectiveness](#).

Value type	Example
string	`Local`

> **Service protocol configuration**
>
> Starting from Operator version 1.22.0, the Operator automatically sets `appProtocol: mongo` on Service ports for mongos instances. This is required for service mesh implementations (such as Istio) because MongoDB uses a server-first protocol, which breaks mTLS without explicit protocol specification. The `appProtocol` field is set automatically and cannot be configured manually.

`sharding.mongos.hostAliases.ip`

The IP address for [Kubernetes host aliases](#) for mongos Pods.

Value type	Example
string	`"10.10.0.2"`

`sharding.mongos.hostAliases.hostnames`

Hostnames for [Kubernetes host aliases](#) for mongos Pods.

Value type	Example
subdoc	

Roles section

The `roles` section in the [deploy/cr.yaml](#) file contains various configuration options [to configure custom MongoDB user roles via the Custom Resource](#).

`roles.role`

The [custom MongoDB role](#) name.

Value type	Example
string	`myClusterwideAdmin`

`roles.db`

Database in which you want to store the user-defined role.

Value type	Example
string	`admin

`roles.authenticationRestrictions.clientSource`

List of the IP addresses or CIDR blocks *from which* users assigned this role can connect. MongoDB servers reject connection requests from users with this role if the requests come from a client that is not present in this array.

Value type	Example
subdoc	`127.0.0.1`

`roles.authenticationRestrictions.serverAddress`

List of the IP addresses or CIDR blocks *to which* users assigned this role can connect. MongoDB servers reject connection requests from users with this role if the client requests to connect to a server that is not present in this array.

Value type	Example
subdoc	`127.0.0.1`

`roles.privileges.actions`

List of custom role actions that users granted this role can perform: For a list of accepted values, see Privilege Actions in the MongoDB Manual.

Value type	Example
subdoc	`addShard`

`roles.privileges.resource.db`

Database for which the custom role actions apply. An empty string ("") indicates that the privilege actions apply to all databases.

Value type	Example
string	`""`

`roles.privileges.resource.collection`

Collection for which the custom role actions apply. An empty string ("") indicates that the privilege actions apply to all of the database's collections.

Value type	Example
string	`""`

`roles.privileges.resource.cluster`

If true, the custom role actions apply to all databases and collections in the MongoDB deployment. False by default. If set to true, values for `roles.privileges.resource.db` and `roles.privileges.resource.collection` shouldn't be provided.

Value type	Example
boolean	`true`

`roles.roles`

An array of roles (with names of the role and the database) from which this role inherits privileges, if any.

Value type	Example

subdoc	`role: read` `db: admin`

Users section

The `users` section in the deploy/cr.yaml file contains various configuration options to configure custom MongoDB users via the Custom Resource.

`users.name`

The username of the MongoDB user.

Value type	Example
string	`my-user`

`users.db`

Database that the user authenticates against.

Value type	Example
string	`admin`

`users.passwordSecretRef.name`

Name of the secret that contains the user's password. If `passwordSecretRef` is not present, password will be generated automatically.

Value type	Example
string	`my-user-password`

`users.passwordSecretRef.key`

Key in the secret that corresponds to the value of the user's password (`password` by default).

Value type	Example
string	`password`

`users.roles.role.name`

Name of the MongoDB role assigned to the user. As built-in roles, so custom roles are supported.

Value type	Example
string	`clusterAdmin`

`users.roles.role.db`

Database that the MongoDB role applies to.

Value type	Example
string	`admin`

Backup Section

The `backup` section in the deploy/cr.yaml file contains the following configuration options for the regular Percona Server for MongoDB backups.

`backup.enabled`

Enables or disables making backups.

Value type	Example
boolean	`true`

`backup.image`

The Percona Server for MongoDB Docker image to use for the backup.

Value type	Example
string	`percona/percona-server-mongodb-operator:1.22.0-backup`

`backup.startingDeadlineSeconds`

The maximum time in seconds for a backup to start. The Operator compares the timestamp of the backup object against the current time. If the backup is not started within the set time, the Operator automatically marks it as “failed”.

If your cluster is starting or is not in the READY state when you start a backup, such backup will be marked as `failed`.

You can override this setting for a specific backup in the `deploy/backup/backup.yaml` configuration file.

Value type	Example
int	`300`

`backup.hookScript.configMapRef.name`

The name of the ConfigMap that contains a custom hook script that you can apply to the `pbm-agent` container before the main process.

Value type	Example
string	`hookscript-configmap`

`backup.hookScript.script`

The inline hook script that you can apply to tthe `pbm-agent` container before the main process.

Value type	Example
string	`#!/usr/bin/env bash   echo "Hookscript started"`

`backup.serviceAccountName`

Name of the separate privileged service account for backups; **service account for backups is not used by the Operator any more, and the option is deprecated since the Operator version 1.16.0**.

Value type	Example
string	`percona-server-mongodb-operator`

`backup.annotations`

The Kubernetes annotations metadata for the backup job.

Value type	Example
string	`sidecar.istio.io/inject: "false"`

`backup.resources.limits.cpu`

Kubernetes CPU limit for backups.

Value type	Example
string	`300m`

`backup.resources.limits.memory`

Kubernetes Memory limit for backups.

Value type	Example
string	`1.2G`

`backup.resources.requests.cpu`

The Kubernetes CPU requests for backups.

Value type	Example
string	`300m`

`backup.resources.requests.memory`

The Kubernetes Memory requests for backups.

Value type	Example
string	`1G`

`backup.containerSecurityContext`

A custom Kubernetes Security Context for a Container to be used instead of the default one.

Value type	Example
subdoc	`privileged: false`

`backup.storages.STORAGE-NAME.main`

Marks the storage as main. All other storages you define are added as profiles. The Operator saves backups to all storages but it saves oplog chunks for point-in-time recovery only to the main storage. You can define only one storage as main. Read more about multiple storages for backups.

Value type	Example
boolean	`true`

`backup.storages.STORAGE-NAME.type`

The cloud storage type used for backups. Only `s3`, `gcs`, `minio`, `azure`, and `filesystem` types are supported.

Value type	Example
string	`s3`

`backup.storages.STORAGE-NAME.s3.insecureSkipTLSVerify`

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.s3.credentialsSecret`

The Kubernetes secret for backups. It should contain `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` keys.

Value type	Example
string	`my-cluster-name-backup-s3`

`backup.storages.STORAGE-NAME.s3.bucket`

The Amazon S3 bucket name for backups.

Value type	Example
string	

`backup.storages.STORAGE-NAME.s3.prefix`

The path (sub-folder) to the backups inside the bucket.

Value type	Example
string	`""`

`backup.storages.STORAGE-NAME.s3.uploadPartSize`

The size of data chunks in bytes to be uploaded to the storage bucket (10 MiB by default).

Value type	Example
int	`10485760`

`backup.storages.STORAGE-NAME.s3.maxUploadParts`

The maximum number of data chunks to be uploaded to the storage bucket (10000 by default).

Value type	Example

int	`10000`

`backup.storages.STORAGE-NAME.s3.storageClass`

The storage class name of the S3 storage.

Value type	Example
string	`STANDARD`

`backup.storages.STORAGE-NAME.s3.retryer.numMaxRetries`

The maximum number of retries to upload data to S3 storage.

Value type	Example
int	`3`

`backup.storages.STORAGE-NAME.s3.retryer.minRetryDelay`

The minimum time in milliseconds to wait till the next retry.

Value type	Example
int	`10`

`backup.storages.STORAGE-NAME.s3.retryer.maxRetryDelay`

The maximum time in minutes to wait till the next retry.

Value type	Example
int	`5`

`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
string	`us-east-1`

`backup.storages.STORAGE-NAME.s3.endpointUrl`

The URL of the S3-compatible storage to be used. It is required for Minio storage and is not needed for the original Amazon S3 cloud.

Value type	Example
string	

`backup.storages.STORAGE-NAME.s3.serverSideEncryption.kmsKeyID`

The ID of the key stored in the AWS KMS used by the Operator for backups server-side encryption

Value type	Example

string	`""`

`backup.storages.STORAGE-NAME.s3.serverSideEncryption.sseAlgorithm`

The key management mode used for backups server-side encryption with the encryption keys stored in AWS KMS - `aws:kms` is the only supported value for now.

Value type	Example
string	`aws:kms`

`backup.storages.STORAGE-NAME.s3.serverSideEncryption.sseCustomerAlgorithm`

The key management mode for backups server-side encryption with customer-provided keys - `AES256` is the only supported value for now.

Value type	Example
string	`AES256`

`backup.storages.STORAGE-NAME.s3.serverSideEncryption.sseCustomerKey`

The locally-stored base64-encoded custom encryption key used by the Operator for backups server-side encryption on S3-compatible storages.

Value type	Example
string	`""`

`backup.storages.STORAGE-NAME.minio.insecureSkipTLSVerify`

Disables verification of the storage server TLS certificate. This enables data upload to S3-compatible storage with a self-issued certificate bypassing TLS verification for private. Read more about using custom CA certificates for S3 storage

Value type	Example
boolean	`true`

`backup.storages.STORAGE-NAME.minio.credentialsSecret`

The Kubernetes secret for backups. It contains the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` keys.

Value type	Example
string	`my-cluster-name-backup-minio`

`backup.storages.STORAGE-NAME.minio.bucket`

The bucket name for backups.

Value type	Example
string	

`backup.storages.STORAGE-NAME.minio.prefix`

The path (sub-folder) to the backups inside the bucket.

Value type	Example
string	`""`

`backup.storages.STORAGE-NAME.minio.region`

The region name of the bucket. This is where the bucket is located

Value type	Example
string	`us-east-1`

`backup.storages.STORAGE-NAME.minio.endpointUrl`

The endpoint URL of the S3-compatible storage service. This points to your storage service and is specific to your cloud provider.

Value type	Example
string	`minio.psmdb.svc.cluster.local:9000/minio/`

`backup.storages.STORAGE-NAME.minio.secure`

Defines whether to use HTTP or HTTPS protocol for communication with the S3 storage.

Value type	Example
boolean	`false`

`backup.storages.STORAGE-NAME.minio.caBundle.name`

The name of the Secret that stores custom TLS certificates for TLS communication with S3 storage. See Configure TLS verification with custom certificates for S3 storage for more information.

Value type	Example
string	`minio-ca-bundle`

`backup.storages.STORAGE-NAME.minio.caBundle.key`

The custom CA certificate for TLS communication with S3 storage. See Configure TLS verification with custom certificates for S3 storage for more information.

Value type	Example
string	`ca.crt`

`backup.storages.STORAGE-NAME.gcs.bucket`

The name of the storage bucket. See the GCS bucket naming guidelines for bucket name requirements.

Value type	Example
string	`""`

`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
string	`""`

`backup.storages.STORAGE-NAME.gcs.credentialsSecret`

The [Kubernetes secret](#) for backups. It contains the GCS credentials as either the service account and JSON keys or HMAC keys.

Value type	Example
string	`"my-cluster-name-backup-gcs"`

`backup.storages.STORAGE-NAME.gcs.chunkSize`

The size of data chunks in bytes to be uploaded to the GCS storage bucket in a single request. Larger data chunks will be split over multiple requests. Default data chunk size is 10MB.

Value type	Example
string	`10485760`

`backup.storages.STORAGE-NAME.gcs.retryer.backoffInitial`

The time to wait to make an initial retry, in seconds. Default value is 1 sec

Value type	Example
:material-code-int: int	`1`

`backup.storages.STORAGE-NAME.gcs.retryer.backoffMax`

The maximum amount of time between retries, in seconds. Default value is 30 sec.

Value type	Example
:material-code-int: int	`30`

`backup.storages.STORAGE-NAME.gcs.retryer.backoffMultiplier`

Defines the time to increase the wait time after each retry. For example, with the default value of 2 seconds, if the first wait time is 1 second, the next will be 2 seconds, then 4 seconds, and so on, until it reaches the maximum.

Value type	Example
:material-code-int: int	`2`

`backup.storages.STORAGE-NAME.azure.credentialsSecret`

The [Kubernetes secret](#) for backups. It should contain `AZURE_STORAGE_ACCOUNT_NAME` and `AZURE_STORAGE_ACCOUNT_KEY` |

Value type	Example
string	`my-cluster-azure-secret`

`backup.storages.STORAGE-NAME.azure.container`

Name of the container for backups.

Value type	Example
S string	`my-container`

`backup.storages.STORAGE-NAME.azure.prefix`

The path (sub-folder) to the backups inside the container.

Value type	Example
S string	`""`

`backup.storages.STORAGE-NAME.azure.endpointUrl`

The private endpoint URL to use instead of the public endpoint.

Value type	Example
S string	`https://accountName.blob.core.windows.net`

`backup.storages.STORAGE-NAME.filesystem.path`

The mount point for a remote filesystem configured to store backups.

Value type	Example
S string	`/mnt/nfs/`

`backup.volumeMounts.mountPath`

Mount path for the remote backup storage.

Value type	Example
S string	`/mnt/nfs/`

`backup.volumeMounts.name`

Name of the remote backup storage.

Value type	Example
S string	`backup-nfs`

`backup.pitr.enabled`

Enables or disables point-in-time-recovery functionality.

Value type	Example
boolean	`false`

`backup.pitr.oplogOnly`

If true, Percona Backup for MongoDB saves oplog chunks even without the base logical backup snapshot (oplog chunks without a base backup can't be used with logical backups to restore a backup by the Operator, but can still be useful for manual restore operations).

Value type	Example
boolean	false

`backup.pitr.oplogSpanMin`

Number of minutes between the uploads of oplogs.

Value type	Example
int	`10`

`backup.pitr.compressionType`

The point-in-time-recovery chunks compression format, can be gzip, snappy, lz4, pgzip, zstd, s2, or none.

Value type	Example
string	`gzip`

`backup.pitr.compressionLevel`

The point-in-time-recovery chunks compression level (higher values result in better but slower compression).

Value type	Example
int	`6`

`backup.configuration.backupOptions.priority`

The list of mongod nodes and their priority for making backups.

Value type	Example
subdoc	`"localhost:28019": 2.5` `"localhost:27018": 2.5`

`backup.configuration.backupOptions.timeouts.startingStatus`

The wait time in seconds Percona Backup for MongoDB should use to start physical backups on all shards. The 0 (zero) value resets the timeout to the default 33 seconds.

Value type	Example
int	`33`

`backup.configuration.backupOptions.oplogSpanMin`

The duration (in minutes) of oplog slices saved by Percona Backup for MongoDB with the logical backup snapshot.

Value type	Example

int	`10`

`backup.configuration.restoreOptions.batchSize`

The number of documents Percona Backup for MongoDB should buffer.

Value type	Example
int	`500`

`backup.configuration.restoreOptions.numInsertionWorkers`

The number of workers that Percona Backup for MongoDB should use to add the documents to buffer.

Value type	Example
int	`10`

`backup.configuration.restoreOptions.numDownloadWorkers`

The number of workers that Percona Backup for MongoDB should use to request data chunks from the storage during the restore.

Value type	Example
int	`4`

`backup.configuration.restoreOptions.maxDownloadBufferMb`

The maximum size of the in-memory buffer that Percona Backup for MongoDB should use use when downloading files from the S3 storage.

Value type	Example
int	`0`

`backup.configuration.restoreOptions.downloadChunkMb`

The size of the data chunk in MB, that Percona Backup for MongoDB should use when downloading from the S3 storage.

Value type	Example
int	`32`

`backup.configuration.restoreOptions.mongodLocation`

The custom path to mongod binaries which Percona Backup for MongoDB should use during restore.

Value type	Example
string	`/usr/bin/mongo`

`backup.configuration.restoreOptions.mongodLocationMap`

The list of custom paths to mongod binaries on every node, which Percona Backup for MongoDB should use during restore.

Value type	Example

subdoc	“node01:2017”: /usr/bin/mongo “node03:27017”: /usr/bin/mongo

`backup.tasks.name`

The name of the backup.

Value type	Example
string	

`backup.tasks.enabled`

Enables or disables this exact backup.

Value type	Example
boolean	`true`

`backup.tasks.schedule`

The scheduled time to make a backup, specified in the [crontab format]().

Value type	Example
string	`0 0 \* \* 6`

`backup.tasks.keep`

This option is deprecated and kept for backward compatibility. Use the `backup.tasks.retention` subsection instead.

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.

Value type	Example
int	`3`

`backup.tasks.retention.type`

Defines how to retain backups. The type of retention defaults to `count`.

Value type	Example
string	`count`

`backup.tasks.retention.count`

Defines the number of backups to store. Older backups are automatically deleted from the cluster.

Value type	Example
string	`count`

`backup.tasks.retention.deleteFromStorage`

Defines if the backups are deleted from the cloud storage too.

Value type	Example
boolean	`true`

`backup.tasks.storageName`

The name of the S3-compatible storage for backups, configured in the storages subsection.

Value type	Example
string	`st-us-west`

`backup.tasks.compressionType`

The backup compression format, can be gzip, snappy, lz4, pgzip, zstd, s2, or none.

Value type	Example
string	`gzip`

`backup.tasks.compressionLevel`

The backup compression level (higher values result in better but slower compression).

Value type	Example
int	`6`

`backup.tasks.type`

The backup type: (can be either `logical` (default) or `physical`; see the Operator backups official documentation for details.

Value type	Example
string	`physical`

Log Collector section

The `logcollector` section contains configuration options for Fluent Bit Log Collector.

`logcollector.enabled`

Enables or disables persistent logging with Fluent Bit.

Value type	Example
boolean	`true`

`logcollector.image`

Log Collector Docker image to use.

Value type	Example

string	`perconalab/fluentbit:main-logcollector`

`logcollector.configuration`

Additional configuration options (see [Fluent Bit official documentation]() for details).

Value type	Example
subdoc	

`logcollector.env.name`

The name of a custom environment variable for the log collector sidecar container (available since Operator 1.22.0). See [Define custom environment variables]().

Value type	Example
string	`MY_ENV`

`logcollector.env.value`

The value of a custom environment variable for the log collector sidecar container (available since Operator 1.22.0). See [Define custom environment variables]().

Value type	Example
string	`MY_VALUE`

`logcollector.envFrom.configMapRef.name`

The name of a ConfigMap from where environment variables will be loaded for the log collector sidecar container (available since Operator 1.22.0). See [Define custom environment variables]().

Value type	Example
string	`my-configmap`

`logcollector.envFrom.secretRef.name`

The name of a Secret from where environment variables will be loaded for the log collector sidecar container (available since Operator 1.22.0). See [Define custom environment variables]().

Value type	Example
string	`my-secret`
### `logcollector.logrotate.configuration`	

Overrides the default logrotate configuration used by the log collector sidecar container.

Value type	Example
subdoc	

`logcollector.logrotate.extraConfig.name`

References a ConfigMap or a Secret containing additional logrotate configuration. The key name must end with `.conf`. The `mongodb.conf` key name is reserved for the main configuration and is not allowed to use.

Value type	Example
S string	`logrotate-config`

`logcollector.logrotate.schedule`

Cron expression for the logrotate schedule (default: `0 0 0 * * *`).

Value type	Example
S string	`"0 0 0 * * *"`

`logcollector.resources.requests.memory`

The Kubernetes memory requests for a Log Collector sidecar container in a Percona Server for MongoDB Pod.

Value type	Example
S string	`100M`

`logcollector.resources.requests.cpu`

Kubernetes CPU requests for a Log collector sidecar container in a Percona Server for MongoDB Pod.

Value type	Example
S string	`200m`

11.2 Backup Resource options

A Backup resource is a Kubernetes object that tells the Operator how to create and manage your database backups. The `deploy/backup/backup.yaml` file is a template for creating backup resources when you make an on-demand backup. It defines the `PerconaServerMongoDBBackup` resource.

This document describes all available options that you can use to customize your backups.

`apiVersion`

Specifies the API version of the Custom Resource. `psmdb.percona.com` indicates the group, and `v1` is the version of the API.

`kind`

Defines the type of resource being created: `PerconaServerMongoDBBackup`.

`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. Note that it is ignored for incremental backups.
- `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.

`clusterName`

Specifies the name of the MongoDB cluster to back up.

Value type	Example
S string	`my-cluster-name`

`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.

Value type	Example
S string	`s3-us-west`

`type`

Specifies the backup type. Supported types are: `logical`, `physical`, `incremental-base`, `incremental`.

Value type	Example
S string	`physical`

`compressionType`

Specifies the compression algorithm for backups. Supported values are: `gzip`, `pgzip`, `zstd`, `snappy`. Read more about compression types in the [Configure backup compression] section of PBM documentation.

Value type	Example
string	`gzip`

`compressionLevel`

Specifies the compression level. Note that the higher value you specify, the more time and computing resources it will take to compress the data. The default value depends on the compression method used. Read more about compression levels in the [Configure backup compression] section of PBM documentation.

Value type	Example
string	`6`

`startingDeadlineSeconds`

The maximum time in seconds for a backup to start. The Operator compares the timestamp of the backup object against the current time. If the backup is not started within the set time, the Operator automatically marks it as “failed”.

If undefined, the Operator uses the value specified in the Custom Resource, or defaults to 120 seconds. Minimum allowed value is 1 second.

Value type	Example
int	`300`

11.3 Restore Resource options

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 `PerconaServerMongoDBRestore` resource.

This document describes all available options that you can use to customize a restore.

`apiVersion`

Specifies the API version of the Custom Resource. `psmdb.percona.com` indicates the group, and `v1` is the version of the API.

`kind`

Defines the type of resource being created: `PerconaServerMongoDBRestore`.

`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 MongoDB cluster to restore.

Value type	Example
S string	`my-cluster-name`

`storageName`

Specifies the name of the storage where a backup is located. The storage name must match the name in the `backup.storages` subsection of the `deploy/cr.yaml` file. Use this option together with the `backupSource` option. For more information on the flow and restore steps, see [Restore from a backup on a new cluster]

Value type	Example
S string	`s3-us-west`

`backupName`

Specifies the name of a backup to be used for a restore. This backup should be from the same cluster.

Value type	Example
S string	`backup1`

The `selective` subsection

Controls the selective restore, which enables you to restore a specific subset of namespaces - databases and collections.

`selective.withUsersAndRoles`

Allows restoring specified custom databases with users and roles that were created against them. Read more about [Selective restore with users and roles](#) in PBM documentation.

Value type	Example
boolean	`true`

`selective.namespaces`

Specifies the list of namespaces to restore. The namespace has the format `<db.collection>`

Value type	Example
array	`["db1.collection1", "db2.collection2"]`

`replsetRemapping`

Defines mapping between source and target replica set names during a restore. This should be a dictionary where each key is the replica set name from the source cluster, and the corresponding value is the desired replica set name in the target cluster. Read more about [restores to the cluster with different replica set names](#).

Value type	Example
array	`sourceRs0: targetRs0`

The `pitr` subsection

Controls how to make a point-in-time restore

`pitr.type`

Specifies the type of a point-in-time restore. Available options:

- `date` - restore to a specific date.
- `latest` - recover to the latest possible transaction

Value type	Example
string	`date`

`pitr.date`

Specifies the timestamp for the restore in the datetime format `YYYY-MM-DD hh:mm:ss`.

Use it together with the `type=date` option.

Value type	Example
string	`YYYY-MM-DD hh:mm:ss`

The `backupSource` subsection

Contains the configuration options to restore from a backup made in a different cluster, namespace, or Kubernetes environment.

`backupSource.type`

Specifies the backup type. Available options: physical, logical, incremental

Value type	Example
S string	`physical`

`backupSource.destination`

Specifies the path to the backup on the storage

Value type	Example
S string	`s3://bucket-name/backup-destination/`

`backupSource.s3.credentialsSecret`

Specifies the Secrets object name with the credentials to access the storage with a backup.

Value type	Example
S string	`my-cluster-name-backup-s3`

`backupSource.s3.serverSideEncryption.kmsKeyID`

Specifies your customer-managed key stored in the AWS Key Management Service (AWS KMS). This key is used to encrypt backup data uploaded to S3 buckets if you don't wish to use the default server-side encryption with Amazon S3 managed keys (SSE-S3)

Value type	Example
S string	`1234abcd-12ab-34cd-56ef-1234567890ab`

`backupSource.s3.serverSideEncryption.sseAlgorithm`

The encryption algorithm used to encrypt data

Value type	Example
S string	`AES256`

`backupSource.s3.serverSideEncryption.sseCustomerAlgorithm`

The encryption algorithm used for server-side encryption with customer-provided keys (SSE-C).

Value type	Example
S string	`AES256`

`backupSource.s3.serverSideEncryption.sseCustomerKey`

The customer-provided encryption key.

Value type	Example
S string	`Y3VzdG9tZXIta2V5`

`backupSource.s3.region`

The [AWS region](#) to use. Please note **this option is mandatory** for Amazon and all S3-compatible storages.

Value type	Example
string	`us-west-2`

`backupSource.s3.bucket`

The [Amazon S3 bucket](#) name for backups.

Value type	Example
string	

`backupSource.s3.endpointUrl`

The URL of the S3-compatible storage to be used (not needed for the original Amazon S3 cloud).

Value type	Example
string	`https://s3.us-west-2.amazonaws.com/`

`backupSource.s3.prefix`

The path to the data directory in the bucket. If undefined, backups are stored in the bucket's root directory.

Value type	Example
string	

`backupSource.azure.credentialsSecret`

Specifies the Secrets object name with the credentials to access the Azure Blob storage with a backup.

Value type	Example
string	

`backupSource.azure.prefix`

The path to the data directory in the bucket. If undefined, backups are stored in the bucket's root directory.

Value type	Example
string	

`backupSource.s3.container`

The name of the storage container. See the [naming conventions](#)

Value type	Example
string	

11.4 Custom resource statuses

Status fields show the current state of a Custom Resource (CR). 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 is ready).

How to view custom resource statuses

To check the status of your Percona custom resources, use the `kubectl get <resource-name>` or `kubectl describe <resource-name>` 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 psmdb -n <namespace>
kubectl get psmdb-backup -n <namespace>
kubectl get psmdb-restore -n <namespace>
```

Sample output for PerconaServerMongoDB

```
NAME        ENDPOINT                    STATUS    AGE
some-name   some-name-mongos.pitr-physical-backup-source-29217.svc.cluster.local:27017    ready    27m
```

View full details

See all status details as well as conditions and events:

```
kubectl get psmdb <cluster-name> -n <namespace> -o yaml
kubectl describe psmdb <cluster-name> -n <namespace>
```

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. View the status of the PBM configuration:

```
kubectl get psmdb <cluster-name> -n <namespace> \
  -o jsonpath='{range .items[0].status.conditions[?(@.type=="PBMReady")]}{.lastTransitionTime}{"\n"}{.reason}{"\n"}{.status}
{"\n"}{.type}{"\n"}{end}'
```

Sample output

```
2026-01-27T11:34:35Z
PBMConfigurationIsUpToDate
True
PBMReady
```

Example 2. Get the latest restorable backup time:

```
kubectl get psmdb-backup <backup-name> -n <namespace> \
  -o jsonpath='{.items[0].status.latestRestorableTime}'
```

Sample output

```
2026-01-27T12:22:17Z
```

PerconaServerMongoDB status

The main cluster state is recorded in the `status.state` section. For component-level states, see the `status.replsets` and `status.mongos` sections.

Common fields:

- `status.state` – overall cluster state
- `status.ready` / `status.size` – number of ready pods and the size of the database cluster
- `status.host` – connection endpoint
- `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.

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
- `reason` – short reason string
- `message` – human-readable details

`status.conditions[].type` values:

Value	Meaning
`initializing`	The cluster or a component is starting up.
`stopping`	The cluster or a component is stopping.
`paused`	The cluster or a component is paused.
`ready`	The cluster or a component is ready.
`error`	The Operator detected an error.
`sharding`	Sharding changes are in progress.
`PBMReady`	PBM agents and storage are ready.
`pendingSmartUpdate`	A smart update is pending but has not started.

`status.conditions[].status` values:

Value	Meaning
`True`	The condition is currently true.
`False`	The condition is currently false.

The Operator sets `reason` and `message` values as free-form strings. Common reasons include `ErrorReconcile`, `RSReady`, `RSStopping`, `RSPaused`, `MongosReady`, `MongosStopping`, `MongosPaused`, `PBMConfigurationIsUpToDate`, `PBMConfigurationIsChanged`.

PerconaServerMongoDBBackup status

Backup progress and results are in `status.state`. You also get destination and timing details that help you validate backups and Point-in-time recovery time ranges.

Common fields:

- `status.state` – backup job state
- `status.type` – backup type (`logical`, `physical`, `incremental`, `incremental-base`)
- `status.destination` – backup path or URL
- `status.size` – backup size
- `status.start` / `status.completed` – start and completion timestamps
- `status.latestRestorableTime` – latest point for point-in-time recovery from this backup
- `status.error` – error details when the backup fails

Backup state values

`status.state` values are:

Value	Meaning
`""`	Backup is created but not processed yet.
`waiting`	Backup is queued or waiting for resources.
`requested`	The Operator accepted the backup request.
`running`	Backup is in progress.
`ready`	Backup completed successfully.
`error`	Backup failed.

PerconaServerMongoDBRestore 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.pbmName` – PBM restore identifier
- `status.pitrTarget` – PITR target time (if set)
- `status.completed` – completion timestamp
- `status.error` – error details when the restore fails

Restore state values

`status.state` values are:

Value	Meaning
`""`	Restore is created but not processed yet.
`waiting`	Restore is queued or waiting for resources.
`requested`	The Operator accepted the restore request.
`rejected`	The Operator rejected the restore request.
`running`	Restore is in progress.
`ready`	Restore completed successfully.
`error`	Restore failed.

11.5 Percona certified images

This page lists Percona's certified Docker images that you can use with Percona Operator for MongoDB.

To find images for a specific Operator version, see Retrieve Percona certified images.

Images released with the Operator version 1.22.0:

Image	Digest
percona/percona-server-mongodb:8.0.19-7	779378a9f52cd9e617d0c356053b4b9c97209b0b759aa819300427836f6f2c66
percona/percona-server-mongodb:8.0.19-7 (ARM64)	531793543992d2e502d3187b778f8468462dd9cdb16a58fa157fe86626594ede
percona/percona-server-mongodb:7.0.30-16	32bc384e2af367e4261bdf1d2b09211bb210425f292494e5837e5a05f6805768
percona/percona-server-mongodb:7.0.30-16 (ARM64)	d333516c8b912d25b12759ab9c6d27d1ae4567e940f7922bd05c8dc06ae3228b
percona/percona-server-mongodb:6.0.27-21	2bd82853fd207ecadc8e5cd3ee598db7b9943f6976d9ae2140f5bdc997b0e8bb
percona/percona-server-mongodb:6.0.27-21 (ARM64)	a23e4fa1b956299fb7403eef70082b13b3ecf75e51bba12f674f552508a42d61
percona/fluentbit:4.0.1-2	d732be16b7f3d0eea8d9c0a50f72136db83c2aed74262300d97abd2cfaefbc26
percona/fluentbit:4.0.1-2 (ARM64)	805c102d3646b8499b74ea9864d4c29137f94e4603470c23da1c032276193a36
percona/pmm-client:3.6.0	174fa4675d3ea4d95fd7b45d11f2bcc98b98b703662e6b2614dfe886a7187b23
percona/pmm-client:3.6.0 (ARM64)	435a9af2083adb68ddab6a97e6d02bd6d31c54562e919ebc09618e886d58d1ae
percona/pmm-client:2.44.1-1	52a8fb5e8f912eef1ff8a117ea323c401e278908ce29928dafc23fac1db4f1e3
percona/pmm-client:2.44.1-1 (ARM64)	390bfd12f981e8b3890550c4927a3ece071377065e001894458047602c744e3b
percona/percona-backup-mongodb:2.12.0	6d56b787c2dc532c9c19cc607a2bc1855a722cdaa52024ea38d4de4aae777f8a
percona/percona-backup-mongodb:2.12.0 (ARM64)	e2177fa6fef0e47ee8392ca7044a6781150ccf7297383239f813558323aa0a8e
percona/percona-server-mongodb-operator:1.22.0	d4384b3853dc5026cc33de1dfd68ac11aaf5d54f4e84b115e05d7d473f4c6188
percona/percona-server-mongodb-operator:1.22.0 (ARM64)	be97aad4c0642bc4e2e5c7c31b4d123d9f6135070682b8fbd6024c7620315e72

Image tag format

Image tags have the format:

`[component_name]-[component_version]`

where:

- `component_name` is the name of the component. For example, `percona-server-mongodb`
- `component_version` is the version of the component. For example, `8.0.4-1`.

Note, that PMM Client images have their own tags. They contain the version of PMM.

Find images for previous versions

11.6 Versions compatibility

Versions of the cluster components and platforms tested with different Operator releases are shown below. Other version combinations may also work but have not been tested.

Cluster components

Operator	MongoDB	Percona Backup for MongoDB
1.22.0	6.0.27-21, 7.0.30-16, 8.0.19-7	2.12.0
1.21.2	6.0 - 8.0	2.11.0
1.21.1	6.0 - 8.0	2.11.0
1.21.0	6.0 - 8.0	2.11.0
1.20.0	6.0 - 8.0	2.9.1
1.19.1	6.0 - 8.0	2.8.0
1.19.0	6.0 - 8.0	2.8.0
1.18.0	5.0 - 7.0	2.7.0
1.17.0	5.0 - 7.0	2.5.0
1.16.2	5.0 - 7.0	2.4.1
1.16.1	5.0 - 7.0	2.4.1
1.16.0	5.0 - 7.0	2.4.1
1.15.0	4.4 - 6.0	2.3.0
1.14.0	4.4 - 6.0	2.0.4, 2.0.5
1.13.0	4.2 - 5.0	1.8.1
1.12.0	4.2 - 5.0	1.7.0
1.11.0	4.0, 4.2, 4.4, 5.0	1.6.1
1.10.0	4.0, 4.2, 4.4, 5.0	1.6.0
1.9.0	4.0, 4.2, 4.4	1.5.0
1.8.0	3.6, 4.0, 4.2, 4.4	1.4.1
1.7.0	3.6, 4.0, 4.2, 4.4	1.4.1
1.6.0	3.6, 4.0, 4.2	1.3.4
1.5.0	3.6, 4.0, 4.2	1.3.1
1.4.0	3.6, 4.0, 4.2	1.1.0
1.3.0	3.6, 4.0	0.4.0
1.2.0	3.6, 4.0	0.4.0
1.1.0	3.6, 4.0	0.4.0

Platforms

| Operator | GKE | EKS | Openshift | AKS | Minikube | |:--------|:------------|:------------|:------------|:------------|:---------------------------------|\ | 1.22.0 | 1.32 - 1.33 | 1.32 - 1.35 | 4.17 - 4.21 | 1.32 - 1.34 | 1.38.0 | | 1.21.2 | 1.31 - 1.33 | 1.31 - 1.34 | 4.16 - 4.19 | 1.31 - 1.33 | 1.37.0 | | 1.21.1 | 1.31 - 1.33 | 1.31 - 1.34 | 4.16 - 4.19 | 1.31 - 1.33 | 1.37.0 | | 1.21.0 | 1.31 - 1.33 | 1.31 - 1.34 | 4.16 - 4.19 | 1.31 - 1.33 | 1.37.0 | | 1.20.0 | 1.30 - 1.32 | 1.30 - 1.32 | 4.14 - 4.18 | 1.30 - 1.32 | 1.35.0 | | 1.19.1 | 1.28 - 1.30 | 1.29 - 1.31 | 4.14.44 - 4.17.11 | 1.28 - 1.31 | 1.34.0 | | 1.19.0 | 1.28 - 1.30 | 1.29 - 1.31 | 4.14.44 - 4.17.11 | 1.28 - 1.31 | 1.34.0 | | 1.18.0 | 1.28 - 1.30 | 1.28 - 1.31 | 4.13.52 - 4.17.3 | 1.28 - 1.31 | 1.34.0 | | 1.17.0 | 1.27 - 1.30 | 1.28 - 1.30 | 4.13.48 - 4.16.9 | 1.28 - 1.30 | 1.33.1 | | 1.16.2 | 1.26 - 1.29 | 1.26 - 1.29 | 4.12.56 - 4.15.11 | 1.27 - 1.29 | 1.33 | | 1.16.1 | 1.26 - 1.29 | 1.26 - 1.29 | 4.12.56 - 4.15.11 | 1.27 - 1.29 | 1.33 | | 1.16.0 | 1.26 - 1.29 | 1.26 - 1.29 | 4.12.56 - 4.15.11 | 1.27 - 1.29 | 1.33 | | 1.15.0 | 1.24 - 1.28 | 1.24 - 1.28 | 4.11 - 4.13 | 1.25 - 1.28 | 1.31.2 | | 1.14.0 | 1.22 - 1.25 | 1.22 - 1.24 | 4.10 - 4.12 | 1.23 - 1.25 | 1.29 | | 1.13.0 | 1.21 - 1.23 | 1.21 - 1.23 | 4.10 - 4.11 | 1.22 - 1.24 | 1.26 | | 1.12.0 | 1.19 - 1.22 | 1.19 - 1.22 | 4.7 - 4.10 | - | 1.23 | | 1.11.0 | 1.19 - 1.22 | 1.18 - 1.22 | 4.7 - 4.9 | - | 1.22 | | 1.10.0 | 1.17 - 1.21 | 1.16 - 1.21 | 4.6 - 4.8 | - | 1.22 | | 1.9.0 | 1.17 - 1.21 | 1.16-1.20 | 4.7 | - | 1.20 | | 1.8.0 | 1.16 - 1.20 | 1.19 | 3.11, 4.7 | - | 1.19 | | 1.7.0 | 1.15 - 1.17 | 1.15 | 3.11, 4.5 | - | 1.10 | | 1.6.0 | 1.15 - 1.17 | 1.15 | 3.11, 4.5 | - | 1.10 | | 1.5.0 | 1.15 - 1.17 | 1.15 | 3.11, 4.5 | - | 1.18 | | 1.4.0 | 1.13, 1.15 | 1.15 | 3.11, 4.2 | - | 1.16 | | 1.3.0 | 1.11, 1.14 | - | 3.11, 4.1 | - | 1.12 | | 1.2.0 | - | - | 3.11, 4.0 | - | - | | 1.1.0 | - | - | 3.11, 4.0 | - | - |

More detailed information about the cluster components for the current version of the Operator can be found in the system requirements and in the list of certified images. For previous releases of the Operator, you can check the same pages in the documentation archive.

11.7 Percona Operator for MongoDB API Documentation

Percona Operator for MongoDB provides an aggregation-layer extension for the Kubernetes API. Please refer to the official Kubernetes API documentation on the API access and usage details. The following subsections describe the Percona XtraDB Cluster API provided by the Operator.

Prerequisites

1. Create the namespace name you will use, if not exist:

   ```
   kubectl create namespace my-namespace-name
   ```

   Trying to create an already-existing namespace will show you a self-explanatory error message. Also, you can use the `default` namespace.

   > **Note**
   >
   > In this document `default` namespace is used in all examples. Substitute `default` with your namespace name if you use a different one.

2. Prepare:

   ```
   # set correct API address
   KUBE_CLUSTER=$(kubectl config view --minify -o jsonpath='{.clusters[0].name}')
   API_SERVER=$(kubectl config view -o jsonpath="{.clusters[?(@.name==\"$KUBE_CLUSTER\")].cluster.server}" | sed -e
   's#https://##')

   # create service account and get token
   kubectl apply --server-side -f deploy/crd.yaml -f deploy/rbac.yaml -n default
   KUBE_TOKEN=$(kubectl get secret $(kubectl get serviceaccount percona-server-mongodb-operator -o
   jsonpath='{.secrets[0].name}' -n default) -o jsonpath='{.data.token}' -n default | base64 --decode )
   ```

Create new Percona Server for MongoDB cluster

Description:

```
The command to create a new Percona Server for MongoDB cluster
```

Kubectl Command:

```
kubectl apply -f percona-server-mongodb-operator/deploy/cr.yaml
```

URL:

```
https://$API_SERVER/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbs
```

Authentication:

```
Authorization: Bearer $KUBE_TOKEN
```

cURL Request:

```
curl -k -v -XPOST "https://$API_SERVER/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbs" \
            -H "Content-Type: application/json" \
            -H "Accept: application/json" \
            -H "Authorization: Bearer $KUBE_TOKEN" \
            -d "@cluster.json"
```

Request Body (cluster.json):

> **Example**
>
> ```
> {
>    "apiVersion": "psmdb.percona.com/v1-5-0",
> ```

```
    "kind": "PerconaServerMongoDB",
    "metadata": {
       "name": "my-cluster-name"
    },
    "spec": {
       "image": "percona/percona-server-mongodb:4.2.8-8",
       "imagePullPolicy": "Always",
       "allowUnsafeConfigurations": false,
       "updateStrategy": "SmartUpdate",
       "secrets": {
          "users": "my-cluster-name-secrets"
       },
       "pmm": {
          "enabled": false,
          "image": "percona/percona-server-mongodb-operator:1.5.0-pmm",
          "serverHost": "monitoring-service"
       },
       "replsets": [
          {
             "name": "rs0",
             "size": 3,
             "affinity": {
                "antiAffinityTopologyKey": "none"
             },
             "podDisruptionBudget": {
                "maxUnavailable": 1
             },
             "expose": {
                "enabled": false,
                "exposeType": "LoadBalancer"
             },
             "arbiter": {
                "enabled": false,
                "size": 1,
                "affinity": {
                   "antiAffinityTopologyKey": "none"
                }
             },
             "resources": {
                "limits": null
             },
             "volumeSpec": {
                "persistentVolumeClaim": {
                   "storageClassName": "standard",
                   "accessModes": [
                      "ReadWriteOnce"
                   ],
                   "resources": {
                      "requests": {
                         "storage": "3Gi"
                      }
                   }
                }
             }
          }
       ],
       "mongod": {
          "net": {
             "port": 27017,
             "hostPort": 0
          },
          "security": {
             "redactClientLogData": false,
             "enableEncryption": true,
             "encryptionKeySecret": "my-cluster-name-mongodb-encryption-key",
             "encryptionCipherMode": "AES256-CBC"
          },
          "setParameter": {
             "ttlMonitorSleepSecs": 60,
             "wiredTigerConcurrentReadTransactions": 128,
             "wiredTigerConcurrentWriteTransactions": 128
          },
          "storage": {
             "engine": "wiredTiger",
             "inMemory": {
                "engineConfig": {
                   "inMemorySizeRatio": 0.9
                }
             },
             "mmapv1": {
                "nsSize": 16,
                "smallfiles": false
             },
             "wiredTiger": {
                "engineConfig": {
                   "cacheSizeRatio": 0.5,
                   "directoryForIndexes": false,
                   "journalCompressor": "snappy"
                },
                "collectionConfig": {
                   "blockCompressor": "snappy"
```

```
            },
            "indexConfig": {
               "prefixCompression": true
            }
         }
      },
      "operationProfiling": {
         "mode": "slowOp",
         "slowOpThresholdMs": 100,
         "rateLimit": 100
      }
   },
   "backup": {
      "enabled": true,
      "restartOnFailure": true,
      "image": "percona/percona-server-mongodb-operator:1.5.0-backup",
      "serviceAccountName": "percona-server-mongodb-operator",
      "storages": null,
      "tasks": null
   }
  }
}
```


Inputs:

Metadata:

1. Name (String, min-length: 1) : `contains name of cluster`

Spec:

1. secrets[users] (String, min-length: 1) : `contains name of secret for the users`
2. allowUnsafeConfigurations (Boolean, Default: false) : `allow unsafe configurations to run`
3. image (String, min-length: 1) : `name of the Percona Server for MongoDB cluster image`

replsets:

1. name (String, min-length: 1) : `name of monogo replicaset`
2. size (Integer, min-value: 1) : `contains size of MongoDB replicaset`
3. expose[exposeType] (Integer, min-value: 1) : `type of service to expose replicaset`
4. arbiter (Object) : `configuration for mongo arbiter`

mongod:

1. net:
    a. port (Integer, min-value: 0) : `contains mongod container port`
    b. hostPort (Integer, min-value: 0) : `host port to expose mongod on`
2. security:
    a. enableEncryption (Boolean, Default: true) : `enable encrypting mongod storage`
    b. encryptionKeySecret (String, min-length: 1) : `name of encryption key secret`
    c. encryptionCipherMode (String, min-length: 1) : `type of encryption cipher to use`
3. setParameter (Object): `configure mongod engineer parameters`
4. storage:
    a. engine (String, min-length: 1, default "wiredTiger"): `name of mongod storage engine`
    b. inMemory (Object) : `wiredTiger engine configuration`
    c. wiredTiger (Object) : `wiredTiger engine configuration`

pmm:

1. serverHost (String, min-length: 1) : `service name for monitoring`
2. image (String, min-length: 1) : `name of pmm image`

backup:

1. image (String, min-length: 1) : `name of MngoDB backup docker image`
2. serviceAccountName (String, min-length: 1) `name of service account to use for backup`
3. storages (Object) : `storage configuration object for backup`

Response:

Example


```
{
   "apiVersion":"psmdb.percona.com/v1-5-0",
   "kind":"PerconaServerMongoDB",
   "metadata":{
      "annotations":{
         "kubectl.kubernetes.io/last-applied-configuration":"{\"apiVersion\":\"psmdb.percona.com/v1-5-
0\",\"kind\":\"PerconaServerMongoDB\",\"metadata\":{\"annotations\":{},\"name\":\"my-cluster-name\",\"namespace\":\"default\"},\"spec\":
{\"allowUnsafeConfigurations\":false,\"backup\":{\"enabled\":true,\"image\":\"percona/percona-server-mongodb-operator:1.5.0-
backup\",\"restartOnFailure\":true,\"serviceAccountName\":\"percona-server-mongodb-
operator\",\"storages\":null,\"tasks\":null},\"image\":\"percona/percona-server-mongodb:4.2.8-8\",\"imagePullPolicy\":\"Always\",\"mongod\":
```

```
{\"net\":{\"hostPort\":0,\"port\":27017},\"operationProfiling\":{\"mode\":\"slowOp\",\"rateLimit\":100,\"slowOpThresholdMs\":100},\"security\":
{\"enableEncryption\":true,\"encryptionCipherMode\":\"AES256-CBC\",\"encryptionKeySecret\":\"my-cluster-name-mongodb-encryption-
key\",\"redactClientLogData\":false},\"setParameter\":
{\"ttlMonitorSleepSecs\":60,\"wiredTigerConcurrentReadTransactions\":128,\"wiredTigerConcurrentWriteTransactions\":128},\"storage\":
{\"engine\":\"wiredTiger\",\"inMemory\":{\"engineConfig\":{\"inMemorySizeRatio\":0.9}},\"mmapv1\":
{\"nsSize\":16,\"smallfiles\":false},\"wiredTiger\":{\"collectionConfig\":{\"blockCompressor\":\"snappy\"},\"engineConfig\":
{\"cacheSizeRatio\":0.5,\"directoryForIndexes\":false,\"journalCompressor\":\"snappy\"},\"indexConfig\":{\"prefixCompression\":true}}}},\"pmm\":
{\"enabled\":false,\"image\":\"percona/percona-server-mongodb-operator:1.5.0-pmm\",\"serverHost\":\"monitoring-service\"},\"replsets\":
[{\"affinity\":{\"antiAffinityTopologyKey\":\"none\"},\"arbiter\":{\"affinity\":
{\"antiAffinityTopologyKey\":\"none\"},\"enabled\":false,\"size\":1},\"expose\":
{\"enabled\":false,\"exposeType\":\"LoadBalancer\"},\"name\":\"rs0\",\"podDisruptionBudget\":{\"maxUnavailable\":1},\"resources\":
{\"limits\":null},\"size\":3,\"volumeSpec\":{\"persistentVolumeClaim\":{\"accessModes\":[\"ReadWriteOnce\"],\"resources\":{\"requests\":
{\"storage\":\"3Gi\"}},\"storageClassName\":\"standard\"}}}],\"secrets\":{\"users\":\"my-cluster-name-
secrets\"},\"updateStrategy\":\"SmartUpdate\"}}\n"
      },
      "creationTimestamp":"2020-07-24T14:27:58Z",
      "generation":1,
      "managedFields":[
         {
            "apiVersion":"psmdb.percona.com/v1-5-0",
            "fieldsType":"FieldsV1",
            "fieldsV1":{
               "f:metadata":{
                  "f:annotations":{
                     ".":{

                     },
                     "f:kubectl.kubernetes.io/last-applied-configuration":{

                     }
                  }
               },
               "f:spec":{
                  ".":{

                  },
                  "f:allowUnsafeConfigurations":{

                  },
                  "f:backup":{
                     ".":{

                     },
                     "f:enabled":{

                     },
                     "f:image":{

                     },
                     "f:restartOnFailure":{

                     },
                     "f:serviceAccountName":{

                     },
                     "f:storages":{

                     },
                     "f:tasks":{

                     }
                  },
                  "f:image":{

                  },
                  "f:imagePullPolicy":{

                  },
                  "f:mongod":{
                     ".":{

                     },
                     "f:net":{
                        ".":{

                        },
                        "f:hostPort":{

                        },
                        "f:port":{

                        }
                     },
                     "f:operationProfiling":{
                        ".":{

                        },
                        "f:mode":{

                        },
                        "f:rateLimit":{
```

```
            },
            "f:slowOpThresholdMs":{

            }
         },
         "f:security":{
            ".":{

            },
            "f:enableEncryption":{

            },
            "f:encryptionCipherMode":{

            },
            "f:encryptionKeySecret":{

            },
            "f:redactClientLogData":{

            }
         },
         "f:setParameter":{
            ".":{

            },
            "f:ttlMonitorSleepSecs":{

            },
            "f:wiredTigerConcurrentReadTransactions":{

            },
            "f:wiredTigerConcurrentWriteTransactions":{

            }
         },
         "f:storage":{
            ".":{

            },
            "f:engine":{

            },
            "f:inMemory":{
               ".":{

               },
               "f:engineConfig":{
                  ".":{

                  },
                  "f:inMemorySizeRatio":{

                  }
               }
            },
            "f:mmapv1":{
               ".":{

               },
               "f:nsSize":{

               },
               "f:smallfiles":{

               }
            },
            "f:wiredTiger":{
               ".":{

               },
               "f:collectionConfig":{
                  ".":{

                  },
                  "f:blockCompressor":{

                  }
               },
               "f:engineConfig":{
                  ".":{

                  },
                  "f:cacheSizeRatio":{

                  },
                  "f:directoryForIndexes":{

                  },
                  "f:journalCompressor":{
```

```
                              }
                           },
                           "f:indexConfig":{
                              ".":{

                              },
                              "f:prefixCompression":{

                              }
                           }
                        }
                     }
                  },
                  "f:pmm":{
                     ".":{

                     },
                     "f:enabled":{

                     },
                     "f:image":{

                     },
                     "f:serverHost":{

                     }
                  },
                  "f:replsets":{

                  },
                  "f:secrets":{
                     ".":{

                     },
                     "f:users":{

                     }
                  },
                  "f:updateStrategy":{

                  }
               }
            },
            "manager":"kubectl",
            "operation":"Update",
            "time":"2020-07-24T14:27:58Z"
         }
      ],
      "name":"my-cluster-name",
      "namespace":"default",
      "resourceVersion":"1268922",
      "selfLink":"/apis/psmdb.percona.com/v1-5-0/namespaces/default/perconaservermongodbs/my-cluster-name",
      "uid":"5207e71a-c83f-4707-b892-63aa93fb615c"
   },
   "spec":{
      "allowUnsafeConfigurations":false,
      "backup":{
         "enabled":true,
         "image":"percona/percona-server-mongodb-operator:1.5.0-backup",
         "restartOnFailure":true,
         "serviceAccountName":"percona-server-mongodb-operator",
         "storages":null,
         "tasks":null
      },
      "image":"percona/percona-server-mongodb:4.2.8-8",
      "imagePullPolicy":"Always",
      "mongod":{
         "net":{
            "hostPort":0,
            "port":27017
         },
         "operationProfiling":{
            "mode":"slowOp",
            "rateLimit":100,
            "slowOpThresholdMs":100
         },
         "security":{
            "enableEncryption":true,
            "encryptionCipherMode":"AES256-CBC",
            "encryptionKeySecret":"my-cluster-name-mongodb-encryption-key",
            "redactClientLogData":false
         },
         "setParameter":{
            "ttlMonitorSleepSecs":60,
            "wiredTigerConcurrentReadTransactions":128,
            "wiredTigerConcurrentWriteTransactions":128
         },
         "storage":{
            "engine":"wiredTiger",
            "inMemory":{
```

```
            "engineConfig":{
               "inMemorySizeRatio":0.9
            }
         },
         "mmapv1":{
            "nsSize":16,
            "smallfiles":false
         },
         "wiredTiger":{
            "collectionConfig":{
               "blockCompressor":"snappy"
            },
            "engineConfig":{
               "cacheSizeRatio":0.5,
               "directoryForIndexes":false,
               "journalCompressor":"snappy"
            },
            "indexConfig":{
               "prefixCompression":true
            }
         }
      }
   },
   "pmm":{
      "enabled":false,
      "image":"percona/percona-server-mongodb-operator:1.5.0-pmm",
      "serverHost":"monitoring-service"
   },
   "replsets":[
      {
         "affinity":{
            "antiAffinityTopologyKey":"none"
         },
         "arbiter":{
            "affinity":{
               "antiAffinityTopologyKey":"none"
            },
            "enabled":false,
            "size":1
         },
         "expose":{
            "enabled":false,
            "exposeType":"LoadBalancer"
         },
         "name":"rs0",
         "podDisruptionBudget":{
            "maxUnavailable":1
         },
         "resources":{
            "limits":null
         },
         "size":3,
         "volumeSpec":{
            "persistentVolumeClaim":{
               "accessModes":[
                  "ReadWriteOnce"
               ],
               "resources":{
                  "requests":{
                     "storage":"3Gi"
                  }
               },
               "storageClassName":"standard"
            }
         }
      }
   ],
   "secrets":{
      "users":"my-cluster-name-secrets"
   },
   "updateStrategy":"SmartUpdate"
 }
}
```

List Percona Server for MongoDB clusters

Description:

```
Lists all Percona Server for MongoDB clusters that exist in your kubernetes cluster
```

Kubectl Command:

```
kubectl get psmdb
```

URL:

```
https://$API_SERVER/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbs?limit=500
```

Authentication:

```
Authorization: Bearer $KUBE_TOKEN
```

cURL Request:

```
curl -k -v -XGET "https://$API_SERVER/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbs?limit=500" \
            -H "Accept:
application/json;as=Table;v=v1;g=meta.k8s.io,application/json;as=Table;v=v1beta1;g=meta.k8s.io,application/json" \
            -H "Authorization: Bearer $KUBE_TOKEN"
```

Request Body:

```
None
```

Response:

Example


```
{
   "kind":"Table",
   "apiVersion":"meta.k8s.io/v1",
   "metadata":{
      "selfLink":"/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbs",
      "resourceVersion":"1273793"
   },
   "columnDefinitions":[
      {
         "name":"Name",
         "type":"string",
         "format":"name",
         "description":"Name must be unique within a namespace. Is required when creating resources, although some resources may allow a client
to request the generation of an appropriate name automatically. Name is primarily intended for creation idempotence and configuration
definition. Cannot be updated. More info: http://kubernetes.io/docs/user-guide/identifiers#names",
         "priority":0
      },
      {
         "name":"Status",
         "type":"string",
         "format":"",
         "description":"Custom resource definition column (in JSONPath format): .status.state",
         "priority":0
      },
      {
         "name":"Age",
         "type":"date",
         "format":"",
         "description":"Custom resource definition column (in JSONPath format): .metadata.creationTimestamp",
         "priority":0
      }
   ],
   "rows":[
      {
         "cells":[
            "my-cluster-name",
            "ready",
            "37m"
         ],
         "object":{
            "kind":"PartialObjectMetadata",
            "apiVersion":"meta.k8s.io/v1",
            "metadata":{
               "name":"my-cluster-name",
               "namespace":"default",
               "selfLink":"/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbs/my-cluster-name",
               "uid":"5207e71a-c83f-4707-b892-63aa93fb615c",
               "resourceVersion":"1273788",
               "generation":1,
               "creationTimestamp":"2020-07-24T14:27:58Z",
               "annotations":{
                  "kubectl.kubernetes.io/last-applied-configuration":"{\"apiVersion\":\"psmdb.percona.com/v1-5-
0\",\"kind\":\"PerconaServerMongoDB\",\"metadata\":{\"annotations\":{},\"name\":\"my-cluster-name\",\"namespace\":\"default\"},\"spec\":
{\"allowUnsafeConfigurations\":false,\"backup\":{\"enabled\":true,\"image\":\"percona/percona-server-mongodb-operator:1.5.0-
backup\",\"restartOnFailure\":true,\"serviceAccountName\":\"percona-server-mongodb-
operator\",\"storages\":null,\"tasks\":null},\"image\":\"percona/percona-server-mongodb:4.2.8-8\",\"imagePullPolicy\":\"Always\",\"mongod\":
{\"net\":{\"hostPort\":0,\"port\":27017},\"operationProfiling\":{\"mode\":\"slowOp\",\"rateLimit\":100,\"slowOpThresholdMs\":100},\"security\":
{\"enableEncryption\":true,\"encryptionCipherMode\":\"AES256-CBC\",\"encryptionKeySecret\":\"my-cluster-name-mongodb-encryption-
```

```
key\",\"redactClientLogData\":false},\"setParameter\":
{\"ttlMonitorSleepSecs\":60,\"wiredTigerConcurrentReadTransactions\":128,\"wiredTigerConcurrentWriteTransactions\":128},\"storage\":
{\"engine\":\"wiredTiger\",\"inMemory\":{\"engineConfig\":{\"inMemorySizeRatio\":0.9}},\"mmapv1\":
{\"nsSize\":16,\"smallfiles\":false},\"wiredTiger\":{\"collectionConfig\":{\"blockCompressor\":\"snappy\"},\"engineConfig\":
{\"cacheSizeRatio\":0.5,\"directoryForIndexes\":false,\"journalCompressor\":\"snappy\"},\"indexConfig\":{\"prefixCompression\":true}}}},\"pmm\":
{\"enabled\":false,\"image\":\"percona/percona-server-mongodb-operator:1.5.0-pmm\",\"serverHost\":\"monitoring-service\"},\"replsets\":
[{\"affinity\":{\"antiAffinityTopologyKey\":\"none\"},\"arbiter\":{\"affinity\":
{\"antiAffinityTopologyKey\":\"none\"},\"enabled\":false,\"size\":1},\"expose\":
{\"enabled\":false,\"exposeType\":\"LoadBalancer\"},\"name\":\"rs0\",\"podDisruptionBudget\":{\"maxUnavailable\":1},\"resources\":
{\"limits\":null},\"size\":3,\"volumeSpec\":{\"persistentVolumeClaim\":{\"accessModes\":[\"ReadWriteOnce\"],\"resources\":{\"requests\":
{\"storage\":\"3Gi\"}},\"storageClassName\":\"standard\"}}}],\"secrets\":{\"users\":\"my-cluster-name-
secrets\"},\"updateStrategy\":\"SmartUpdate\"}}\n"
               },
               "managedFields":[
                  {
                     "manager":"kubectl",
                     "operation":"Update",
                     "apiVersion":"psmdb.percona.com/v1-5-0",
                     "time":"2020-07-24T14:27:58Z",
                     "fieldsType":"FieldsV1",
                     "fieldsV1":{
                        "f:metadata":{
                           "f:annotations":{
                              ".":{

                              },
                              "f:kubectl.kubernetes.io/last-applied-configuration":{

                              }
                           }
                        },
                        "f:spec":{
                           ".":{

                           },
                           "f:allowUnsafeConfigurations":{

                           },
                           "f:backup":{
                              ".":{

                              },
                              "f:enabled":{

                              },
                              "f:image":{

                              },
                              "f:serviceAccountName":{

                              }
                           },
                           "f:image":{

                           },
                           "f:imagePullPolicy":{

                           },
                           "f:mongod":{
                              ".":{

                              },
                              "f:net":{
                                 ".":{

                                 },
                                 "f:port":{

                                 }
                              },
                              "f:operationProfiling":{
                                 ".":{

                                 },
                                 "f:mode":{

                                 },
                                 "f:rateLimit":{

                                 },
                                 "f:slowOpThresholdMs":{

                                 }
                              },
                              "f:security":{
                                 ".":{

                                 },
                                 "f:enableEncryption":{

                                 },
```

```
                              "f:encryptionCipherMode":{

                              },
                              "f:encryptionKeySecret":{

                              }
                           },
                           "f:setParameter":{
                              ".":{

                              },
                              "f:ttlMonitorSleepSecs":{

                              },
                              "f:wiredTigerConcurrentReadTransactions":{

                              },
                              "f:wiredTigerConcurrentWriteTransactions":{

                              }
                           },
                           "f:storage":{
                              ".":{

                              },
                              "f:engine":{

                              },
                              "f:inMemory":{
                                 ".":{

                                 },
                                 "f:engineConfig":{
                                    ".":{

                                    },
                                    "f:inMemorySizeRatio":{

                                    }
                                 }
                              },
                              "f:mmapv1":{
                                 ".":{

                                 },
                                 "f:nsSize":{

                                 }
                              },
                              "f:wiredTiger":{
                                 ".":{

                                 },
                                 "f:collectionConfig":{
                                    ".":{

                                    },
                                    "f:blockCompressor":{

                                    }
                                 },
                                 "f:engineConfig":{
                                    ".":{

                                    },
                                    "f:cacheSizeRatio":{

                                    },
                                    "f:journalCompressor":{

                                    }
                                 },
                                 "f:indexConfig":{
                                    ".":{

                                    },
                                    "f:prefixCompression":{

                                    }
                                 }
                              }
                           }
                        },
                        "f:pmm":{
                           ".":{

                           },
                           "f:image":{

                           },
                           "f:serverHost":{
```

```
            }
         },
         "f:secrets":{
            ".":{

            },
            "f:users":{

            }
         },
         "f:updateStrategy":{

         }
      }
   }
},
{
   "manager":"percona-server-mongodb-operator",
   "operation":"Update",
   "apiVersion":"psmdb.percona.com/v1",
   "time":"2020-07-24T15:04:55Z",
   "fieldsType":"FieldsV1",
   "fieldsV1":{
      "f:spec":{
         "f:backup":{
            "f:containerSecurityContext":{
               ".":{

               },
               "f:runAsNonRoot":{

               },
               "f:runAsUser":{

               }
            },
            "f:podSecurityContext":{
               ".":{

               },
               "f:fsGroup":{

               }
            }
         },
         "f:clusterServiceDNSSuffix":{

         },
         "f:replsets":{

         },
         "f:runUid":{

         },
         "f:secrets":{
            "f:ssl":{

            },
            "f:sslInternal":{

            }
         }
      },
      "f:status":{
         ".":{

         },
         "f:conditions":{

         },
         "f:observedGeneration":{

         },
         "f:replsets":{
            ".":{

            },
            "f:rs0":{
               ".":{

               },
               "f:ready":{

               },
               "f:size":{

               },
               "f:status":{

               }
```

```
                        }
                     },
                     "f:state":{

                     }
                  }
               }
            }
         ]
      }
   }
}
   ]
}
```


Get status of Percona Server for MongoDB cluster

Description:

```
Gets all information about specified Percona Server for MongoDB cluster
```

Kubectl Command:

```
kubectl get psmdb/my-cluster-name -o json
```

URL:

```
https://$API_SERVER/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbs/my-cluster-name
```

Authentication:

```
Authorization: Bearer $KUBE_TOKEN
```

cURL Request:

```
curl -k -v -XGET "https://$API_SERVER/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbs/my-cluster-name" \
            -H "Accept: application/json" \
            -H "Authorization: Bearer $KUBE_TOKEN"
```

Request Body:

```
None
```

Response:

Example


```
{
   "apiVersion":"psmdb.percona.com/v1",
   "kind":"PerconaServerMongoDB",
   "metadata":{
      "annotations":{
         "kubectl.kubernetes.io/last-applied-configuration":"{\"apiVersion\":\"psmdb.percona.com/v1-5-
0\",\"kind\":\"PerconaServerMongoDB\",\"metadata\":{\"annotations\":{},\"name\":\"my-cluster-name\",\"namespace\":\"default\"},\"spec\":
{\"allowUnsafeConfigurations\":false,\"backup\":{\"enabled\":true,\"image\":\"percona/percona-server-mongodb-operator:1.5.0-
backup\",\"restartOnFailure\":true,\"serviceAccountName\":\"percona-server-mongodb-
operator\",\"storages\":null,\"tasks\":null},\"image\":\"percona/percona-server-mongodb:4.2.8-8\",\"imagePullPolicy\":\"Always\",\"mongod\":
{\"net\":{\"hostPort\":0,\"port\":27017},\"operationProfiling\":{\"mode\":\"slowOp\",\"rateLimit\":100,\"slowOpThresholdMs\":100},\"security\":
{\"enableEncryption\":true,\"encryptionCipherMode\":\"AES256-CBC\",\"encryptionKeySecret\":\"my-cluster-name-mongodb-encryption-
key\",\"redactClientLogData\":false},\"setParameter\":
{\"ttlMonitorSleepSecs\":60,\"wiredTigerConcurrentReadTransactions\":128,\"wiredTigerConcurrentWriteTransactions\":128},\"storage\":
{\"engine\":\"wiredTiger\",\"inMemory\":{\"engineConfig\":{\"inMemorySizeRatio\":0.9}},\"mmapv1\":
{\"nsSize\":16,\"smallfiles\":false},\"wiredTiger\":{\"collectionConfig\":{\"blockCompressor\":\"snappy\"},\"engineConfig\":
{\"cacheSizeRatio\":0.5,\"directoryForIndexes\":false,\"journalCompressor\":\"snappy\"},\"indexConfig\":{\"prefixCompression\":true}}}},\"pmm\":
{\"enabled\":false,\"image\":\"percona/percona-server-mongodb-operator:1.5.0-pmm\",\"serverHost\":\"monitoring-service\"},\"replsets\":
[{\"affinity\":{\"antiAffinityTopologyKey\":\"none\"},\"arbiter\":{\"affinity\":
{\"antiAffinityTopologyKey\":\"none\"},\"enabled\":false,\"size\":1},\"expose\":
{\"enabled\":false,\"exposeType\":\"LoadBalancer\"},\"name\":\"rs0\",\"podDisruptionBudget\":{\"maxUnavailable\":1},\"resources\":
{\"limits\":null},\"size\":3,\"volumeSpec\":{\"persistentVolumeClaim\":{\"accessModes\":[\"ReadWriteOnce\"],\"resources\":{\"requests\":
{\"storage\":\"3Gi\"}},\"storageClassName\":\"standard\"}}}],\"secrets\":{\"users\":\"my-cluster-name-
secrets\"},\"updateStrategy\":\"SmartUpdate\"}}\n"
      },
      "creationTimestamp":"2020-07-24T14:27:58Z",
```

```
      "generation":1,
      "managedFields":[
         {
            "apiVersion":"psmdb.percona.com/v1-5-0",
            "fieldsType":"FieldsV1",
            "fieldsV1":{
               "f:metadata":{
                  "f:annotations":{
                     ".":{

                     },
                     "f:kubectl.kubernetes.io/last-applied-configuration":{

                     }
                  }
               },
               "f:spec":{
                  ".":{

                  },
                  "f:allowUnsafeConfigurations":{

                  },
                  "f:backup":{
                     ".":{

                     },
                     "f:enabled":{

                     },
                     "f:image":{

                     },
                     "f:serviceAccountName":{

                     }
                  },
                  "f:image":{

                  },
                  "f:imagePullPolicy":{

                  },
                  "f:mongod":{
                     ".":{

                     },
                     "f:net":{
                        ".":{

                        },
                        "f:port":{

                        }
                     },
                     "f:operationProfiling":{
                        ".":{

                        },
                        "f:mode":{

                        },
                        "f:rateLimit":{

                        },
                        "f:slowOpThresholdMs":{

                        }
                     },
                     "f:security":{
                        ".":{

                        },
                        "f:enableEncryption":{

                        },
                        "f:encryptionCipherMode":{

                        },
                        "f:encryptionKeySecret":{

                        }
                     },
                     "f:setParameter":{
                        ".":{

                        },
                        "f:ttlMonitorSleepSecs":{

                        },
                        "f:wiredTigerConcurrentReadTransactions":{
```

```
         },
         "f:wiredTigerConcurrentWriteTransactions":{

         }
      },
      "f:storage":{
         ".":{

         },
         "f:engine":{

         },
         "f:inMemory":{
            ".":{

            },
            "f:engineConfig":{
               ".":{

               },
               "f:inMemorySizeRatio":{

               }
            }
         },
         "f:mmapv1":{
            ".":{

            },
            "f:nsSize":{

            }
         },
         "f:wiredTiger":{
            ".":{

            },
            "f:collectionConfig":{
               ".":{

               },
               "f:blockCompressor":{

               }
            },
            "f:engineConfig":{
               ".":{

               },
               "f:cacheSizeRatio":{

               },
               "f:journalCompressor":{

               }
            },
            "f:indexConfig":{
               ".":{

               },
               "f:prefixCompression":{

               }
            }
         }
      }
   },
   "f:pmm":{
      ".":{

      },
      "f:image":{

      },
      "f:serverHost":{

      }
   },
   "f:secrets":{
      ".":{

      },
      "f:users":{

      }
   },
   "f:updateStrategy":{

   }
}
```

```
            },
            "manager":"kubectl",
            "operation":"Update",
            "time":"2020-07-24T14:27:58Z"
         },
         {
            "apiVersion":"psmdb.percona.com/v1",
            "fieldsType":"FieldsV1",
            "fieldsV1":{
               "f:spec":{
                  "f:backup":{
                     "f:containerSecurityContext":{
                        ".":{

                        },
                        "f:runAsNonRoot":{

                        },
                        "f:runAsUser":{

                        }
                     },
                     "f:podSecurityContext":{
                        ".":{

                        },
                        "f:fsGroup":{

                        }
                     }
                  },
                  "f:clusterServiceDNSSuffix":{

                  },
                  "f:replsets":{

                  },
                  "f:runUid":{

                  },
                  "f:secrets":{
                     "f:ssl":{

                     },
                     "f:sslInternal":{

                     }
                  }
               },
               "f:status":{
                  ".":{

                  },
                  "f:conditions":{

                  },
                  "f:observedGeneration":{

                  },
                  "f:replsets":{
                     ".":{

                     },
                     "f:rs0":{
                        ".":{

                        },
                        "f:ready":{

                        },
                        "f:size":{

                        },
                        "f:status":{

                        }
                     }
                  },
                  "f:state":{

                  }
               }
            },
            "manager":"percona-server-mongodb-operator",
            "operation":"Update",
            "time":"2020-07-24T15:09:40Z"
         }
      ],
      "name":"my-cluster-name",
      "namespace":"default",
      "resourceVersion":"1274523",
```

```
      "selfLink":"/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbs/my-cluster-name",
      "uid":"5207e71a-c83f-4707-b892-63aa93fb615c"
   },
   "spec":{
      "allowUnsafeConfigurations":false,
      "backup":{
         "enabled":true,
         "image":"percona/percona-server-mongodb-operator:1.5.0-backup",
         "restartOnFailure":true,
         "serviceAccountName":"percona-server-mongodb-operator",
         "storages":null,
         "tasks":null
      },
      "image":"percona/percona-server-mongodb:4.2.8-8",
      "imagePullPolicy":"Always",
      "mongod":{
         "net":{
            "hostPort":0,
            "port":27017
         },
         "operationProfiling":{
            "mode":"slowOp",
            "rateLimit":100,
            "slowOpThresholdMs":100
         },
         "security":{
            "enableEncryption":true,
            "encryptionCipherMode":"AES256-CBC",
            "encryptionKeySecret":"my-cluster-name-mongodb-encryption-key",
            "redactClientLogData":false
         },
         "setParameter":{
            "ttlMonitorSleepSecs":60,
            "wiredTigerConcurrentReadTransactions":128,
            "wiredTigerConcurrentWriteTransactions":128
         },
         "storage":{
            "engine":"wiredTiger",
            "inMemory":{
               "engineConfig":{
                  "inMemorySizeRatio":0.9
               }
            },
            "mmapv1":{
               "nsSize":16,
               "smallfiles":false
            },
            "wiredTiger":{
               "collectionConfig":{
                  "blockCompressor":"snappy"
               },
               "engineConfig":{
                  "cacheSizeRatio":0.5,
                  "directoryForIndexes":false,
                  "journalCompressor":"snappy"
               },
               "indexConfig":{
                  "prefixCompression":true
               }
            }
         }
      },
      "pmm":{
         "enabled":false,
         "image":"percona/percona-server-mongodb-operator:1.5.0-pmm",
         "serverHost":"monitoring-service"
      },
      "replsets":[
         {
            "affinity":{
               "antiAffinityTopologyKey":"none"
            },
            "arbiter":{
               "affinity":{
                  "antiAffinityTopologyKey":"none"
               },
               "enabled":false,
               "size":1
            },
            "expose":{
               "enabled":false,
               "exposeType":"LoadBalancer"
            },
            "name":"rs0",
            "podDisruptionBudget":{
               "maxUnavailable":1
            },
            "resources":{
               "limits":null
            },
            "size":3,
```

```
        "volumeSpec":{
          "persistentVolumeClaim":{
            "accessModes":[
              "ReadWriteOnce"
            ],
            "resources":{
              "requests":{
                "storage":"3Gi"
              }
            },
            "storageClassName":"standard"
          }
        }
      }
    ],
    "secrets":{
      "users":"my-cluster-name-secrets"
    },
    "updateStrategy":"SmartUpdate"
  },
  "status":{
    "conditions":[
      {
        "lastTransitionTime":"2020-07-24T14:28:03Z",
        "status":"True",
        "type":"ClusterInitializing"
      },
      {
        "lastTransitionTime":"2020-07-24T14:28:39Z",
        "status":"True",
        "type":"Error"
      },
      {
        "lastTransitionTime":"2020-07-24T14:28:41Z",
        "status":"True",
        "type":"ClusterInitializing"
      },
      {
        "lastTransitionTime":"2020-07-24T14:28:41Z",
        "status":"True",
        "type":"Error"
      },
      {
        "lastTransitionTime":"2020-07-24T14:29:10Z",
        "status":"True",
        "type":"ClusterReady"
      },
      {
        "lastTransitionTime":"2020-07-24T14:49:46Z",
        "status":"True",
        "type":"ClusterInitializing"
      },
      {
        "lastTransitionTime":"2020-07-24T14:50:00Z",
        "status":"True",
        "type":"ClusterInitializing"
      },
      {
        "lastTransitionTime":"2020-07-24T14:52:31Z",
        "status":"True",
        "type":"ClusterInitializing"
      },
      {
        "lastTransitionTime":"2020-07-24T14:52:43Z",
        "status":"True",
        "type":"Error"
      },
      {
        "lastTransitionTime":"2020-07-24T14:53:01Z",
        "status":"True",
        "type":"ClusterInitializing"
      },
      {
        "lastTransitionTime":"2020-07-24T14:53:05Z",
        "status":"True",
        "type":"ClusterInitializing"
      },
      {
        "lastTransitionTime":"2020-07-24T14:53:05Z",
        "status":"True",
        "type":"ClusterReady"
      }
    ],
    "observedGeneration":1,
    "replsets":{
      "rs0":{
        "ready":3,
        "size":3,
        "status":"ready"
      }
    },
```

```
      "state":"ready"
   }
}
```


Scale up/down Percona Server for MongoDB cluster

Description:

```
Increase or decrease the size of the Percona Server for MongoDB cluster nodes to fit the current high availability needs
```

Kubectl Command:

```
kubectl patch psmdb my-cluster-name --type=merge --patch '{
"spec": {"replsets":{ "size": "5" }
}}'
```

URL:

```
https://$API_SERVER/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbs/my-cluster-name
```

Authentication:

```
Authorization: Bearer $KUBE_TOKEN
```

cURL Request:

```
curl -k -v -XPATCH "https://$API_SERVER/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbs/my-cluster-name" \
            -H "Authorization: Bearer $KUBE_TOKEN" \
            -H "Content-Type: application/merge-patch+json"
            -H "Accept: application/json" \
            -d '{
                  "spec": {"replsets":{ "size": "5" }
                  }}'
```

Request Body:

Example

```
{
"spec": {"replsets":{ "size": "5" }
}}
```

Input:

> **spec**:
>
> replsets
>
> 1. size (Int or String, Defaults: 3): `Specify the sie of the replsets cluster to scale up or down to`

Response:

Example


```
{
   "apiVersion":"psmdb.percona.com/v1",
   "kind":"PerconaServerMongoDB",
   "metadata":{
      "annotations":{
         "kubectl.kubernetes.io/last-applied-configuration":"{\"apiVersion\":\"psmdb.percona.com/v1-5-
0\",\"kind\":\"PerconaServerMongoDB\",\"metadata\":{\"annotations\":{},\"name\":\"my-cluster-name\",\"namespace\":\"default\"},\"spec\":
{\"allowUnsafeConfigurations\":false,\"backup\":{\"enabled\":true,\"image\":\"percona/percona-server-mongodb-operator:1.5.0-
backup\",\"restartOnFailure\":true,\"serviceAccountName\":\"percona-server-mongodb-
operator\",\"storages\":null,\"tasks\":null},\"image\":\"percona/percona-server-mongodb:4.2.8-8\",\"imagePullPolicy\":\"Always\",\"mongod\":
{\"net\":{\"hostPort\":0,\"port\":27017},\"operationProfiling\":{\"mode\":\"slowOp\",\"rateLimit\":100,\"slowOpThresholdMs\":100},\"security\":
{\"enableEncryption\":true,\"encryptionCipherMode\":\"AES256-CBC\",\"encryptionKeySecret\":\"my-cluster-name-mongodb-encryption-
key\",\"redactClientLogData\":false},\"setParameter\":
{\"ttlMonitorSleepSecs\":60,\"wiredTigerConcurrentReadTransactions\":128,\"wiredTigerConcurrentWriteTransactions\":128},\"storage\":
```

```
{\"engine\":\"wiredTiger\",\"inMemory\":{\"engineConfig\":{\"inMemorySizeRatio\":0.9}},\"mmapv1\":
{\"nsSize\":16,\"smallfiles\":false},\"wiredTiger\":{\"collectionConfig\":{\"blockCompressor\":\"snappy\"},\"engineConfig\":
{\"cacheSizeRatio\":0.5,\"directoryForIndexes\":false,\"journalCompressor\":\"snappy\"},\"indexConfig\":{\"prefixCompression\":true}}}},\"pmm\":
{\"enabled\":false,\"image\":\"percona/percona-server-mongodb-operator:1.5.0-pmm\",\"serverHost\":\"monitoring-service\"},\"replsets\":
[{\"affinity\":{\"antiAffinityTopologyKey\":\"none\"},\"arbiter\":{\"affinity\":
{\"antiAffinityTopologyKey\":\"none\"},\"enabled\":false,\"size\":1},\"expose\":
{\"enabled\":false,\"exposeType\":\"LoadBalancer\"},\"name\":\"rs0\",\"podDisruptionBudget\":{\"maxUnavailable\":1},\"resources\":
{\"limits\":null},\"size\":3,\"volumeSpec\":{\"persistentVolumeClaim\":{\"accessModes\":[\"ReadWriteOnce\"],\"resources\":{\"requests\":
{\"storage\":\"3Gi\"}},\"storageClassName\":\"standard\"}}}],\"secrets\":{\"users\":\"my-cluster-name-
secrets\"},\"updateStrategy\":\"SmartUpdate\"}}\n"
      },
      "creationTimestamp":"2020-07-24T14:27:58Z",
      "generation":4,
      "managedFields":[
         {
            "apiVersion":"psmdb.percona.com/v1-5-0",
            "fieldsType":"FieldsV1",
            "fieldsV1":{
               "f:metadata":{
                  "f:annotations":{
                     ".":{

                     },
                     "f:kubectl.kubernetes.io/last-applied-configuration":{

                     }
                  }
               },
               "f:spec":{
                  ".":{

                  },
                  "f:allowUnsafeConfigurations":{

                  },
                  "f:backup":{
                     ".":{

                     },
                     "f:enabled":{

                     },
                     "f:image":{

                     },
                     "f:serviceAccountName":{

                     }
                  },
                  "f:image":{

                  },
                  "f:imagePullPolicy":{

                  },
                  "f:mongod":{
                     ".":{

                     },
                     "f:net":{
                        ".":{

                        },
                        "f:port":{

                        }
                     },
                     "f:operationProfiling":{
                        ".":{

                        },
                        "f:mode":{

                        },
                        "f:rateLimit":{

                        },
                        "f:slowOpThresholdMs":{

                        }
                     },
                     "f:security":{
                        ".":{

                        },
                        "f:enableEncryption":{

                        },
                        "f:encryptionCipherMode":{

                        },
```

```
      "f:encryptionKeySecret":{

      }
    },
    "f:setParameter":{
      ".":{

      },
      "f:ttlMonitorSleepSecs":{

      },
      "f:wiredTigerConcurrentReadTransactions":{

      },
      "f:wiredTigerConcurrentWriteTransactions":{

      }
    },
    "f:storage":{
      ".":{

      },
      "f:engine":{

      },
      "f:inMemory":{
        ".":{

        },
        "f:engineConfig":{
          ".":{

          },
          "f:inMemorySizeRatio":{

          }
        }
      },
      "f:mmapv1":{
        ".":{

        },
        "f:nsSize":{

        }
      },
      "f:wiredTiger":{
        ".":{

        },
        "f:collectionConfig":{
          ".":{

          },
          "f:blockCompressor":{

          }
        },
        "f:engineConfig":{
          ".":{

          },
          "f:cacheSizeRatio":{

          },
          "f:journalCompressor":{

          }
        },
        "f:indexConfig":{
          ".":{

          },
          "f:prefixCompression":{

          }
        }
      }
    }
  },
  "f:pmm":{
    ".":{

    },
    "f:image":{

    },
    "f:serverHost":{

    }
  },
```

```
            "f:secrets":{
               ".":{

               },
               "f:users":{

               }
            },
            "f:updateStrategy":{

            }
         }
      },
      "manager":"kubectl",
      "operation":"Update",
      "time":"2020-07-24T14:27:58Z"
   },
   {
      "apiVersion":"psmdb.percona.com/v1",
      "fieldsType":"FieldsV1",
      "fieldsV1":{
         "f:spec":{
            "f:backup":{
               "f:containerSecurityContext":{
                  ".":{

                  },
                  "f:runAsNonRoot":{

                  },
                  "f:runAsUser":{

                  }
               },
               "f:podSecurityContext":{
                  ".":{

                  },
                  "f:fsGroup":{

                  }
               }
            },
            "f:clusterServiceDNSSuffix":{

            },
            "f:runUid":{

            },
            "f:secrets":{
               "f:ssl":{

               },
               "f:sslInternal":{

               }
            }
         },
         "f:status":{
            ".":{

            },
            "f:conditions":{

            },
            "f:observedGeneration":{

            },
            "f:replsets":{
               ".":{

               },
               "f:rs0":{
                  ".":{

                  },
                  "f:ready":{

                  },
                  "f:size":{

                  },
                  "f:status":{

                  }
               }
            },
            "f:state":{

            }
         }
```

```
        },
        "manager":"percona-server-mongodb-operator",
        "operation":"Update",
        "time":"2020-07-24T15:35:14Z"
      },
      {
        "apiVersion":"psmdb.percona.com/v1",
        "fieldsType":"FieldsV1",
        "fieldsV1":{
          "f:spec":{
            "f:replsets":{
              ".":{

              },
              "f:size":{

              }
            }
          }
        },
        "manager":"kubectl",
        "operation":"Update",
        "time":"2020-07-24T15:43:19Z"
      }
    ],
    "name":"my-cluster-name",
    "namespace":"default",
    "resourceVersion":"1279009",
    "selfLink":"/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbs/my-cluster-name",
    "uid":"5207e71a-c83f-4707-b892-63aa93fb615c"
  },
  "spec":{
    "allowUnsafeConfigurations":false,
    "backup":{
      "enabled":true,
      "image":"percona/percona-server-mongodb-operator:1.5.0-backup",
      "restartOnFailure":true,
      "serviceAccountName":"percona-server-mongodb-operator",
      "storages":null,
      "tasks":null
    },
    "image":"percona/percona-server-mongodb:4.2.8-8",
    "imagePullPolicy":"Always",
    "mongod":{
      "net":{
        "hostPort":0,
        "port":27017
      },
      "operationProfiling":{
        "mode":"slowOp",
        "rateLimit":100,
        "slowOpThresholdMs":100
      },
      "security":{
        "enableEncryption":true,
        "encryptionCipherMode":"AES256-CBC",
        "encryptionKeySecret":"my-cluster-name-mongodb-encryption-key",
        "redactClientLogData":false
      },
      "setParameter":{
        "ttlMonitorSleepSecs":60,
        "wiredTigerConcurrentReadTransactions":128,
        "wiredTigerConcurrentWriteTransactions":128
      },
      "storage":{
        "engine":"wiredTiger",
        "inMemory":{
          "engineConfig":{
            "inMemorySizeRatio":0.9
          }
        },
        "mmapv1":{
          "nsSize":16,
          "smallfiles":false
        },
        "wiredTiger":{
          "collectionConfig":{
            "blockCompressor":"snappy"
          },
          "engineConfig":{
            "cacheSizeRatio":0.5,
            "directoryForIndexes":false,
            "journalCompressor":"snappy"
          },
          "indexConfig":{
            "prefixCompression":true
          }
        }
      }
    },
    "pmm":{
```

```
         "enabled":false,
         "image":"percona/percona-server-mongodb-operator:1.5.0-pmm",
         "serverHost":"monitoring-service"
      },
      "replsets":{
         "size":"5"
      },
      "secrets":{
         "users":"my-cluster-name-secrets"
      },
      "updateStrategy":"SmartUpdate"
   },
   "status":{
      "conditions":[
         {
            "lastTransitionTime":"2020-07-24T14:28:03Z",
            "status":"True",
            "type":"ClusterInitializing"
         },
         {
            "lastTransitionTime":"2020-07-24T14:28:39Z",
            "status":"True",
            "type":"Error"
         },
         {
            "lastTransitionTime":"2020-07-24T14:28:41Z",
            "status":"True",
            "type":"ClusterInitializing"
         },
         {
            "lastTransitionTime":"2020-07-24T14:28:41Z",
            "status":"True",
            "type":"Error"
         },
         {
            "lastTransitionTime":"2020-07-24T14:29:10Z",
            "status":"True",
            "type":"ClusterReady"
         },
         {
            "lastTransitionTime":"2020-07-24T14:49:46Z",
            "status":"True",
            "type":"ClusterInitializing"
         },
         {
            "lastTransitionTime":"2020-07-24T14:50:00Z",
            "status":"True",
            "type":"ClusterInitializing"
         },
         {
            "lastTransitionTime":"2020-07-24T14:52:31Z",
            "status":"True",
            "type":"ClusterInitializing"
         },
         {
            "lastTransitionTime":"2020-07-24T14:52:43Z",
            "status":"True",
            "type":"Error"
         },
         {
            "lastTransitionTime":"2020-07-24T14:53:01Z",
            "status":"True",
            "type":"ClusterInitializing"
         },
         {
            "lastTransitionTime":"2020-07-24T14:53:05Z",
            "status":"True",
            "type":"ClusterInitializing"
         },
         {
            "lastTransitionTime":"2020-07-24T14:53:05Z",
            "status":"True",
            "type":"ClusterReady"
         }
      ],
      "observedGeneration":1,
      "replsets":{
         "rs0":{
            "ready":3,
            "size":3,
            "status":"ready"
         }
      },
      "state":"ready"
   }
}
```

Update Percona Server for MongoDB cluster image

Description:

```
Change the image of Percona Server for MongoDB containers inside the cluster
```

Kubectl Command:

```
kubectl patch psmdb my-cluster-name --type=merge --patch '{
"spec": {"psmdb":{ "image": "percona/percona-server-mongodb-operator:1.4.0-mongod4.2" }
}}'
```

URL:

```
https://$API_SERVER/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbs/my-cluster-name
```

Authentication:

```
Authorization: Bearer $KUBE_TOKEN
```

cURL Request:

```
curl -k -v -XPATCH "https://$API_SERVER/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbs/my-cluster-name" \
            -H "Authorization: Bearer $KUBE_TOKEN" \
            -H "Accept: application/json" \
            -H "Content-Type: application/merge-patch+json"
            -d '{
              "spec": {"psmdb":{ "image": "percona/percona-server-mongodb-operator:1.4.0-mongod4.2" }
              }}'
```

Request Body:

Example

```
{
"spec": { "image ": "percona/percona-server-mongodb:4.2.8-8" }
}
```

Input:

> **spec**:
>
> psmdb:
>
> 1. image (String, min-length: 1) : `name of the image to update for Percona Server for MongoDB`

Response:

Example


```
{
   "apiVersion":"psmdb.percona.com/v1",
   "kind":"PerconaServerMongoDB",
   "metadata":{
      "annotations":{
         "kubectl.kubernetes.io/last-applied-configuration":"{\"apiVersion\":\"psmdb.percona.com/v1-5-
0\",\"kind\":\"PerconaServerMongoDB\",\"metadata\":{\"annotations\":{},\"name\":\"my-cluster-name\",\"namespace\":\"default\"},\"spec\":
{\"allowUnsafeConfigurations\":false,\"backup\":{\"enabled\":true,\"image\":\"percona/percona-server-mongodb-operator:1.5.0-
backup\",\"restartOnFailure\":true,\"serviceAccountName\":\"percona-server-mongodb-
operator\",\"storages\":null,\"tasks\":null},\"image\":\"percona/percona-server-mongodb:4.2.8-8\",\"imagePullPolicy\":\"Always\",\"mongod\":
{\"net\":{\"hostPort\":0,\"port\":27017},\"operationProfiling\":{\"mode\":\"slowOp\",\"rateLimit\":100,\"slowOpThresholdMs\":100},\"security\":
{\"enableEncryption\":true,\"encryptionCipherMode\":\"AES256-CBC\",\"encryptionKeySecret\":\"my-cluster-name-mongodb-encryption-
key\",\"redactClientLogData\":false},\"setParameter\":
{\"ttlMonitorSleepSecs\":60,\"wiredTigerConcurrentReadTransactions\":128,\"wiredTigerConcurrentWriteTransactions\":128},\"storage\":
{\"engine\":\"wiredTiger\",\"inMemory\":{\"engineConfig\":{\"inMemorySizeRatio\":0.9}},\"mmapv1\":
{\"nsSize\":16,\"smallfiles\":false},\"wiredTiger\":{\"collectionConfig\":{\"blockCompressor\":\"snappy\"},\"engineConfig\":
{\"cacheSizeRatio\":0.5,\"directoryForIndexes\":false,\"journalCompressor\":\"snappy\"},\"indexConfig\":{\"prefixCompression\":true}}}},\"pmm\":
{\"enabled\":false,\"image\":\"percona/percona-server-mongodb-operator:1.5.0-pmm\",\"serverHost\":\"monitoring-service\"},\"replsets\":
[{\"affinity\":{\"antiAffinityTopologyKey\":\"none\"},\"arbiter\":{\"affinity\":
{\"antiAffinityTopologyKey\":\"none\"},\"enabled\":false,\"size\":1},\"expose\":
{\"enabled\":false,\"exposeType\":\"LoadBalancer\"},\"name\":\"rs0\",\"podDisruptionBudget\":{\"maxUnavailable\":1},\"resources\":
```

```
{\"limits\":null},\"size\":3,\"volumeSpec\":{\"persistentVolumeClaim\":{\"accessModes\":[\"ReadWriteOnce\"],\"resources\":{\"requests\":
{\"storage\":\"3Gi\"}},\"storageClassName\":\"standard\"}}}],\"secrets\":{\"users\":\"my-cluster-name-
secrets\"},\"updateStrategy\":\"SmartUpdate\"}}\n"
      },
      "creationTimestamp":"2020-07-24T14:27:58Z",
      "generation":5,
      "managedFields":[
         {
            "apiVersion":"psmdb.percona.com/v1-5-0",
            "fieldsType":"FieldsV1",
            "fieldsV1":{
               "f:metadata":{
                  "f:annotations":{
                     ".":{

                     },
                     "f:kubectl.kubernetes.io/last-applied-configuration":{

                     }
                  }
               },
               "f:spec":{
                  ".":{

                  },
                  "f:allowUnsafeConfigurations":{

                  },
                  "f:backup":{
                     ".":{

                     },
                     "f:enabled":{

                     },
                     "f:image":{

                     },
                     "f:serviceAccountName":{

                     }
                  },
                  "f:image":{

                  },
                  "f:imagePullPolicy":{

                  },
                  "f:mongod":{
                     ".":{

                     },
                     "f:net":{
                        ".":{

                        },
                        "f:port":{

                        }
                     },
                     "f:operationProfiling":{
                        ".":{

                        },
                        "f:mode":{

                        },
                        "f:rateLimit":{

                        },
                        "f:slowOpThresholdMs":{

                        }
                     },
                     "f:security":{
                        ".":{

                        },
                        "f:enableEncryption":{

                        },
                        "f:encryptionCipherMode":{

                        },
                        "f:encryptionKeySecret":{

                        }
                     },
                     "f:setParameter":{
                        ".":{
```

```
         },
         "f:ttlMonitorSleepSecs":{

         },
         "f:wiredTigerConcurrentReadTransactions":{

         },
         "f:wiredTigerConcurrentWriteTransactions":{

         }
      },
      "f:storage":{
         ".":{

         },
         "f:engine":{

         },
         "f:inMemory":{
            ".":{

            },
            "f:engineConfig":{
               ".":{

               },
               "f:inMemorySizeRatio":{

               }
            }
         },
         "f:mmapv1":{
            ".":{

            },
            "f:nsSize":{

            }
         },
         "f:wiredTiger":{
            ".":{

            },
            "f:collectionConfig":{
               ".":{

               },
               "f:blockCompressor":{

               }
            },
            "f:engineConfig":{
               ".":{

               },
               "f:cacheSizeRatio":{

               },
               "f:journalCompressor":{

               }
            },
            "f:indexConfig":{
               ".":{

               },
               "f:prefixCompression":{

               }
            }
         }
      }
   },
   "f:pmm":{
      ".":{

      },
      "f:image":{

      },
      "f:serverHost":{

      }
   },
   "f:secrets":{
      ".":{

      },
      "f:users":{

      }
```

```
            },
            "f:updateStrategy":{

            }
         }
      },
      "manager":"kubectl",
      "operation":"Update",
      "time":"2020-07-24T14:27:58Z"
   },
   {
      "apiVersion":"psmdb.percona.com/v1",
      "fieldsType":"FieldsV1",
      "fieldsV1":{
         "f:spec":{
            "f:backup":{
               "f:containerSecurityContext":{
                  ".":{

                  },
                  "f:runAsNonRoot":{

                  },
                  "f:runAsUser":{

                  }
               },
               "f:podSecurityContext":{
                  ".":{

                  },
                  "f:fsGroup":{

                  }
               }
            },
            "f:clusterServiceDNSSuffix":{

            },
            "f:runUid":{

            },
            "f:secrets":{
               "f:ssl":{

               },
               "f:sslInternal":{

               }
            }
         },
         "f:status":{
            ".":{

            },
            "f:conditions":{

            },
            "f:observedGeneration":{

            },
            "f:replsets":{
               ".":{

               },
               "f:rs0":{
                  ".":{

                  },
                  "f:ready":{

                  },
                  "f:size":{

                  },
                  "f:status":{

                  }
               }
            },
            "f:state":{

            }
         }
      },
      "manager":"percona-server-mongodb-operator",
      "operation":"Update",
      "time":"2020-07-24T15:35:14Z"
   },
   {
      "apiVersion":"psmdb.percona.com/v1",
```

```
            "fieldsType":"FieldsV1",
            "fieldsV1":{
               "f:spec":{
                  "f:image ":{

                  },
                  "f:replsets":{
                     ".":{

                     },
                     "f:size":{

                     }
                  }
               }
            },
            "manager":"kubectl",
            "operation":"Update",
            "time":"2020-07-27T12:21:39Z"
         }
      ],
      "name":"my-cluster-name",
      "namespace":"default",
      "resourceVersion":"1279853",
      "selfLink":"/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbs/my-cluster-name",
      "uid":"5207e71a-c83f-4707-b892-63aa93fb615c"
   },
   "spec":{
      "allowUnsafeConfigurations":false,
      "backup":{
         "enabled":true,
         "image":"percona/percona-server-mongodb-operator:1.5.0-backup",
         "restartOnFailure":true,
         "serviceAccountName":"percona-server-mongodb-operator",
         "storages":null,
         "tasks":null
      },
      "image ":"percona/percona-server-mongodb:4.2.8-8",
      "imagePullPolicy":"Always",
      "mongod":{
         "net":{
            "hostPort":0,
            "port":27017
         },
         "operationProfiling":{
            "mode":"slowOp",
            "rateLimit":100,
            "slowOpThresholdMs":100
         },
         "security":{
            "enableEncryption":true,
            "encryptionCipherMode":"AES256-CBC",
            "encryptionKeySecret":"my-cluster-name-mongodb-encryption-key",
            "redactClientLogData":false
         },
         "setParameter":{
            "ttlMonitorSleepSecs":60,
            "wiredTigerConcurrentReadTransactions":128,
            "wiredTigerConcurrentWriteTransactions":128
         },
         "storage":{
            "engine":"wiredTiger",
            "inMemory":{
               "engineConfig":{
                  "inMemorySizeRatio":0.9
               }
            },
            "mmapv1":{
               "nsSize":16,
               "smallfiles":false
            },
            "wiredTiger":{
               "collectionConfig":{
                  "blockCompressor":"snappy"
               },
               "engineConfig":{
                  "cacheSizeRatio":0.5,
                  "directoryForIndexes":false,
                  "journalCompressor":"snappy"
               },
               "indexConfig":{
                  "prefixCompression":true
               }
            }
         }
      },
      "pmm":{
         "enabled":false,
         "image":"percona/percona-server-mongodb-operator:1.5.0-pmm",
         "serverHost":"monitoring-service"
      },
```

```
      "replsets":{
         "size":"5"
      },
      "secrets":{
         "users":"my-cluster-name-secrets"
      },
      "updateStrategy":"SmartUpdate"
   },
   "status":{
      "conditions":[
         {
            "lastTransitionTime":"2020-07-24T14:28:03Z",
            "status":"True",
            "type":"ClusterInitializing"
         },
         {
            "lastTransitionTime":"2020-07-24T14:28:39Z",
            "status":"True",
            "type":"Error"
         },
         {
            "lastTransitionTime":"2020-07-24T14:28:41Z",
            "status":"True",
            "type":"ClusterInitializing"
         },
         {
            "lastTransitionTime":"2020-07-24T14:28:41Z",
            "status":"True",
            "type":"Error"
         },
         {
            "lastTransitionTime":"2020-07-24T14:29:10Z",
            "status":"True",
            "type":"ClusterReady"
         },
         {
            "lastTransitionTime":"2020-07-24T14:49:46Z",
            "status":"True",
            "type":"ClusterInitializing"
         },
         {
            "lastTransitionTime":"2020-07-24T14:50:00Z",
            "status":"True",
            "type":"ClusterInitializing"
         },
         {
            "lastTransitionTime":"2020-07-24T14:52:31Z",
            "status":"True",
            "type":"ClusterInitializing"
         },
         {
            "lastTransitionTime":"2020-07-24T14:52:43Z",
            "status":"True",
            "type":"Error"
         },
         {
            "lastTransitionTime":"2020-07-24T14:53:01Z",
            "status":"True",
            "type":"ClusterInitializing"
         },
         {
            "lastTransitionTime":"2020-07-24T14:53:05Z",
            "status":"True",
            "type":"ClusterInitializing"
         },
         {
            "lastTransitionTime":"2020-07-24T14:53:05Z",
            "status":"True",
            "type":"ClusterReady"
         }
      ],
      "observedGeneration":1,
      "replsets":{
         "rs0":{
            "ready":3,
            "size":3,
            "status":"ready"
         }
      },
      "state":"ready"
   }
}
```

Backup Percona Server for MongoDB cluster

Description:

```
Takes a backup of the Percona Server for MongoDB cluster containers data to be able to recover from disasters or make a roll-back later
```

Kubectl Command:

```
kubectl apply -f percona-server-mongodb-operator/deploy/backup/backup.yaml
```

URL:

```
https://$API_SERVER/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbbackups
```

Authentication:

```
Authorization: Bearer $KUBE_TOKEN
```

cURL Request:

```
curl -k -v -XPOST "https://$API_SERVER/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbbackups" \
            -H "Accept: application/json" \
            -H "Content-Type: application/json" \
            -d "@backup.json" -H "Authorization: Bearer $KUBE_TOKEN"
```

Request Body (backup.json):

Example


```
{
   "apiVersion":"psmdb.percona.com/v1",
   "kind":"PerconaServerMongoDBBackup",
   "metadata":{
      "name":"backup1",
      "namespace":"default"
   },
   "spec":{
      "psmdbCluster":"my-cluster-name",
      "storageName":"s3-us-west"
   }
}
```


Input:

1. **metadata**:

> name(String, min-length:1) : `name of backup to create`

1. **spec**:

```
1. psmdbCluster(String, min-length:1) : `name of Percona Server for MongoDB cluster`

2. storageName(String, min-length:1) : `name of storage claim to use`
```

Response:

Example


```
{
   "apiVersion":"psmdb.percona.com/v1",
   "kind":"PerconaServerMongoDBBackup",
   "metadata":{
      "annotations":{
         "kubectl.kubernetes.io/last-applied-configuration":"
{\"apiVersion\":\"psmdb.percona.com/v1\",\"kind\":\"PerconaServerMongoDBBackup\",\"metadata\":{\"annotations\":
{},\"name\":\"backup1\",\"namespace\":\"default\"},\"spec\":{\"psmdbCluster\":\"my-cluster-name\",\"storageName\":\"s3-us-west\"}}\n"
      },
      "creationTimestamp":"2020-07-27T13:45:43Z",
      "generation":1,
      "managedFields":[
         {
            "apiVersion":"psmdb.percona.com/v1",
            "fieldsType":"FieldsV1",
            "fieldsV1":{
               "f:metadata":{
                  "f:annotations":{
                     ".":{

                     },
                     "f:kubectl.kubernetes.io/last-applied-configuration":{

                     }
                  }
               },
               "f:spec":{
                  ".":{

                  },
                  "f:psmdbCluster":{

                  },
                  "f:storageName":{

                  }
               }
            },
            "manager":"kubectl",
            "operation":"Update",
            "time":"2020-07-27T13:45:43Z"
         }
      ],
      "name":"backup1",
      "namespace":"default",
      "resourceVersion":"1290243",
      "selfLink":"/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbbackups/backup1",
      "uid":"e695d1c7-898e-44b0-b356-537284f6c046"
   },
   "spec":{
      "psmdbCluster":"my-cluster-name",
      "storageName":"s3-us-west"
   }
}
```


Restore Percona Server for MongoDB cluster

Description:

```
Restores Percona Server for MongoDB cluster data to an earlier version to recover from a problem or to make a roll-back
```

Kubectl Command:

```
kubectl apply -f percona-server-mongodb-operator/deploy/backup/restore.yaml
```

URL:

```
https://$API_SERVER/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbrestores
```

Authentication:

```
Authorization: Bearer $KUBE_TOKEN
```

cURL Request:

```
curl -k -v -XPOST "https://$API_SERVER/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbrestores" \
            -H "Accept: application/json" \
            -H "Content-Type: application/json" \
            -d "@restore.json" \
            -H "Authorization: Bearer $KUBE_TOKEN"
```

Request Body (restore.json):

Example

```
{
   "apiVersion":"psmdb.percona.com/v1",
   "kind":"PerconaServerMongoDBRestore",
   "metadata":{
      "name":"restore1",
      "namespace":"default"
   },
   "spec":{
      "backupName":"backup1",
      "clusterName":"my-cluster-name"
   }
}
```

Input:

1. **metadata**:

> name(String, min-length:1): `name of restore to create`

1. **spec**:

```
1. clusterName(String, min-length:1) : `name of Percona Server for MongoDB cluster`

2. backupName(String, min-length:1) : `name of backup to restore from`
```

Response:

Example


```
{
   "apiVersion":"psmdb.percona.com/v1",
   "kind":"PerconaServerMongoDBRestore",
   "metadata":{
      "annotations":{
         "kubectl.kubernetes.io/last-applied-configuration":"
{\"apiVersion\":\"psmdb.percona.com/v1\",\"kind\":\"PerconaServerMongoDBRestore\",\"metadata\":{\"annotations\":
{},\"name\":\"restore1\",\"namespace\":\"default\"},\"spec\":{\"backupName\":\"backup1\",\"clusterName\":\"my-cluster-name\"}}\n"
      },
      "creationTimestamp":"2020-07-27T13:52:56Z",
      "generation":1,
      "managedFields":[
         {
            "apiVersion":"psmdb.percona.com/v1",
            "fieldsType":"FieldsV1",
            "fieldsV1":{
               "f:metadata":{
                  "f:annotations":{
                     ".":{

                     },
                     "f:kubectl.kubernetes.io/last-applied-configuration":{

                     }
                  }
               },
               "f:spec":{
                  ".":{

                  },
                  "f:backupName":{

                  },
                  "f:clusterName":{

                  }
               }
            },
            "manager":"kubectl",
            "operation":"Update",
            "time":"2020-07-27T13:52:56Z"
         }
      ],
      "name":"restore1",
      "namespace":"default",
      "resourceVersion":"1291198",
      "selfLink":"/apis/psmdb.percona.com/v1/namespaces/default/perconaservermongodbrestores/restore1",
      "uid":"17e982fe-ac41-47f4-afba-fea380b0c76e"
   },
   "spec":{
      "backupName":"backup1",
      "clusterName":"my-cluster-name"
   }
}
```

11.8 Frequently Asked Questions

Why do we need to follow "the Kubernetes way" when Kubernetes was never intended to run databases?

As it is well known, the Kubernetes approach is targeted at stateless applications but provides ways to store state (in Persistent Volumes, etc.) if the application needs it. Generally, a stateless mode of operation is supposed to provide better safety, sustainability, and scalability, it makes the already-deployed components interchangeable. You can find more about substantial benefits brought by Kubernetes to databases in [this blog post]().

The architecture of state-centric applications (like databases) should be composed in a right way to avoid crashes, data loss, or data inconsistencies during hardware failure. Percona Operator for MongoDB provides out-of-the-box functionality to automate provisioning and management of highly available MongoDB database clusters on Kubernetes.

How can I contact the developers?

The best place to discuss Percona Operator for MongoDB with developers and other community members is the [community forum]().

If you would like to report a bug, use the Percona Operator for MongoDB [project in JIRA]().

What is the difference between the Operator quickstart and advanced installation ways?

As you have noticed, the installation section of docs contains both quickstart and advanced installation guides.

The quickstart guide is simpler. It has fewer installation steps in favor of predefined default choices. Particularly, in advanced installation guides, you separately apply the Custom Resource Definition and Role-based Access Control configuration files with possible edits in them. At the same time, quickstart guides rely on the all-inclusive bundle configuration.

At another point, quickstart guides are related to specific platforms you are going to use (Minikube, Google Kubernetes Engine, etc.) and therefore include some additional steps needed for these platforms.

Generally, rely on the quickstart guide if you are a beginner user of the specific platform and/or you are new to the Percona Operator for MongoDB as a whole.

Which versions of MongoDB does the Operator support?

Percona Operator for MongoDB works with all active major versions of Percona Server for MongoDB. The exact version is determined by the Docker image in use.

The Operator uses Percona-certified Docker images, which you can find [here](). For example, Percona Server for MongoDB 8.0 is supported with the following recommended version: 8.0.19-7.

Check the [Percona Server for MongoDB release notes]() for more details on the exact version of the latest major version. Use the version switcher to check other major versions.

How can I add custom sidecar containers to my cluster?

The Operator allows you to deploy additional containers to the Pod. Such containers are called *sidecar* containers.

You can use sidecar containers to run debugging tools, some specific monitoring solutions, etc. Add such sidecar container to the `deploy/cr.yaml` configuration file, specifying its name and image, and possibly a command to run:

```
spec:
  replsets:
  - name: rs0
    ....
    sidecars:
    - image: busybox
      command: ["/bin/sh"]
      args: ["-c", "while true; do echo echo $(date -u) 'test' >> /dev/null; sleep 5; done"]
      name: rs-sidecar-1
    ....
```

You can add `sidecars` subsection to `replsets`, `sharding.configsvrReplSet`, and `sharding.mongos` sections.

> **Note**
>
> Custom sidecar containers [can easily access other components of your cluster](). Therefore they should be used carefully and by experienced users only.

Find more information on sidecar containers in the appropriate documentation page.

How to provoke the initial sync of a Pod?

There are certain situations where it might be necessary to delete all MongoDB instance data to force the resync. For example, there may be the following reasons:

- rebuilding the node to defragment the database,
- recreating the member failing to sync due to some bug.

In the case of a "regular" MongoDB, wiping the dbpath would trigger such resync. In the case of a MongoDB cluster controlled by the Operator, you will need to do the following steps:

1. Find out the names of the Persistent Volume Claim and Pod you are going to delete (use `kubectl get pvc` command for PVC and `kubectl get pod` one for Pods).
2. Delete the appropriate PVC and Pod. For example, wiping out the `my-cluster-name-rs0-2` Pod should look as follows:

```
kubectl delete pod/my-cluster-name-rs0-2 pvc/mongod-data-my-cluster-name-rs0-2
```

The Operator will automatically recreate the needed Pod and PVC after deletion.

11.9 Copyright and licensing information

Documentation licensing

11.10 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.

12 Release notes

12.1 Percona Operator for MongoDB Release Notes

- Percona Operator for MongoDB 1.22.0 (2026-02-25)
- Percona Operator for MongoDB 1.21.2 (2026-01-12)
- Percona Operator for MongoDB 1.21.1 (2025-10-30)
- Percona Operator for MongoDB 1.21.0 (2025-10-20)
- Percona Operator for MongoDB 1.20.1 (2025-06-04)
- Percona Operator for MongoDB 1.20.0 (2025-05-19)
- Percona Operator for MongoDB 1.19.1 (2025-02-20)
- Percona Operator for MongoDB 1.19.0 (2025-01-21)
- Percona Operator for MongoDB 1.18.0 (2024-11-14)Vault integration for system user password management You can now integrate the Operator with HashiCorp Vault for system user password management. This allows organizations to centralize password management while keeping the Operator responsible for applying those passwords to the database. When this integration is enabled, the Operator authenticates to Vault using either the Kubernetes authentication method or a Vault token. It retrieves system user passwords during cluster creation and generates the corresponding Kubernetes Secret from this data. The Operator periodically checks Vault for password changes and updates the Secret when differences are detected. If Vault is temporarily unavailable or the Operator cannot retrieve the passwords, it logs the event and continues cluster reconciliation to ensure the cluster availability. Organizations benefit from this integration when they need: · Centralized credential governance · Auditable password rotation · Compliance with internal security policies · Separation of duties (DBA vs. security team) · Consistent password lifecycle management across environments Vault becomes the single source of truth, while the Operator ensures Percona Server for MongoDB always uses the correct credentials.
- Percona Operator for MongoDB 1.17.0 (2024-09-09)
- Percona Operator for MongoDB 1.16.2 (2024-07-23)
- Percona Operator for MongoDB 1.16.1 (2024-06-24)
- Percona Operator for MongoDB 1.16.0 (2024-05-24)
- Percona Operator for MongoDB 1.15.0 (2023-10-09)
- Percona Operator for MongoDB 1.14.0 (2023-03-13)
- Percona Operator for MongoDB 1.13.0 (2022-09-08)
- Percona Operator for MongoDB 1.12.0 (2022-05-05)
- *Percona Distribution for MongoDB Operator* 1.11.0 (2021-12-21)
- *Percona Distribution for MongoDB Operator* 1.10.0 (2021-09-30)
- *Percona Distribution for MongoDB Operator* 1.9.0 (2021-07-29)
- *Percona Kubernetes Operator for Percona Server for MongoDB* 1.8.0 (2021-05-06)
- *Percona Kubernetes Operator for Percona Server for MongoDB* 1.7.0 (2021-03-08)
- *Percona Kubernetes Operator for Percona Server for MongoDB* 1.6.0 (2020-12-22)
- *Percona Kubernetes Operator for Percona Server for MongoDB* 1.5.0 (2020-09-07)
- *Percona Kubernetes Operator for Percona Server for MongoDB* 1.4.0 (2020-03-31)
- *Percona Kubernetes Operator for Percona Server for MongoDB* 1.3.0 (2019-12-11)
- *Percona Kubernetes Operator for Percona Server for MongoDB* 1.2.0 (2019-09-20)
- *Percona Kubernetes Operator for Percona Server for MongoDB* 1.1.0 (2019-07-15)
- *Percona Kubernetes Operator for Percona Server for MongoDB* 1.0.0 (2019-05-29)

Percona Operator for MongoDB 1.22.0 (2026-02-25)

Get started with the Operator → Upgrade →

What's new at a glance

Backup and restore

- Restore into clusters with different replica set names using remapping
- Configure a longer PBM startup timeout to prevent false backup failures
- Use the `minio` storage type for backups to S3-compatible storage services
- Verify TLS communication for a S3-compatible storage with a private CA certificates
- Track PBM readiness via the new `PBMReady` status condition

Operations

- Automatically resize PVCs based on usage thresholds
- Configure log rotation for persistent logs
- Define custom environment variables for `mongod`, `mongos`, and `logcollector`
- Run pre-start `hookScript` logic for `mongod`, `mongos`, and `pbm-agent`

Automation and integrations

- Better service mesh compatibility with automatic `appProtocol: mongo` on services
- Improved GitOps workflows via automated CRD upgrades from a dedicated Helm chart
- Disable authentication for dev/test environments to speed up setup
- Integrate with HashCorp Vault for system users management

Release Highlights

Ensure smooth integration with service meshes

The Operator now supports the Kubernetes `appProtocol` field and automatically sets the `mongo` application layer protocol on all services.

For users running MongoDB with service meshes like Istio, this enhancement ensures that the mesh can correctly recognize and interpret MongoDB traffic. The result is reliable cluster formation and secure mutual TLS (mTLS) connections in Kubernetes environments.

With this enhancement, you gain the full benefits of service meshes such as advanced traffic management, observability, and stronger security without manual configuration or workarounds.

Restore to a cluster with different replica set names

You can now make physical and logical restores to clusters with different replica set names using the Operator. This feature builds on the implementation in Percona Backup for MongoDB and is now embedded directly into the Operator.

To make a restore, define source and target replica set mappings directly in your `PerconaServerMongoDBRestore` object:

```
apiVersion: psmdb.percona.com/v1
kind: PerconaServerMongoDBRestore
metadata:
  name: restore-with-remapping
spec:
  clusterName: some-name-target
  backupName: backup1
  replsetRemapping:
    shard0: rs0
    shard1: rs1
    csReplSet: cfg
```

Define mappings as key-value pairs where the key is the source replica set name and the value is the target replica set name. Read more about how to set it up in the documentation.

This enhancement expands the number of environments where restores are possible, making recovery more flexible and reliable.

Speed up development or testing pipelines by disabling authentication

You can now disable authentication in Percona Server for MongoDB clusters managed by the Operator. This helps you:

- Simplify setup in local environments and speed up development
- Simplify migration from external MongoDB instances that don't use authentication
- Debug authentication issues without credential complexity
- Connect applications that don't support MongoDB authentication

Disable authentication by adding `security.authorization: disabled` to your MongoDB component configuration in the Custom Resource. For sharded clusters, set this for each replica set, the config server replica set, and `mongos`. See the documentation for preconditions and step-by-step guidance.

Note that disabling authentication poses security risk, that's why use this feature with caution.

Ability to define custom environment variables

You can now use environment variables to parameterize and control database Pod behavior without hardcoding values in manifests.

This is useful when you need to:

- Align container behavior with your platform (cache size settings, time zone, locale)
- Pass non-sensitive runtime flags to custom entrypoints
- Provide credentials or third-party API tokens from a Secret without baking them into images

You can set environment variables for `mongod`, `mongos`, and `logcollector` containers in these ways:

- Directly in the Custom Resource
- Via a ConfigMap
- Via a Secret

Custom environment variables make deployments flexible, reproducible, and environment-aware. Instead of maintaining separate manifests for each environment, you inject variables dynamically.

See our documentation on how to define custom environment variables.

Automatic CRD updates for Helm installations

By default all components for the Operator Deployment are installed from the same Helm repository. To prevent accidental CRD updates, Helm v3 doesn't automatically update the CRDs during the upgrade, and you must update them manually, after you update the Operator deployment.

Starting with version 1.22.0, the Operator has a separate `psmdb-operator-crds` chart for CRDs. This lets you:

- Manage CRDs with Helm instead of updating them manually, reducing overhead and human error
- Ensure compatibility with GitOps tools like ArgoCD and FluxCD that expect Helm to manage all resources
- Take advantage of Helm's version control and rollback capabilities

You can use the CRD chart on new installs and add it to your upgrade workflow. See the upgrade documentation for step-by-step instructions.

This change is fully compatible with the previous behavior, where CRDs are installed from the `crds/` folder of the main Helm chart. We keep this behavior as default for the next three releases to give you more time to adjust your workflows and update.

Configure log rotation for persistent logs

You can now customize log rotation for persistent logs. This helps you keep the right amount of data for troubleshooting or compliance, rotate additional files like audit logs, and schedule rotations to fit your operational windows.

You can configure log rotation in these ways:

- Override the default configuration via the Custom Resource

- Define additional configuration via a ConfigMap or a Secret. In this case, the Operator adds your options to the default configuration
- Set a new rotation schedule

See our documentation for step-by-step instructions for each option.

Configurable timeout for PBM to start backups

You can now configure how long the Operator waits for PBM to report that a backup has begun. Instead of relying on a fixed timeout of 120 seconds, you can tune this value to match your cluster's performance characteristics.

This timeout is controlled by the `startingDeadlineSeconds` option in the Custom Resource.

With this improvement, you:

- Eliminate false backup failures when PBM starts slowly under load
- Give PBM enough time to start in environments with insufficient CPU or memory
- Get more predictable backup flows where PBM and the Operator stay in sync
- Enable reliable backups in large or memory-intensive clusters. Increasing both memory resources and the PBM startup timeout ensures PBM has enough resources and time to initialize.

Support of the `minio` storage type for backups to S3-compatible storages

The Operator now adds the `minio` storage type, aligning with recent PBM changes. The use of the `minio` storage type instructs PBM to use the native MinIO SDK to interact with MinIO and other S3-compatible storage services. This addition helps avoid connectivity and compatibility issues when S3-compatible services don't support Signature Version 4 (SigV4) used in AWS SDK v2.

If your S3-compatible storage shows connectivity issues or depends on older signing mechanisms, consider switching to the `minio` storage type.

Adjust your Custom Resource configuration as follows:

```
minio:
  type: minio
  minio:
    bucket: MINIO-BACKUP-BUCKET-NAME-HERE
    region: us-east-1
    credentialsSecret: my-cluster-name-backup-minio
    endpointUrl: minio.psmdb.svc.cluster.local:9000/minio/
    insecureSkipTLSVerify: false
    prefix: ""
    secure: false
```

The use of the `minio` storage type offers a more stable and straightforward configuration for custom S3 storage services. It also keeps your Operator setup aligned with PBM's current behavior.

Use Private Certificate Authorities (CA) for TLS communication with S3-compatible storage services

You can now provide a Private Certificate Authority (CA) bundle to verify TLS communication with an S3-compatible storage service. This is essential for organizations utilizing internal PKI to secure backup and restore traffic.

The configuration is straightforward: create the Secret that stores your private CA and certificates to authorize in the S3-compatible storage. Then reference this Secret and specify the CA certificate in the `caBundle` option in the Custom Resource. The Operator will verify TLS communication against it.

Note that you must use the `minio` storage type for your S3-compatible storage services. Read more about this storage type in the Native MinIO support as a backup storage section.

Here's the example configuration:

```
backup:
  storages:
    minio:
      type: minio
      minio:
        caBundle:
           name: minio-ca-bundle
           key: ca.crt
```

You can specify several storages and use private CA certificates with each. In this case, the Operator merges the certificates into a single file, which is used for secure communication with the respective storage.

With this improvement, you ensure the following:

- Security without compromise – no more bypassing identity checks.
- Alignment with your internal standards – use the CA your company already trusts.
- Confidence in backup and restore flows – every S3 interaction is properly verified.

Read more about the use of private CA certificates in our documentation.

Hook Script support

A new `hookScript` option lets you run a custom shell script before `mongod`, `mongos`, or a `pbm-agent` starts, without maintaining custom images or modifying the Operator code. You can define the script inline or provide it via a ConfigMap. The script runs inside the container and can block startup if it exits with a non-zero status. See the documentation for setup details.

`hookScript` is supported across all major components: replica set members, hidden, arbiter, and non-voting nodes, the config server replica set, `mongos` Pods, and `pbm-agent` containers.

This feature is intended for environments that require additional startup logic. An example of such logic can be retrieving external secrets, enforcing security checks, adjusting configuration files, or registering nodes in external systems. It lets you adapt startup behavior while keeping the deployment fully managed by the Operator.

Automatic storage resizing

Starting with version 1.22.0, the Operator can automatically resize Persistent Volume Claims (PVCs) for replica sets and config server 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 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.

Cluster readiness now reflects PBM state

The `PerconaServerMongoDB` Custom Resource status now includes a new `PBMReady` condition. It provides a clear, stable signal that PBM has finished processing all configured storages, so automation tools can reliably trigger backups and other operations.

To check the status, use either the `kubectl describe psmdb <cluster-name>` or `kubectl get psmdb <cluster-name> -o yaml` command.

During initialization or configuration changes, the `PBMReady` condition is `false` and you clearly see the reason.

```
- type: PBMReady
  status: "False"
  reason: PBMConfigurationIsChanged
  lastTransitionTime: "2025-12-24T09:06:20Z"
```

Once PBM is fully configured and any required resync is complete, the condition changes to `true`:

```yaml
- type: PBMReady
  status: "True"
  reason: PBMConfigurationIsUpToDate
  lastTransitionTime: "2025-12-24T09:06:40Z"
```

Vault integration for system user password management

You can now integrate the Operator with HashiCorp Vault for system user password management. This allows organizations to centralize password management while keeping the Operator responsible for applying those passwords to the database.

When this integration is enabled, the Operator authenticates to Vault using either the Kubernetes authentication method or a Vault token. It retrieves system user passwords during cluster creation and generates the corresponding Kubernetes Secret from this data. The Operator periodically checks Vault for password changes and updates the Secret when differences are detected. If Vault is temporarily unavailable or the Operator cannot retrieve the passwords, it logs the event and continues cluster reconciliation to ensure the cluster availability.

Organizations benefit from this integration when they need:

- Centralized credential governance
- Auditable password rotation
- Compliance with internal security policies
- Separation of duties (DBA vs. security team)
- Consistent password lifecycle management across environments

Vault becomes the single source of truth, while the Operator ensures Percona Server for MongoDB always uses the correct credentials.

Learn more about the workflow and the setup in our documentation.

Deprecation, rename and removal

Deprecated support for Percona Server for MongoDB 6.0

The Operator deprecates the support of Percona Server for MongoDB 6.0 as this major version entered end-of-life stage. You can still run Percona Server for MongoDB 6.0 in the Operator and existing functionality remains compatible. However, we will no longer test new features and improvements against this version.

Percona Server for MongoDB 6.0 will be removed from the Operator in version 1.23.0.

Deprecated support for PMM2

The Operator deprecates support for PMM2 as this version entered end-of-life stage. PMM2 remains available so you can still monitor the health of your database using this version. However, we encourage you to plan migration to PMM3 to enjoy all features and fixes that this version provides. See the PMM upgrade documentation for steps.

The support for PMM2 will be dropped in the Operator in two releases.

Deprecated options

These options are deprecated and will be removed in version 1.25.0:

- `spec.enableVolumeExpansion`. This option remains working for backward compatibility but it will be removed in version 1.25.0. Use the `enableVolumeScaling` option under the `spec.storageScaling` subsection instead.
- `spec.enableExternalVolumeAutoscaling`. This option remains working for backward compatibility but it will be removed in version 1.25.0. Use the `enableExternalAutoscaling` option under the `spec.storageScaling` subsection instead.

After the upgrade, reconfigure your Custom Resource to use the new structure.

CRD Changes

- The `.spec.startingDeadlineSeconds` option has now a minimum value of `1` and the default value of `120`

Changelog

New Features

- K8SPSMDB-1418- Added the ability to load custom SSL certificates for backup operations to S3 storage. This enables secure communication with S3-compatible storage using the certificates approved and trusted by your company
- K8SPSMDB-1419 - Added the ability to restore data to clusters with different replica set names than the original source via the replica set remapping support
- K8SPSMDB-1472 - Added the ability to automatically resize PVC based on user-defined thresholds
- K8SPSMDB-1503 - Improved Split Horizon support by automatically including specified horizon domains in certificates generated by the Operator, eliminating manual TLS configuration steps.
- K8SPSMDB-1531 - Added the ability to define and execute custom scripts at specific lifecycle events of a container via the `hookScript` support.
- K8SPSMDB-1548 Added the ability to specify custom environment variables for containers for cluster components via the Custom Resource. This provides more granular control over container configurations.

Improvements

- K8SPSMDB-1062 - Added the `appProtocol` support to Service objects, improving compatibility with service meshes and ingress controllers that require explicit protocol definitions.
- K8SPSMDB-1410 - Enhanced backup status visibility by adding a specific condition to indicate when PBM initialization and storage resync are complete. This enables external tools to safely trigger backups
- K8SPSMDB-1445 - Updated base images to RHEL 10, ensuring the Operator benefits from the latest OS-level security patches and performance improvements.
- K8SPSMDB-1448 - Integrated HashiCorp Vault for system user credentials management, enabling you to store and manage them in a centralized place. The Operator securely fetches and manages database passwords from a central Vault instance during cluster creation.
- K8SPSMDB-1451 Fix replica set init checking
- K8SPSMDB-1470 - Automated Custom Resource Definitions (CRDs) updates during chart upgrades by creating a dedicated CRD Helm chart and adding it as a dependency to the main chart.
- K8SPSMDB-1518 Added the ability to customize logrotate configuration. This gives users the flexibility to define their own rotation rules and retention policies.
- K8SPSMDB-1520 - Added native MinIO support for backup storage, utilizing the MinIO Go client to improve compatibility with S3-compatible services that do not support SigV4 used in AWS SDK 2
- K8SPSMDB-1532 - Added support for revisionHistoryLimit in Helm charts, allowing users to control how many old ReplicaSets are retained by Kubernetes for easier rollbacks and cleaner environments.
- K8SPSMDB-1539 - Improved the Helm chart by adding a `namespaceOverride` field to the `values.yaml` file. This improvement allows users to explicitly configure the deployment namespace within their values file, providing greater flexibility for managing resources in environments where the target namespace differs from the default Helm installation namespace.

Bugs Fixed

- K8SPSMDB-1208 - Fixed a deletion deadlock where clusters stuck in the “initializing” state (due to unschedulable pods) could not be removed from the namespace
- K8SPSMDB-1296 - Fixed a bug where MongoDB pods were incorrectly marked as “Ready” while still in the initial sync (STARTUP2) state. The health check logic now correctly identifies the STARTUP2 state as unready, ensuring that pods only receive traffic once they are fully synchronized with the replica set.
- K8SPSMDB-1359 - Added the ability to disable authentication for non-production or testing environments by using a Custom Resource. (Thank you Abdullah Alaqeel for reporting this issue)
- K8SPSMDB-1387 - Resolved a startup failure that occurred when cert-manager was configured with the –enable-certificate-owner-ref option, which previously blocked MongoDB clusters from starting
- K8SPSMDB-1390 - Reduced log noise by stopping the Operator from incorrectly reporting and attempting to delete “outdated backup jobs” for the wrong clusters in shared namespaces.
- K8SPSMDB-1429 - Fixed a race condition during restores in sharded clusters, where a restore task would fail if triggered immediately after all StatefulSets reported `ready` but before the cluster state was fully stabilized.
- K8SPSMDB-1431 - Improved error reporting for backup configuration, ensuring the cluster enters an “Error” state if a referenced backup storage secret is missing, rather than reporting a misleading “Ready” status.

- K8SPSMDB-1466 - Fixed an Operator crash caused by stale discovery information in environments using Kubernetes Multi-Cluster Services (MCS), improving stability for multi-cluster deployments.
- K8SPSMDB-1477 - Fixed a bug where scheduled backups were not deleted due to a change in the ancestor label format. The Operator now correctly recognizes both old and new labels, ensuring that orphaned backup resources are properly cleaned up to prevent unnecessary storage consumption.
- K8SPSMDB-1488 - Ensured PBM configuration updates when credentials in the storage secret are modified, correctly triggering a resync for MinIO and other S3-compatible storage types.
- K8SPSMDB-1493 - Fixed a synchronization deadlock between backups and SmartUpdate operations, where a backup in the “requested” state could indefinitely block a cluster update.
- K8SPSMDB-1524 - Fixed a bug where missing the `region` parameter in S3 storage configurations caused the pbm-agent to enter an infinite resynchronization loop. The Operator now makes the `region` field mandatory for S3-compatible storages and uses a default values for those that don’t require the region to ensure stable configuration detection and prevent backup delays.
- K8SPSMDB-1527 - Added the `PBMReady` condition to the cluster status to ensure the Operator only marks a cluster as ready once Percona Backup for MongoDB is fully initialized and operational.
- K8SPSMDB-1541 - Restored support for Primary-Secondary-Arbiter (PSA) architectures, resolving an issue where arbiter nodes were not correctly handled by the Operator logic.
- K8SPSMDB-1542 - Added a configurable PBM starting deadline, allowing users to adjust the timeout for backups in the “starting” phase to suit their network and storage performance.
- K8SPSMDB-1552 - Fixed the issue with the Operator reporting the panic state if Secret retrieval fails
- K8SPSMDB-1553 - Preserved annotations during PVC resizing, ensuring that the `kubectl.kubernetes.io/restartedAt` annotation is not lost when a StatefulSet is recreated to accommodate volume expansion.
- K8SPSMDB-1560 - Fixed a bug where log rotation incorrectly produced duplicate files, resolving issues with unnecessary storage growth. The updated logic ensures only a single log file is generated and changes the default rotation schedule from hourly to daily for more efficient resource management.
- K8SPSMDB-1561 - Mounted the PVC to the PMM3 container. This enables the node_exporter running in the pmm-client container to collect metrics on the database’s persistent volume. (Thank you Stephen Foulkes for contributing to this issue)
- K8SPSMDB-1538 - Fixed a template condition bug in the Helm chart that prevented arbiter affinity from being applied correctly. The logic now correctly checks for `arbiter.affinity` independently of the parent replica set configuration, ensuring that arbiters are scheduled on the intended nodes even when general replica set affinity is not defined.

Supported software

The Operator was developed and tested with the following software:

- Percona Server for MongoDB 6.0.27-21, 7.0.30-16, and 8.0.19-7
- Percona Backup for MongoDB 2.12.0
- PMM Client: 2.44.1-1
- PMM3 Client: 3.6.0
- cert-manager: 1.19.3
- LogCollector based on fluent-bit: 4.0.1-2

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:

- Google Kubernetes Engine (GKE) 1.32 - 1.33
- Amazon Elastic Kubernetes Service (EKS) 1.32 - 1.35
- Azure Kubernetes Service (AKS) 1.32 - 1.34
- OpenShift Container Platform 4.17 - 4.21
- Minikube 1.38.0 based on Kubernetes v1.35.0

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 the backward compatibility offered by Kubernetes itself.

Percona certified images

Find Percona's certified Docker images that you can use with the Percona Operator for MongoDB in the following table:

Image	Digest
percona/percona-server-mongodb:8.0.19-7	779378a9f52cd9e617d0c356053b4b9c97209b0b759aa819300427836f6f2c66
percona/percona-server-mongodb:8.0.19-7 (ARM64)	531793543992d2e502d3187b778f8468462dd9cdb16a58fa157fe86626594ede
percona/percona-server-mongodb:7.0.30-16	32bc384e2af367e4261bdf1d2b09211bb210425f292494e5837e5a05f6805768
percona/percona-server-mongodb:7.0.30-16 (ARM64)	d333516c8b912d25b12759ab9c6d27d1ae4567e940f7922bd05c8dc06ae3228b
percona/percona-server-mongodb:6.0.27-21	2bd82853fd207ecadc8e5cd3ee598db7b9943f6976d9ae2140f5bdc997b0e8bb
percona/percona-server-mongodb:6.0.27-21 (ARM64)	a23e4fa1b956299fb7403eef70082b13b3ecf75e51bba12f674f552508a42d61
percona/fluentbit:4.0.1-2	d732be16b7f3d0eea8d9c0a50f72136db83c2aed74262300d97abd2cfaefbc26
percona/fluentbit:4.0.1-2 (ARM64)	805c102d3646b8499b74ea9864d4c29137f94e4603470c23da1c032276193a36
percona/pmm-client:3.6.0	174fa4675d3ea4d95fd7b45d11f2bcc98b98b703662e6b2614dfe886a7187b23
percona/pmm-client:3.6.0 (ARM64)	435a9af2083adb68ddab6a97e6d02bd6d31c54562e919ebc09618e886d58d1ae
percona/pmm-client:2.44.1-1	52a8fb5e8f912eef1ff8a117ea323c401e278908ce29928dafc23fac1db4f1e3
percona/pmm-client:2.44.1-1 (ARM64)	390bfd12f981e8b3890550c4927a3ece071377065e001894458047602c744e3b
percona/percona-backup-mongodb:2.12.0	6d56b787c2dc532c9c19cc607a2bc1855a722cdaa52024ea38d4de4aae777f8a
percona/percona-backup-mongodb:2.12.0 (ARM64)	e2177fa6fef0e47ee8392ca7044a6781150ccf7297383239f813558323aa0a8e
percona/percona-server-mongodb-operator:1.22.0	d4384b3853dc5026cc33de1dfd68ac11aaf5d54f4e84b115e05d7d473f4c6188
percona/percona-server-mongodb-operator:1.22.0 (ARM64)	be97aad4c0642bc4e2e5c7c31b4d123d9f6135070682b8fbd6024c7620315e72

Find previous version images in the documentation archive

Percona Operator for MongoDB 1.21.2 (2026-01-12)

Get started with the Operator →

Release Highlights

Security update: Mitigation of heap memory disclosure vulnerability

This release includes the fix for security vulnerability CVE-2025-14847: CWE-130, which is about how MongoDB uses `zlib` compression library. Attackers with network access to `mongod` or `mongos` can extract fragments of uninitialized server memory without authentication if zlib compression is enabled. This memory may contain sensitive data, which poses a serious information disclosure risk.

The issue is resolved upstream and is included in Percona Server for MongoDB 6.0.27-21, 7.0.28-15 and 8.0.17-6. Percona Operator for MongoDB includes these updated Percona Server for MongoDB images.

We strongly recommend upgrading to this latest version of the Operator to ensure your deployments remain secure.

Learn more about the vulnerability in the upstream bug report and in Percona Blog: Urgent Security Update: Patching "Mongobleed" (CVE-2025-14847) in Percona Server for MongoDB →.

Supported software

The Operator was developed and tested with the following software:

- Percona Server for MongoDB 6.0.27-21, 7.0.28-15, and 8.0.17-6
- Percona Backup for MongoDB 2.11.0
- PMM Client: 2.44.1-1
- PMM3 Client: 3.5.0
- cert-manager: 1.18.2
- LogCollector based on fluent-bit 4.0.1

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 Kubernetes Service (EKS) 1.31 - 1.34
- Azure Kubernetes Service (AKS) 1.31 - 1.33
- OpenShift Container Platform 4.16 - 4.19
- Minikube 1.37.0 based on Kubernetes v1.34.0

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 the backward compatibility offered by Kubernetes itself.

Percona certified images

Find Percona's certified Docker images that you can use with the Percona Operator for MongoDB in the following table:

Image	Digest
percona/percona-server-mongodb:8.0.17-6	ae6380469f6b73d3517ec4eae7b2f12ff6310dc2deae8e52fe514276c45e9440
percona/percona-server-mongodb:8.0.17-6 (ARM64)	f1170f8bf68d051816cd4d956ca1f6ee9885c6cf0e1e5db5dc00a137af3603ee

percona/percona-server-mongodb:7.0.28-15	d131a4375c3e669f97da6cdf5eef847099c731fd956341345f37e6e6fb68d699
percona/percona-server-mongodb:7.0.28-15 (ARM64)	6bc8ee24a7e60ec8ef32002165584320b9cc0eb6067a5f304cee6f1ea708f9b3
percona/percona-server-mongodb:6.0.27-21	2bd82853fd207ecadc8e5cd3ee598db7b9943f6976d9ae2140f5bdc997b0e8bb
percona/percona-server-mongodb:6.0.27-21 (ARM64)	a23e4fa1b956299fb7403eef70082b13b3ecf75e51bba12f674f552508a42d61
percona/percona-backup-mongodb:2.11.0	d09f5de92cfbc5a7a42a8cc86742a07481c98b3b42cffdc6359b3ec1f63de3a5
percona/percona-backup-mongodb:2.11.0 (ARM64)	a60d095439537b982209582d428b3b39a01e31e88b2b62d2dcbd99ea4e2d9928
percona/pmm-client:2.44.1-1	52a8fb5e8f912eef1ff8a117ea323c401e278908ce29928dafc23fac1db4f1e3
percona/pmm-client:2.44.1-1 (ARM64)	390bfd12f981e8b3890550c4927a3ece071377065e001894458047602c744e3b
percona/pmm-client:3.5.0	352aee74f25b3c1c4cd9dff1f378a0c3940b315e551d170c09953bf168531e4a
percona/pmm-client:3.5.0 (ARM64)	cbbb074d51d90a5f2d6f1d98a05024f6de2ffdcb5acab632324cea4349a820bd
percona/fluentbit:4.0.1	a4ab7dd10379ccf74607f6b05225c4996eeff53b628bda94e615781a1f58b779
percona/percona-server-mongodb-operator:1.21.2	76d59626914f4d18eb0c19d8e31d2480f7a358daa3ded777cafb7e3717c7508d
percona/percona-server-mongodb-operator:1.21.2 (ARM64)	b6adecc41de81f69a4faf552aeca31c06411f012378be248ead70a538c8ea365

Find previous version images in the documentation archive

Percona Operator for MongoDB 1.21.1 (2025-10-30)

Get started with the Operator →

Release Highlights

This release resolves the MongoDB connection leak issue that occurred during PBM operations. It also addresses issues with the Operator's retrieval and display of the PBM version.

Changelog

Fixed bugs

- K8SPSMDB-1504 - Fixed connection leaks issue during PBM operations and the Operator crashing with out-of-memory error by properly closing connections after the PBM operation is complete.
- K8SPSMDB-1506 - Fixed an issue with the Operator not being able to get PBM version due to the change of log where latest PBM versions print the version. The Operator now combines both stderr and stdout to correctly retrieve the PBM version.

Supported software

The Operator was developed and tested with the following software:

- Percona Server for MongoDB 6.0.25-20, 7.0.24-13, and 8.0.12-4
- Percona Backup for MongoDB 2.11.0
- PMM Client: 2.44.1
- PMM3 Client: 3.4.1
- cert-manager: 1.18.2
- LogCollector based on fluent-bit 4.0.1

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 Container Platform 4.16 - 4.19
- Azure Kubernetes Service (AKS) 1.31-1.33
- 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 the backward compatibility offered by Kubernetes itself.

Percona certified images

Find Percona's certified Docker images that you can use with the Percona Operator for MongoDB in the following table:

Image	Digest
percona/percona-server-mongodb-operator:1.21.1 (x86_64)	155f6ee71dcfc52ff30ed4e2c4396fc3d3534c83b4794de4d90c79542fbb0e34
percona/percona-server-mongodb-operator:1.21.1 (ARM64)	88926b82a5551c36592d1c83b2e80d3c3560f0809cdb7b5d6648038123b65097

percona/percona-server-mongodb:8.0.12-4	ab8793879409788b5a19f7e332a3700520e8eeaf4b068ec8cc7d1b680f097307
percona/percona-server-mongodb:8.0.12-4 (ARM64)	d367e225b57783bc2ff8451571c7568dc3b240176cf149a01cc3a7b13fb52a78
percona/percona-server-mongodb:8.0.8-3	e4580ca292f07fd7800e139121aea4b2c1dfa6aa34f3657d25a861883fd3de41
percona/percona-server-mongodb:8.0.8-3 (ARM64)	96cfee2102499aba05e63ca7862102c2b1da1cf9f4eea0cbea3793a07c183925
percona/percona-server-mongodb:8.0.4-1-multi	873b201ce3d66d97b1225c26db392c5043a73cc19ee8db6f2dc1b8efd4783bcf
percona/percona-server-mongodb:8.0.4-1-multi (ARM64)	222ccf746ad4ffdfccf41b41edaa0d318d28f663e13c9629f8dad5a5078434e5
percona/percona-server-mongodb:7.0.24-13	71d5389e91014cf6c486c4d28ee2b3f19f16eb421d9d65b36d70b9f712a43eaa
percona/percona-server-mongodb:7.0.24-13 (ARM64)	22012034c3e30029b34dda235aa14642377522ba307d742f64d7f69ed6feccf9
percona/percona-server-mongodb:7.0.18-11	0115a72f5e60d86cb4f4b7eae32118c0910e8c96831e013de12798a1771c4c91
percona/percona-server-mongodb:7.0.18-11 (ARM64)	86c17067f3e233f522612389ed2500231cbb22ce93524c476b9aa8d464d06f0b
percona/percona-server-mongodb:7.0.15-9-multi	7bffdf2e71c121e2ab37b4fa7e2f513237abdd65266da384bf8197cee1316917
percona/percona-server-mongodb:7.0.15-9-multi (ARM64)	fdc4875df82572267445811445ebf517f63e509be54d1a2599fe58e1c525e1d8
percona/percona-server-mongodb:7.0.14-8-multi	ed932d4e7231dcb793bf609f781226a8393aa8958b103339f4a503a8f70ed17e
percona/percona-server-mongodb:7.0.14-8-multi (ARM64)	052f84ee926ad9b5146f08a7e887820342d65b757a284c2f0ea8e937bb51cd7b
percona/percona-server-mongodb:7.0.12-7	7f00e19878bd143119772cd5468f1f0f9857dfcd2ae2f814d52ef3fa7cff6899
percona/percona-server-mongodb:6.0.25-20	0254c10fb8c249c108cd0a6e5885dfe76785e8fdd6ceb23ce98854234672e5d6
percona/percona-server-mongodb:6.0.25-20 (ARM64)	0fd4d1ca4da6377450964f225bd1d508730be9c1fca1c36c3bfcc107678d9a50
percona/percona-server-mongodb:6.0.21-18	579d2fdc617ea42ab2be8c2682955b489dbf49ab19771b7a5d9c77da4dd323e7
percona/percona-server-mongodb:6.0.21-18 (ARM64)	b9d2b7e8c4a97b2d20e2aaccfbd183f65f8ccd9f2ea13939515e18e02bc64871
percona/percona-server-mongodb:6.0.19-16-multi	c8ff08c4b8a96679e2daf4845873fdd4d2c48646b84db19f0c5fe02e8f3808b4
percona/percona-server-mongodb:6.0.19-16-multi (ARM64)	6908b28ced260b762cd38a642c06dd802cbef0a43ab5f22afe7b583b234ebcec
percona/percona-server-mongodb:6.0.18-15-multi	d197ce16ab0eed6df25e632b92dea5ce448e549e02028f39b78f5730c2ffef36
percona/percona-server-mongodb:6.0.18-15-multi (ARM64)	7fd1d8f74f71dea6ad423e8e202a0617bdd1e8783f2b5cb071b5281685ce0adf
percona/percona-server-mongodb:6.0.16-13	1497e58e39497d8425ccd053898dc323338d6eb3f0e3c4c223f9d5a468da7931
percona/fluentbit:4.0.1	a4ab7dd10379ccf74607f6b05225c4996eeff53b628bda94e615781a1f58b779
percona/pmm-client:3.4.1	1c59d7188f8404e0294f4bfb3d2c3600107f808a023668a170a6b8036c56619b
percona/pmm-client:2.44.1-1	52a8fb5e8f912eef1ff8a117ea323c401e278908ce29928dafc23fac1db4f1e3
percona/percona-backup-mongodb:2.11.0	d09f5de92cfbc5a7a42a8cc86742a07481c98b3b42cffdc6359b3ec1f63de3a5
percona/percona-backup-mongodb:2.11.0 (ARM64)	a60d095439537b982209582d428b3b39a01e31e88b2b62d2dcbd99ea4e2d9928

Find previous version images in the [documentation archive](#)

Percona Operator for MongoDB 1.21.0 (2025-10-20)

Get started with the Operator →

Release Highlights

Warning

We strongly recommend that you **do not upgrade to this version** due to a MongoDB connection leak issue that occurs during PBM operations.

For more details, see K8SPSMDB-1504.

The issue is resolved and included in the Operator version 1.21.1. Please plan your upgrades to this version.

This release of Percona Operator for MongoDB includes the following new features and improvements:

Percona Server for MongoDB 8.0 is now the default version

For you to enjoy all features and improvements that come with the latest major version out of the box, the Operator now deploys the cluster with Percona Server for MongoDB 8.0 by default. You can always change the version to your desired one for the installation and update. Check the list of Percona certified images for the database versions available for this release. For previous Operator versions, learn how to query the Version Service and retrieve the available images from it.

PMM3 support

The Operator is natively integrated with PMM 3, enabling you to monitor the health and performance of your Percona Distribution for MongoDB deployment and at the same time enjoy enhanced performance, new features, and improved security that PMM 3 provides.

Note that the Operator supports both PMM2 and PMM3. The decision on what PMM version is used depends on the authentication method you provide in the Operator configuration: PMM2 uses API keys while PMM3 uses service account tokens. If the Operator configuration contains both authentication methods with non-empty values, PMM3 takes the priority.

To use PMM, ensure that the PMM client image is compatible with the PMM Server version. Check Percona certified images for the correct client image.

For how to configure monitoring with PMM see the documentation.

Hidden nodes support

In addition to arbiters and non-voting nodes, you can now deploy hidden nodes in your Percona Server for MongoDB cluster. These nodes hold a full copy of the data but remain invisible to client applications. They are good for tasks like backups and reporting, since they access the data without affecting normal traffic.

Hidden nodes are added as voting members and can participate in primary elections. Therefore, the Operator enforces rules to ensure the number of voting members is odd and doesn't exceed seven, which is the maximum allowed number of voting members:

- If the total number of voting members is even, the Operator converts one node to non-voting to maintain an odd number of voters. The node to convert is typically the last Pod in the list.
- If the number of voting members is odd and not more than 7, all nodes participate in elections.
- If the number of voting members exceeds 7, the Operator automatically converts some nodes to non-voting to stay within MongoDB's limit.

To inspect the current configuration, connect to the cluster with the `clusterAdmin` privileges and run the `rs.config().members` command.

Support for Google Cloud Client library in PBM

The Operator comes with the latest PBM version 2.11.0, which includes the support of Google Cloud Client library and authentication with service account keys.

To use Google Cloud Storage for backups with service account keys, you need to do the following:

1. Create a service account key
2. Create a Secrets object with this key
3. Configure the storage in the Custom Resource

See the Configure Google Cloud Storage documentation for detailed steps.

The configuration of Google Cloud Storage with HMAC keys remains unchanged. However, PBM has a known issue for using HMAC keys with GCS, which was reported in [PBM-1605](). The issue is in uploading large files (~512MB+) to the storage when the network is unstable. Such backups may be corrupted or incomplete but they are incorrectly treated as valid backups and pose a risk of restore failures. Therefore, we recommend migrating to the native GCS connection type with service account (JSON) keys after the upgrade.

Improve operational resilience and observability with persistent cluster-level logging for MongoDB Pods

Debugging distributed systems just got easier. The Percona Operator for MongoDB now supports persistent logging, ensuring that logs from your `mongod` instances are stored persistently, even across Pod restarts.

Cluster-level logging is done with Fluent Bit, running as a sidecar container within each database Pods.

Currently, logs are collected only for the `mongod` instances. All other logs are ephemeral, meaning they will not persist after a Pod restart. Logs are stored for 7 days and are rotated afterwards.

Learn more about persistent logging in the documentation

Improved backup retention for streamlined management of scheduled backups in cloud storage

A new backup retention configuration gives you more control over how backups are managed in storage and retained in Kubernetes.

With the `deleteFromStorage` flag, you can disable automatic deletion from AWS S3 or Azure Blob storage and instead rely on native cloud lifecycle policies. This makes backup cleanup more efficient and better aligned with flexible storage strategies.

The legacy `keep` option is now deprecated and mapped to the new `retention` block for compatibility. We encourage you to start using the `backup.tasks.retention` configuration:

```
spec:
  backup:
    tasks:
      - name: daily-s3-us-west
        enabled: true
        schedule: "0 0 ** *"
        retention:
          count: 3
          type: count
          deleteFromStorage: true
        storageName: s3-us-west
        compressionType: gzip
        compressionLevel: 6
```

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 as the environment variable for the Operator deployment.

This enhancement significantly improves scalability and responsiveness, especially in multi-cluster environments.

Added labels to identify the version of the Operator

Custom Resource Definition (CRD) is compatible with the last three Operator versions. To know which Operator version is attached to it, we've added labels 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 perconaservermongodbs.psmdb.percona.com --show-labels
```

View backup size

You can now see the size of each backup when viewing the backup list either via the command line or from Everest or other apps integrated with the Operator. This improvement makes it easier to monitor storage usage and manage your backups efficiently.

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.enableExternalVolumeAutoscaling` option to `true` in the Custom Resource manifest.

Deprecation, rename and removal

- The `backup.schedule.keep` field is deprecated and will be removed in future releases. We recommend using the `backup.schedule.retention` instead as follows:

```
yaml
  schedule:
    - name: "sat-night-backup"
      schedule: "0 0 ** 6"
      retention:
        count: 3
        type: count
        deleteFromStorage: true
      storageName: s3-us-west
```

- The S3-compatible implementation of Google Cloud Storage (GCS) with using HMAC keys is deprecated in the Operator. We encourage you to switch to using to the native GCS connection type with service account (JSON) keys after the upgrade.

Changelog

New features

- K8SPSMDB-297: Added persistent logging with the Fluent Bit log collector
- K8SPSMDB-1268 - Added support for PMM v3.
- K8SPSMDB-723 - Added the ability to add hidden members to MongoDB replica sets for specialized purposes.

Improvements

- K8SPSMDB-1072 - Added the ability to configure retention policy for scheduled backups
- K8SPSMDB-1216 - Updated the command to describe the `mongod` instance role to `db.hello()`, which is the currently used one.
- K8SPSMDB-1243 - Added the ability to pass PBM restore configuration options to the Operator.
- K8SPSMDB-1261 - Improved the test suite for physical backups to run on every supported platform individually.
- K8SPSMDB-1262 - Improved the test suite for on demand backups to run on OpenShift
- K8SPSMDB-1272 - The `helm upgrade` command now displays warnings to clarify when CRDs are not updated.
- K8SPSMDB-1284 - Clearer error messages are now displayed if a filesystem backup deletion fails.
- K8SPSMDB-1285 - CRDs now include labels that make it easy to identify their associated Operator version.
- K8SPSMDB-1304 - Added labels recommended by Kubernetes to the Operator deployment object
- K8SPSMDB-1318 - Added the ability to configure concurrent reconciles to speed up cluster reconciliation in setups where the Operator manages several database clusters.
- K8SPSMDB-1319 - Scheduled database backups now wait for the database to be healthy before starting, preventing unnecessary failures.
- k8spsmdb-1339 - Added validation for the selected restore time, preventing the point-in-time restore process from starting with an invalid date or time.
- K8SPSMDB-1344, K8SPSMDB-871 - Added the ability to retrieve and store the backup size
- K8SPSMDB-1398 - Added the ability to configure the use of an external autoscaler (Thank you Terry for contribution)
- K8SPSMDB-1412 - Added the support for Google Cloud Storage with authentication via service account keys.

Fixed bugs

- K8SPSMDB-1154 - MongoDB clusters using the `inMemory` storage engine now deploy correctly (Thank you user KOS for reporting this issue).
- K8SPSMDB-1292 - Fixed the issue with physical restores failing when TLS configuration is defined by using it to construct the correct MongoDB connection string URL.
- K8SPSMDB-1297 - Exposed the data directory for the `pmm-client` sidecar container to enable it to gather required metrics.
- K8SPSMDB-1308 - Improved PBM restore logging to store logs for the latest restore in the `/data/db/pbm-restore-logs`.
- K8SPSMDB-1336 - Logical backups can now be restored to a new cluster without encountering `Time monotonicity violation` errors or service restarts.
- K8SPSMDB-1371 - Physical point-in-time recovery using the `latest` type no longer crashes but gracefully fails the restore process when oplog data is unavailable.
- K8SPSMDB-1400 - Resolved an issue that caused physical restores to fail on AKS and EKS environments.
- K8SPSMDB-1425 - Restoring a MongoDB cluster with point-in-time recovery now succeeds even when source and target storage prefixes differ.
- K8SPSMDB-1480 - Fixed an issue that caused cluster errors when scaling replica sets resulted in an invalid number of voting members.

Documentation improvements

- The multi-cluster and multi-region deployment section has been improved and expanded with the information about multi-cluster deployment and its value as well as how it works. It provides improved guidance on multi-cluster services, a step-by-step tutorial for enabling multi-cluster deployments on GKE, and revised instructions for deploying and interconnecting sites for replication. The docs also walk you through planned switchover and controlled failover procedures in disaster scenarios.
- Updated the Scale Percona Server for MongoDB on Kubernetes topic with the information about the `pvc-resize-in-progress` annotation and how it works.
- Updated the Configure backup storage with the Google Cloud Storage configuration.
- Configuration for config server split horizons is now accurately documented, simplifying multi-cluster deployments and external DNS integration.
- The Data-at-rest encryption topic is updated with the correct steps for using HashiCorp Vault.
- New documentation is available detailing important considerations for upgrading your Kubernetes cluster before updating any Operator.

Supported software

The Operator was developed and tested with the following software:

- Percona Server for MongoDB 6.0.25-20, 7.0.24-13, and 8.0.12-4.
- Percona Backup for MongoDB 2.11.0.
- PMM Client: 3.4.1
- LogCollector based on fluent-bit 4.0.1

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 Container Platform 4.16 - 4.19
- Azure Kubernetes Service (AKS) 1.31-1.33
- 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 the backward compatibility offered by Kubernetes itself.

Percona certified images

Find Percona’s certified Docker images that you can use with the Percona Operator for MongoDB in the following table:

Image	Digest
percona/percona-server-mongodb-operator:1.21.0	791a27c0df745e1b3531b6bbdba0b4ff67c46a38df62c23bc3203bcf0563e4cb
percona/percona-server-mongodb-operator:1.21.0 (ARM64)	56bec3f64f64497bc1468ec64dc5ca44a282da4c2666e1a9d0f96a00d329f88f
percona/percona-server-mongodb:8.0.12-4 (x86_64)	ab8793879409788b5a19f7e332a3700520e8eeaf4b068ec8cc7d1b680f097307
percona/percona-server-mongodb:8.0.12-4 (ARM64)	d367e225b57783bc2ff8451571c7568dc3b240176cf149a01cc3a7b13fb52a78
percona/percona-server-mongodb:8.0.8-3 (x86_64)	e4580ca292f07fd7800e139121aea4b2c1dfa6aa34f3657d25a861883fd3de41
percona/percona-server-mongodb:8.0.8-3 (ARM64)	96cfee2102499aba05e63ca7862102c2b1da1cf9f4eea0cbea3793a07c183925
percona/percona-server-mongodb:8.0.4-1-multi (x86_64)	873b201ce3d66d97b1225c26db392c5043a73cc19ee8db6f2dc1b8efd4783bcf
percona/percona-server-mongodb:8.0.4-1-multi (ARM64)	222ccf746ad4ffdfccf41b41edaa0d318d28f663e13c9629f8dad5a5078434e5
percona/percona-server-mongodb:7.0.24-13 (x86_64)	71d5389e91014cf6c486c4d28ee2b3f19f16eb421d9d65b36d70b9f712a43eaa
percona/percona-server-mongodb:7.0.24-13 (ARM64)	22012034c3e30029b34dda235aa14642377522ba307d742f64d7f69ed6feccf9
percona/percona-server-mongodb:7.0.18-11 (x86_64)	0115a72f5e60d86cb4f4b7eae32118c0910e8c96831e013de12798a1771c4c91
percona/percona-server-mongodb:7.0.18-11 (ARM64)	86c17067f3e233f522612389ed2500231cbb22ce93524c476b9aa8d464d06f0b
percona/percona-server-mongodb:7.0.15-9-multi (x86_64)	7bffdf2e71c121e2ab37b4fa7e2f513237abdd65266da384bf8197cee1316917
percona/percona-server-mongodb:7.0.15-9-multi (ARM64)	fdc4875df82572267445811445ebf517f63e509be54d1a2599fe58e1c525e1d8
percona/percona-server-mongodb:7.0.14-8-multi (x86_64)	ed932d4e7231dcb793bf609f781226a8393aa8958b103339f4a503a8f70ed17e
percona/percona-server-mongodb:7.0.14-8-multi (ARM64)	052f84ee926ad9b5146f08a7e887820342d65b757a284c2f0ea8e937bb51cd7b
percona/percona-server-mongodb:7.0.12-7 (x86_64)	7f00e19878bd143119772cd5468f1f0f9857dfcd2ae2f814d52ef3fa7cff6899
percona/percona-server-mongodb:6.0.25-20 (x86_64)	0254c10fb8c249c108cd0a6e5885dfe76785e8fdd6ceb23ce98854234672e5d6
percona/percona-server-mongodb:6.0.25-20 (ARM64)	0fd4d1ca4da6377450964f225bd1d508730be9c1fca1c36c3bfcc107678d9a50
percona/percona-server-mongodb:6.0.21-18 (x86_64)	579d2fdc617ea42ab2be8c2682955b489dbf49ab19771b7a5d9c77da4dd323e7
percona/percona-server-mongodb:6.0.21-18 (ARM64)	b9d2b7e8c4a97b2d20e2aaccfbd183f65f8ccd9f2ea13939515e18e02bc64871
percona/percona-server-mongodb:6.0.19-16-multi (x86_64)	c8ff08c4b8a96679e2daf4845873fdd4d2c48646b84db19f0c5fe02e8f3808b4
percona/percona-server-mongodb:6.0.19-16-multi (ARM64)	6908b28ced260b762cd38a642c06dd802cbef0a43ab5f22afe7b583b234ebcec
percona/percona-server-mongodb:6.0.18-15-multi (x86_64)	d197ce16ab0eed6df25e632b92dea5ce448e549e02028f39b78f5730c2ffef36
percona/percona-server-mongodb:6.0.18-15-multi (ARM64)	7fd1d8f74f71dea6ad423e8e202a0617bdd1e8783f2b5cb071b5281685ce0adf
percona/percona-server-mongodb:6.0.16-13	1497e58e39497d8425ccd053898dc323338d6eb3f0e3c4c223f9d5a468da7931
percona/pmm-client:3.4.1	1c59d7188f8404e0294f4bfb3d2c3600107f808a023668a170a6b8036c56619b
percona/pmm-client:2.44.1-1	52a8fb5e8f912eef1ff8a117ea323c401e278908ce29928dafc23fac1db4f1e3
percona/fluentbit:4.0.1	a4ab7dd10379ccf74607f6b05225c4996eeff53b628bda94e615781a1f58b779
percona/percona-backup-mongodb:2.11.0	d09f5de92cfbc5a7a42a8cc86742a07481c98b3b42cffdc6359b3ec1f63de3a5

percona/percona-backup-mongodb:2.11.0 (ARM64)	a60d095439537b982209582d428b3b39a01e31e88b2b62d2dcbd99ea4e2d9928

Find previous version images in the documentation archive

Percona Operator for MongoDB 1.20.1 (2025-06-04)

Get started with the Operator →

Release Highlights

This release of Percona Operator for MongoDB fixes the failing backup that was caused by the Operator sending multiple requests to PBM. The issue was fixed by bypassing the cache for the backup controller and enabling direct communication with the API server for sending backup requests.

Changelog

Bugs Fixed

- K8SPSMDB-1395 - Fixed the issue with failing backups due to the Operator sending multiple backup requests based on the stale status data

Supported software

The Operator was developed and tested with the following software:

- Percona Server for MongoDB 8.0.8-3, 7.0.18-11, and 6.0.21-18
- Percona Backup for MongoDB 2.9.1
- PMM Client 2.44.1
- cert-manager 1.17.2

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:

- Google Kubernetes Engine (GKE) 1.30 - 1.32
- Amazon Elastic Container Service for Kubernetes (EKS) 1.30 - 1.32
- OpenShift Container Platform 4.14 - 4.18
- Azure Kubernetes Service (AKS) 1.30 - 1.32
- Minikube 1.35.0 with Kubernetes 1.32.0

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 the backward compatibility offered by Kubernetes itself.

Percona certified images

Find Percona's certified Docker images that you can use with the Percona Operator for MongoDB in the following table:

Images released with the Operator version 1.20.1:

Image	Digest
percona/percona-server-mongodb-operator:1.20.1	b559cdd349916d806f6b13b4ac43fdbae982298fad2088b649631a356020ee46
percona/percona-server-mongodb-operator:1.20.1 (ARM64)	5a66e497dd1650e5a1123659292fe4c615e0ab5ce7e5d8437bf2101f91b625e1
percona/pmm-client:2.44.1	8b2eaddffd626f02a2d5318ffebc0c277fe8457da6083b8cfcada9b6e6168616
percona/pmm-client:2.44.1 (ARM64)	337fecd4afdb3f6daf2caa2b341b9fe41d0418a0e4ec76980c7f29be9d08b5ea
percona/percona-backup-mongodb:2.9.1 (x86_64)	976bfbaa548eb70dd90bf0bd2dcfe40b2994d749ef644af3a0590f4856e4d7e2

percona/percona-backup-mongodb:2.9.1 (ARM64)	ebc6e5c5aa3ed97991d3fd90e9201597b485ddc0eae8d7ee4311ecb785c03bf0
percona/percona-server-mongodb:8.0.8-3 (x86_64)	e4580ca292f07fd7800e139121aea4b2c1dfa6aa34f3657d25a861883fd3de41
percona/percona-server-mongodb:8.0.8-3 (ARM64)	96cfee2102499aba05e63ca7862102c2b1da1cf9f4eea0cbea3793a07c183925
percona/percona-server-mongodb:8.0.4-1-multi (x86_64)	873b201ce3d66d97b1225c26db392c5043a73cc19ee8db6f2dc1b8efd4783bcf
percona/percona-server-mongodb:8.0.4-1-multi (ARM64)	222ccf746ad4ffdfccf41b41edaa0d318d28f663e13c9629f8dad5a5078434e5
percona/percona-server-mongodb:7.0.18-11 (x86_64)	0115a72f5e60d86cb4f4b7eae32118c0910e8c96831e013de12798a1771c4c91
percona/percona-server-mongodb:7.0.18-11 (ARM64)	86c17067f3e233f522612389ed2500231cbb22ce93524c476b9aa8d464d06f0b
percona/percona-server-mongodb:7.0.15-9-multi (x86_64)	7bffdf2e71c121e2ab37b4fa7e2f513237abdd65266da384bf8197cee1316917
percona/percona-server-mongodb:7.0.15-9-multi (ARM64)	fdc4875df82572267445811445ebf517f63e509be54d1a2599fe58e1c525e1d8
percona/percona-server-mongodb:7.0.14-8-multi (x86_64)	ed932d4e7231dcb793bf609f781226a8393aa8958b103339f4a503a8f70ed17e
percona/percona-server-mongodb:7.0.14-8-multi (ARM64)	052f84ee926ad9b5146f08a7e887820342d65b757a284c2f0ea8e937bb51cd7b
percona/percona-server-mongodb:7.0.12-7	7f00e19878bd143119772cd5468f1f0f9857dfcd2ae2f814d52ef3fa7cff6899
percona/percona-server-mongodb:6.0.21-18 (x86_64)	579d2fdc617ea42ab2be8c2682955b489dbf49ab19771b7a5d9c77da4dd323e7
percona/percona-server-mongodb:6.0.21-18 (ARM64)	b9d2b7e8c4a97b2d20e2aaccfbd183f65f8ccd9f2ea13939515e18e02bc64871
percona/percona-server-mongodb:6.0.19-16-multi (x86_64)	c8ff08c4b8a96679e2daf4845873fdd4d2c48646b84db19f0c5fe02e8f3808b4
percona/percona-server-mongodb:6.0.19-16-multi (ARM64)	6908b28ced260b762cd38a642c06dd802cbef0a43ab5f22afe7b583b234ebcec
percona/percona-server-mongodb:6.0.18-15-multi (x86_64)	d197ce16ab0eed6df25e632b92dea5ce448e549e02028f39b78f5730c2ffef36
percona/percona-server-mongodb:6.0.18-15-multi (ARM64)	7fd1d8f74f71dea6ad423e8e202a0617bdd1e8783f2b5cb071b5281685ce0adf
percona/percona-server-mongodb:6.0.16-13	1497e58e39497d8425ccd053898dc323338d6eb3f0e3c4c223f9d5a468da7931
percona/percona-server-mongodb:6.0.15-12	f12dd271d78cf3e70088fea0c420e8c03703457d8a5959b645053546bff94dea

Find previous version images in the documentation archive

Percona Operator for MongoDB 1.20.0 (2025-05-19)

Get started with the Operator →

Release Highlights

This release of Percona Operator for MongoDB includes the following new features and improvements:

Point-in-time recovery from any backup storage

The Operator now natively supports multiple backup storages inheriting this feature from Percona Backup for MongoDB (PBM). This enables you to make a point-in-time recovery from any backup stored on any storage - PBM and the Operator maintain the data consistency for you. And you no longer have to wait till the Operator reconfigures a cluster after you select a different storage for a backup or a restore. As a result, overall performance of your backup flow improves.

Improve RTO with the added support of incremental physical backups (tech preview)

Using incremental physical backups in the Operator, you can now back up only the changes happened since the previous backup. Since increments are smaller in size than the whole backup, the backup completion is faster and you also save on the storage and data transfer costs. Using incremental backups and point-in-time recovery improves your recovery time objective (RTO).

You do need the base backup to start the incremental backup chain and you must make the whole chain from the same storage. Also note that the `percona.com/delete-backup` finalizer and the `.spec.backup.tasks.[].keep` option apply for the incremental base backup but are ignored for subsequent incremental backups.

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: mongo-cluster
```

Changelog

New Features

- K8SPSMDB-1237 - Added support for incremental physical backups
- K8SPSMDB-1329 - Allowed setting loadBalancerClass service type and using a custom implementation of a load balancer rather than the cloud provider default one.

Improvements

- K8SPSMDB-621 - Set `PBM_MONGODB_URI` env variable in PBM container to avoid defining it for every shell session and improve setup automation (Thank you Damiano Albani for reporting this issue)
- K8SPSMDB-1219 - Improved the support of multiple storages for backups by using the Multi Storage support functionality in PBM. This enables users to make point-in-time recovery from any storage
- K8SPSMDB-1223 - Improved the `MONGODB_PBM_URI` connection string construction by enabling every `pbm-agent` to connect to local mongoDB directly
- K8SPSMDB-1226 - Documented how to pass custom configuration for PBM
- K8SPSMDB-1234 - Added the ability to use non default ports 27017 for MongoDB cluster components: `mongod`, `mongos` and `configsvrReplSet` Pods
- K8SPSMDB-1236 - Added a check for a username to be unique when defining it via the Custom Resource manifest
- K8SPSMDB-1253 - Made the `SmartUpdate` the default update strategy

- K8SPSMDB-1276 - Added logic to the getMongoUri function to compare the content of the existing TLS and CA certificate files with the secret data. Files are only overwritten if the data has changed, preventing redundant writes and ensuring smoother operations during backup checks. (Thank you Anton Averianov for reporting and contributing to this issue)
- K8SPSMDB-1316 - Added the ability to define a custom cluster name for `pmm-admin` component
- K8SPSMDB-1325 Added the `directShardOperations` role for a `mongo` user used for monitoring MongoDB 8 and above
- K8SPSMDB-1337 Add imagePullSecrets for PMM and backup images

Bugs Fixed

- K8SPSMDB-1197 - Fixed the healthcheck log rotation routine to delete log file created 1 day before.
- K8SPSMDB-1231 - Fixed the issue with a single-node cluster to temporarily report the Error state during initial provisioning by ignoring the `No mongod containers in running state` error.
- K8SPSMDB-1239 - Fixed the issue with cron jobs running simultaneously
- K8SPSMDB-1245 - Improved Telemetry for cluster-wide deployments to handle both an empty value and a comma-separated list of namespaces
- K8SPSMDB-1256 - Fixed the issue with PBM failing with the `length of read message too large` error by verifying the existence of TLS files when constructing the `PBM_MONGODB_URI` connection string URI
- K8SPSMDB-1263 - Fixed the issue with the Operator losing connection to `mongod` pods during backup and throwing an error by retrying to connect and proceed with the backup
- K8SPSMDB-1274 - Disable balancer before logical restore to meet the PBM restore requirements
- K8SPSMDB-1275 - Fixed the issue with the Operator failing when the `getLastErrorModes` write concern value is set for a replica set by using the data type for a value that matches MongoDB behavior (Thank you user `clrxbl` for reporting and contributing to this issue)
- K8SPSMDB-1294 - Fixed the API mismatch error with the multi-cluster Services (MCS) enabled in the Operator by using the `DiscoveryClient.ServerPreferredResources` method to align with the `kubectl` behavior.
- K8SPSMDB-1302 - Fixed the issue with the Operator being stuck during physical restore when the update strategy is set to SmartUpdate
- K8SPSMDB-1306 - Fixed the Operator panics if a user configures PBM priorities without timeouts
- K8SPSMDB-1347 - Fixed the issue with the Operator throwing errors when auto generating password for multiple users by properly updating the secret after a password generation

Upgrade considerations

The added support for multiple backup storages requires specifying the main storage. If you use a single storage, it will automatically be marked as main in the Custom Resource manifest during the upgrade. If you use multiple storages, you must define one of them as the main storage when you upgrade to version 1.20.0. The following command shows how to set the `s3-us-west` storage as the main one:

```
kubectl patch psmdb my-cluster-name --type=merge --patch '{
    "spec": {
      "crVersion": "1.20.0",
      "image": "percona/percona-server-mongodb:7.0.18-11",
      "backup": {
        "image": "percona/percona-backup-mongodb:2.9.1",
        "storages": {
          "s3-us-west": {
            "main": true
          }
        }
      },
      "pmm": {
        "image": "percona/pmm-client:2.44.1"
      }
    }
  }'
```

Supported software

The Operator was developed and tested with the following software:

- Percona Server for MongoDB 6.0.21-18, 7.0.18-11, and 8.0.8-3.

- Percona Backup for MongoDB 2.9.1.
- PMM Client: 2.44.1

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) 1.30-1.32
- Amazon Elastic Container Service for Kubernetes (EKS) 1.30-1.32
- OpenShift Container Platform 4.14 - 4.18
- Azure Kubernetes Service (AKS) 1.30-1.32
- Minikube 1.35.0 based on Kubernetes 1.32.0

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 the backward compatibility offered by Kubernetes itself.

Percona certified images

Find Percona's certified Docker images that you can use with the Percona Operator for MongoDB in the following table:

Images released with the Operator version 1.20.0:

Image	Digest
percona/percona-server-mongodb-operator:1.20.0 (x86_64)	01da3139b0f7f64a27f3642ca06581ea065a02891b13ce2375d61471011d6dd4
percona/percona-server-mongodb-operator:1.20.0 (ARM64)	26d885398af42d18928f51f070aff770df900eb5ddf46e3e0bc2570720089bb1
percona/pmm-client:2.44.1	8b2eaddffd626f02a2d5318ffebc0c277fe8457da6083b8cfcada9b6e6168616
percona/pmm-client:2.44.1 (ARM64)	337fecd4afdb3f6daf2caa2b341b9fe41d0418a0e4ec76980c7f29be9d08b5ea
percona/percona-backup-mongodb:2.9.1 (x86_64)	976bfbaa548eb70dd90bf0bd2dcfe40b2994d749ef644af3a0590f4856e4d7e2
percona/percona-backup-mongodb:2.9.1 (ARM64)	ebc6e5c5aa3ed97991d3fd90e9201597b485ddc0eae8d7ee4311ecb785c03bf0
percona/percona-server-mongodb:8.0.8-3 (x86_64)	e4580ca292f07fd7800e139121aea4b2c1dfa6aa34f3657d25a861883fd3de41
percona/percona-server-mongodb:8.0.8-3 (ARM64)	96cfee2102499aba05e63ca7862102c2b1da1cf9f4eea0cbea3793a07c183925
percona/percona-server-mongodb:8.0.4-1-multi (x86_64)	873b201ce3d66d97b1225c26db392c5043a73cc19ee8db6f2dc1b8efd4783bcf
percona/percona-server-mongodb:8.0.4-1-multi (ARM64)	222ccf746ad4ffdfccf41b41edaa0d318d28f663e13c9629f8dad5a5078434e5
percona/percona-server-mongodb:7.0.18-11 (x86_64)	0115a72f5e60d86cb4f4b7eae32118c0910e8c96831e013de12798a1771c4c91
percona/percona-server-mongodb:7.0.18-11 (ARM64)	86c17067f3e233f522612389ed2500231cbb22ce93524c476b9aa8d464d06f0b
percona/percona-server-mongodb:7.0.15-9-multi (x86_64)	7bffdf2e71c121e2ab37b4fa7e2f513237abdd65266da384bf8197cee1316917
percona/percona-server-mongodb:7.0.15-9-multi (ARM64)	fdc4875df82572267445811445ebf517f63e509be54d1a2599fe58e1c525e1d8
percona/percona-server-mongodb:7.0.14-8-multi (x86_64)	ed932d4e7231dcb793bf609f781226a8393aa8958b103339f4a503a8f70ed17e
percona/percona-server-mongodb:7.0.14-8-multi (ARM64)	052f84ee926ad9b5146f08a7e887820342d65b757a284c2f0ea8e937bb51cd7b
percona/percona-server-mongodb:7.0.12-7	7f00e19878bd143119772cd5468f1f0f9857dfcd2ae2f814d52ef3fa7cff6899
percona/percona-server-mongodb:6.0.21-18 (x86_64)	579d2fdc617ea42ab2be8c2682955b489dbf49ab19771b7a5d9c77da4dd323e7

percona/percona-server-mongodb:6.0.21-18 (ARM64)	b9d2b7e8c4a97b2d20e2aaccfbd183f65f8ccd9f2ea13939515e18e02bc64871
percona/percona-server-mongodb:6.0.19-16-multi (x86_64)	c8ff08c4b8a96679e2daf4845873fdd4d2c48646b84db19f0c5fe02e8f3808b4
percona/percona-server-mongodb:6.0.19-16-multi (ARM64)	6908b28ced260b762cd38a642c06dd802cbef0a43ab5f22afe7b583b234ebcec
percona/percona-server-mongodb:6.0.18-15-multi (x86_64)	d197ce16ab0eed6df25e632b92dea5ce448e549e02028f39b78f5730c2ffef36
percona/percona-server-mongodb:6.0.18-15-multi (ARM64)	7fd1d8f74f71dea6ad423e8e202a0617bdd1e8783f2b5cb071b5281685ce0adf
percona/percona-server-mongodb:6.0.16-13	1497e58e39497d8425ccd053898dc323338d6eb3f0e3c4c223f9d5a468da7931
percona/percona-server-mongodb:6.0.15-12	f12dd271d78cf3e70088fea0c420e8c03703457d8a5959b645053546bff94dea

Find previous version images in the documentation archive

Percona Operator for MongoDB 1.19.1

- **Date**

  February 20, 2025

- **Installation**

  Installing Percona Operator for MongoDB

Bugs Fixed

- K8SPSMDB-1274: Revert to disabling MongoDB balancer during restores to follow requirements of Percona Backup for MongoDB 2.8.0.

Known limitations

- PBM-1493: For sharded MongoDB 8.0 deployments, Percona Operator for MongoDB versions 1.19.0 and 1.19.1 have a known issue causing point-in-time recovery failures. Avoid upgrading to these Operator versions until a fix is released in Percona Backup for MongoDB and added into the newer versions of the Operator

Supported Platforms

The Operator was developed and tested with Percona Server for MongoDB 6.0.19-16, 7.0.15-9, and 8.0.4-1. Other options may also work but have not been tested. The Operator also uses Percona Backup for MongoDB 2.8.0.

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 1.19.1:

- Google Kubernetes Engine (GKE) 1.28-1.30
- Amazon Elastic Container Service for Kubernetes (EKS) 1.29-1.31
- OpenShift Container Platform 4.14.44 - 4.17.11
- Azure Kubernetes Service (AKS) 1.28-1.31
- Minikube 1.34.0 based on Kubernetes 1.31.0

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 the backward compatibility offered by Kubernetes itself.

Percona Operator for MongoDB 1.19.0

- **Date**

  January 21, 2025

- **Installation**

  [Installing Percona Operator for MongoDB]

Release Highlights

Using remote file server for backups (tech preview)

The new `filesystem` backup storage type was added in this release in addition to already existing `s3` and `azure` types. It allows users to mount a remote file server to a local directory, and make Percona Backup for MongoDB using this directory as a storage for backups. The approach is based on common Network File System (NFS) protocol, and should be useful in network-restricted environments without S3-compatible storage or in cases with a non-standard storage service supporting NFS access.

To use NFS-capable remote file server as a backup storage, user needs to mount the remote storage as a sidecar volume in the `replsets` section of the Custom Resource (and also `configsvrReplSet` in case of a sharded cluster):

```
replsets:
  ...
  sidecarVolumes:
  - name: backup-nfs
    nfs:
      server: "nfs-service.storage.svc.cluster.local"
      path: "/psmdb-some-name-rs0"
  ...
```

Finally, this new storage needs to be configured in the same Custom Resource as a normal storage for backups:

```
backup:
  ...
  storages:
    backup-nfs:
      filesystem:
        path: /mnt/nfs/
          type: filesystem
  ...
  volumeMounts:
  - mountPath: /mnt/nfs/
    name: backup-nfs
```

See more in our documentation about this storage type.

Generated passwords for custom MongoDB users

A new improvement for the declarative management of custom MongoDB users brings the possibility to use automatic generation of users passwords. When you specify a new user in `deploy/cr.yaml` configuration file, you can omit specifying a reference to an already existing Secret with the user's password, and the Operator will generate it automatically:

```
...
users:
  - name: my-user
    db: admin
    roles:
      - name: clusterAdmin
        db: admin
      - name: userAdminAnyDatabase
        db: admin
```

Find more details on this automatically created Secret in our documentation.

Percona Server for MongoDB 8.0 support

Percona Server for MongoDB 8.0 is now supported by the Operator in addition to 6.0 and 7.0 versions. The appropriate images are now included into the list of Percona-certified images. See this blogpost for details about the latest MongoDB 8.0 features with the added reliability and performance improvements.

New Features

- K8SPSMDB-1109: Backups can now be stored on a remote file server
- K8SPSMDB-921: IAM Roles for Service Accounts (IRSA) allow automating access to AWS S3 buckets based on Identity Access Management with no need to specify the S3 credentials explicitly
- K8SPSMDB-1133: Manual change of Replica Set Member Priority in Percona Server MongoDB Operator is now possible with the new `replsetOverrides.MEMBER-NAME.priority` Custom Resource option
- K8SPSMDB-1164: Add the possibility to create users in the `$external` database for external authentication purposes

Improvements

- K8SPSMDB-1123: Percona Server for MongoDB 8.0 is now supported
- K8SPSMDB-1171: The declarative user management was enchanced with the possibility to automatically generate passwords
- K8SPSMDB-1174: Telemetry was improved to to track whether the custom users and roles management, automatic volume expansion, and multi-cluster services features are enabled
- K8SPSMDB-1179: It is now possible to configure externalTrafficPolicy for mongod, configsvr and mongos instances
- K8SPSMDB-1205: Backups in unmanaged clusters are now supported, removing a long-standing limitation of cross-site replication that didn't allow backups on replica clusters

Bugs Fixed

- K8SPSMDB-1215: Fix a bug where ExternalTrafficPolicy was incorrectly set for LoadBalancer and NodePort services (Thanks to Anton Averianov for contributing)
- K8SPSMDB-675: Fix a bug where disabling sharding failed on a running cluster with enabled backups
- K8SPSMDB-754: Fix a bug where some error messages had "INFO" log level and therefore were not seen in logs with the "ERROR" log level turned on
- K8SPSMDB-1088: Fix a bug which caused the Operator starting two backup operations if the user patches the backup object while its state is empty or Waiting
- K8SPSMDB-1156: Fix a bug that prevented the Operator with enabled backups to recover from invalid TLS configurations (Thanks to KOS for reporting)
- K8SPSMDB-1172: Fix a bug where backup user's password username with special characters caused Percona Backup for MongoDB to fail
- K8SPSMDB-1212: Stop disabling balancer during restores, because it is not required for Percona Backup for MongoDB 2.x

Deprecation, Rename and Removal

- The `psmdbCluster` option from the `deploy/backup/backup.yaml` manifest used for on-demand backups, which was deprecated since the Operator version 1.12.0 in favor of the `clusterName` option, has been removed and is no longer supported.
- Percona Server for MongoDB 5.0 has reached its end of life and in no longer supported by the Operator

Supported Platforms

The Operator was developed and tested with Percona Server for MongoDB 6.0.19-16, 7.0.15-9, and 8.0.4-1. Other options may also work but have not been tested. The Operator also uses Percona Backup for MongoDB 2.8.0.

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 1.19.0:

- Google Kubernetes Engine (GKE) 1.28-1.30
- Amazon Elastic Container Service for Kubernetes (EKS) 1.29-1.31
- OpenShift Container Platform 4.14.44 - 4.17.11
- Azure Kubernetes Service (AKS) 1.28-1.31
- Minikube 1.34.0 based on Kubernetes 1.31.0

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 the backward compatibility offered by Kubernetes itself.

Percona Operator for MongoDB 1.18.0

- **Date**

  November 14, 2024

- **Installation**

  Installing Percona Operator for MongoDB

Release Highlights

Enhancements of the declarative user management

The declarative management of custom MongoDB users was improved compared to its initial implementation in the previous release, where the Operator did not track and sync user-related changes in the Custom Resource and the database. Also, starting from now you can create custom MongoDB roles on various databases just like users in the `deploy/cr.yaml` manifest:

```
...
roles:
  - name: clusterAdmin
    db: admin
  - name: userAdminAnyDatabase
    db: admin
```

See the documentation to find more details about this feature.

Support for selective restores

Percona Backup for MongoDB 2.0.0 has introduced a new functionality that allows partial restores, which means selectively restoring only with the desired subset of data. Now the Operator also supports this feature, allowing you to restore a specific database or a collection from a backup. You can achieve this by using an additional `selective` section in the `PerconaServerMongoDBRestore` Custom Resource:

```
spec:
  selective:
    withUsersAndRoles: true
    namespaces:
    - "db.collection"
```

You can find more on selective restores and their limitations in our documentation.

Splitting the replica set of the database cluster over multiple Kubernetes clusters

Recent improvements in cross-site replication made it possible to keep the replica set of the database cluster in different data centers. The Operator itself cannot deploy MongoDB replicas to other data centers, but this still can be achieved with a number of Operator deployments, equal to the size of your replica set: one Operator to control the replica set via cross-site replication, and at least two Operators to bootstrap the unmanaged clusters with other MongoDB replica set instances. Splitting the replica set of the database cluster over multiple Kubernetes clusters can be useful to get a fault-tolerant system in which all replicas are in different data centers. You can find more about configuring such a multi-datacenter MongoDB cluster and the limitations of this solution on the dedicated documentation page.

New Features

- K8SPSMDB-894: It is now possible to restore a subset of data (a specific database or a collection) from a backup which is useful to reduce time on restore operations when fixing corrupted data fragment
- K8SPSMDB-1113: The new `percona.com/delete-pitr-chunks` finalizer allows the deletion of PITR log files from the backup storage when deleting a cluster so that leftover data does not continue to take up space in the cloud
- K8SPSMDB-1124 and K8SPSMDB-1146: Declarative user management now covers creating and managing user roles, and syncs user-related changes between the Custom Resource and the database
- K8SPSMDB-1140 and K8SPSMDB-1141: Multi-datacenter cluster deployment is now possible

Improvements

- K8SPSMDB-739: A number of Service exposure options in the `replsets`, `sharding.configsvrReplSet`, and `sharding.mongos` were renamed for unification with other Percona Operators
- K8SPSMDB-1002: New Custom Resource options under the replsets.primaryPreferTagSelector` subsection allow providing Primary instance selection preferences based on specific zone and region, which may be especially useful within the planned zone switchover process (Thanks to sergelogvinov for contribution)
- K8SPSMDB-1096: Restore logs were improved to contain pbm-agent logs in mongod containers, useful to debug failures in the backup restoration process
- K8SPSMDB-1135: Split-horizon DNS for external (unmanaged) nodes is now configurable via the `replsets.externalNodes` subsection in Custom Resource
- K8SPSMDB-1152: Starting from now, the Operator uses multi-architecture images of Percona Server for MongoDB and Percona Backup for MongoDB, making it easier to deploy a cluster on ARM
- K8SPSMDB-1160: The PVC resize feature introduced in previous release can now be enabled or disabled via the `enableVolumeExpansion` Custom Resource option (`false` by default), which protects the cluster from storage resize triggered by mistake
- K8SPSMDB-1132: A new `secrets.keyFile` Custom Resource option allows to configure custom name for the Secret with the MongoDB internal auth key file

Bugs Fixed

- K8SPSMDB-912: Fix a bug where the full backup connection string including the password was visible in logs in case of the Percona Backup for MongoDB errors
- K8SPSMDB-1047: Fix a bug where the Operator was changing writeConcernMajorityJournalDefault to “true” during the replica set reconfiguring, ignoring the value set by user
- K8SPSMDB-1168: Fix a bug where successful backups could obtain a failed state in case of the Operator configured with `watchAllNamespaces: true` and having the same name for MongoDB clusters across multiple namespaces (Thanks to Markus Küffner for contribution)
- K8SPSMDB-1170: Fix a bug that prevented deletion of a cluster with the active `percona.com/delete-psmdb-pods-in-order` finalizer in case of the cluster error state (e.g. when mongo replset failed to reconcile)
- K8SPSMDB-1184: Fix a bug where the Operator failed to reconcile when using the container security context with `readOnlyRootFilesystem` set to `true` (Thanks to applejag for contribution)
- K8SPSMDB-1180: Fix a bug where rotation functionality didn’t work for scheduled backups

Deprecation, Rename and Removal

- The new `enableVolumeExpansion` Custom Resource option allows users to disable the automated storage scaling with Volume Expansion capability. The default value of this option is `false`, which means that the automated scaling is turned off by default.
- A number of Service exposure Custom Resource options in the `replsets`, `sharding.configsvrReplSet`, and `sharding.mongos` subsections were renamed to provide a unified experience with other Percona Operators:
  - `expose.serviceAnnotations` option renamed to `expose.annotations`
  - `expose.serviceLabels` option renamed to `expose.labels`
  - `expose.exposeType` option renamed to `expose.type`

Supported Platforms

The Operator was developed and tested with Percona Server for MongoDB 5.0.29-25, 6.0.18-15, and 7.0.14-8. Other options may also work but have not been tested. The Operator also uses Percona Backup for MongoDB 2.7.0.

The following platforms were tested and are officially supported by the Operator 1.18.0:

- Google Kubernetes Engine (GKE) 1.28-1.30
- Amazon Elastic Container Service for Kubernetes (EKS) 1.28-1.31
- OpenShift Container Platform 4.13.52 - 4.17.3
- Azure Kubernetes Service (AKS) 1.28-1.31
- Minikube 1.34.0 based on Kubernetes 1.31.0

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 the backward compatibility offered by Kubernetes itself.

Percona Operator for MongoDB 1.17.0

- **Date**

  September 09, 2024
- **Installation**

  [Installing Percona Operator for MongoDB]

Release Highlights

Declarative user management (technical preview)

Before the Operator version 1.17.0 custom MongoDB users had to be created manually. Now the declarative creation of custom MongoDB users is supported via the `users` subsection in the Custom Resource. You can specify a new user in `deploy/cr.yaml` manifest, setting the user's login name and database, PasswordSecretRef (a reference to a key in a Secret resource containing user's password) and as well as MongoDB roles on various databases which should be assigned to this user:

```
...
users:
- name: my-user
  db: admin
  passwordSecretRef:
    name: my-user-password
    key: my-user-password-key
  roles:
    - name: clusterAdmin
      db: admin
    - name: userAdminAnyDatabase
      db: admin
```

See documentation to find more details about this feature with additional explanations and the list of current limitations.

Liveness check improvements

Several improvements in logging were made related to the liveness checks, to allow getting more information for debugging, and to make these logs persist on failures to allow further examination.

Liveness check logs are stored in the `/data/db/mongod-data/logs/mongodb-healthcheck.log` file, which can be accessed in the corresponding Pod if needed. Starting from now, Liveness check generates more log messages, and the default log level is set to `DEBUG`.

Each time the health check fails, the current log is saved to a gzip compressed file named `mongodb-healthcheck-<timestamp>.log.gz`, and the `mongodb-healthcheck.log` log file is reset. Logs older than 24 hours are automatically deleted.

New Features

- K8SPSMDB-253: It is now possible to create and manage users via the Custom Resource

Improvements

- K8SPSMDB-899: Add Labels for all Kubernetes objects created by Operator (backups/restores, Secrets, Volumes, etc.) to make them clearly distinguishable
- K8SPSMDB-919: The Operator now checks if the needed Secrets exist and connects to the storage to check the validity of credentials and the existence of a backup before starting the restore process
- K8SPSMDB-934: Liveness checks are providing more debug information and keeping separate log archives for each failure with the 24 hours retention
- K8SPSMDB-1057: Finalizers were renamed to contain fully qualified domain names (FQDNs), avoiding potential conflicts with other finalizer names in the same Kubernetes environment
- K8SPSMDB-1108: The new Custom Resource option allows setting custom containerSecurityContext for PMM containers
- K8SPSMDB-994: Remove a limitation where it wasn't possible to create a new cluster with splitHorizon enabled, leaving the only way to enable it later on the running cluster

Bugs Fixed

- K8SPSMDB-925: Fix a bug where the Operator generated “failed to start balancer” and “failed to get mongos connection” log messages when using Mongos with servicePerPod and LoadBalancer services, while the cluster was operating properly
- K8SPSMDB-1105: The memory requests and limits for backups were increased in the `deploy/cr.yaml` configuration file example to reflect the Percona Backup for MongoDB minimal pbm-agents requirement of 1 Gb RAM needed for stable operation
- K8SPSMDB-1074: Fix a bug where MongoDB Cluster could not failover in case of all Pods downtime and `exposeType` Custom Resource option set to either `NodePort` or `LoadBalancer`
- K8SPSMDB-1089: Fix a bug where it was impossible to delete a cluster in error state with finalizers present
- K8SPSMDB-1092: Fix a bug where Percona Backup for MongoDB log messages during physical restore were not accessible with the `kubectl logs` command
- K8SPSMDB-1094: Fix a bug where it wasn’t possible to create a new cluster with `upgradeOptions.setFCV` Custom Resource option set to `true`
- K8SPSMDB-1110: Fix a bug where nil Custom Resource annotations were causing the Operator panic

Deprecation, Rename and Removal

Finalizers were renamed to contain fully qualified domain names to comply with the Kubernetes standards.

- `PerconaServerMongoDB` Custom Resource:
  - `delete-psmdb-pods-in-order` finalizer renamed to `percona.com/delete-psmdb-pods-in-order`
  - `delete-psmdb-pvc` finalizer renamed to `percona.com/delete-psmdb-pvc`
- `PerconaServerMongoDBBackup` Custom Resource:
  - `delete-backup` finalizer renamed to `percona.com/delete-backup`

Key change in `psmdb-db` Helm chart: the parameter for defining system users is renamed from `users` to `systemUsers`. The `users` parameter now handles the new Declarative user management feature. This change impacts users upgrading to this version via Helm: make sure that values manifests are changed accordingly.

Supported Platforms

The Operator was developed and tested with Percona Server for MongoDB 5.0.28-24, 6.0.16-13, and 7.0.12-7. Other options may also work but have not been tested. The Operator also uses Percona Backup for MongoDB 2.5.0.

The following platforms were tested and are officially supported by the Operator 1.17.0:

- Google Kubernetes Engine (GKE) 1.27-1.30
- Amazon Elastic Container Service for Kubernetes (EKS) 1.28-1.30
- OpenShift Container Platform 4.13.48 - 4.16.9
- Azure Kubernetes Service (AKS) 1.28-1.30
- Minikube 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 the backward compatibility offered by Kubernetes itself.

Percona Operator for MongoDB 1.16.2

- **Date**

  July 23, 2024

- **Installation**

  [Installing Percona Operator for MongoDB]

Bugs Fixed

- K8SPSMDB-1117: Fix a bug where the Operator incorrectly compares `G` with `Gi` and and tries to downscale PVC size after upgrade

Supported Platforms

The Operator was developed and tested with Percona Server for MongoDB 5.0.26-22, 6.0.15-12, and 7.0.8-5. Other options may also work but have not been tested. The Operator also uses Percona Backup for MongoDB 2.4.1.

The following platforms were tested and are officially supported by the Operator 1.16.2:

- Google Kubernetes Engine (GKE) 1.26-1.29
- Amazon Elastic Container Service for Kubernetes (EKS) 1.26-1.29
- OpenShift Container Platform 4.12.56 - 4.15.11
- Azure Kubernetes Service (AKS) 1.27-1.29
- Minikube 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 the backward compatibility offered by Kubernetes itself.

Percona Operator for MongoDB 1.16.1

- **Date**

  June 24, 2024

- **Installation**

  Installing Percona Operator for MongoDB

Bugs Fixed

- K8SPSMDB-1101: Fix a bug where manually generated TLS certificates couldn't be applied because Operator was replacing them with auto-generated ones

Supported Platforms

The Operator was developed and tested with Percona Server for MongoDB 5.0.26-22, 6.0.15-12, and 7.0.8-5. Other options may also work but have not been tested. The Operator also uses Percona Backup for MongoDB 2.4.1.

The following platforms were tested and are officially supported by the Operator 1.16.1:

- Google Kubernetes Engine (GKE) 1.26-1.29
- Amazon Elastic Container Service for Kubernetes (EKS) 1.26-1.29
- OpenShift Container Platform 4.12.56 - 4.15.11
- Azure Kubernetes Service (AKS) 1.27-1.29
- Minikube 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 the backward compatibility offered by Kubernetes itself.

Percona Operator for MongoDB 1.16.0

- **Date**

  May 24, 2024
- **Installation**

  Installing Percona Operator for MongoDB

Release Highlights

General availability of Physical Backups

Two releases ago we added experimental support for Physical Backups and Restores to significantly reduce Recovery Time Objective (RTO), especially for big data sets. With this release Percona announces the general availability of physical backups and restores for Percona Server for MongoDB with the Operator.

Automated volume expansion

Kubernetes supports the Persistent Volume expansion as a stable feature since v1.24. Using it with the Operator previously involved manual operations. Now this is automated, and users can resize their PVCs by just changing the value of the `resources.requests.storage` option in the PerconaServerMongoDB custom resource. This feature is in a technical preview stage and is not recommended for production environments.

Update from September 16, 2025 Though the Operator automates the storage resizing, the users must still trigger the process by modifying the Custom Resource and applying the new configuration.

Support for MongoDB 7

Starting from this release, MongoDB 7.0 is now supported. Read our take on top-5 changes in MongoDB version 7 in this blog post.

Support for ARM architecture (technical preview)

ARM architecture meets the intensive growth of its usage nowadays, both in a segment of highly efficient cloud computing based on systems like AWS Graviton, and the Internet of Things or Edge. Officially certified images for ARM are now available for the Operator, as well as Percona Server for MongoDB and Percona Backup for MongoDB, while database monitoring based on PMM Client is yet to follow.

Fixing the overloaded allowUnsafeConfigurations flag

In the previous Operator versions `allowUnsafeConfigurations` Custom Resource option was used to allow configuring a cluster with unsafe parameters, such as starting it with less than 3 replica set instances. In fact, setting this option to `true` resulted in a wide range of reduced safety features without the user's explicit intent: disabling TLS, allowing backups in unhealthy clusters, etc.

With this release, a separate `unsafeFlags` Custom Resource section is introduced for the fine-grained control of the safety loosening features:

```
unsafeFlags:
  tls: false
  replsetSize: false
  mongosSize: false
  terminationGracePeriod: false
  backupIfUnhealthy: false
```

Also, TLS configuration is now enabled or disabled by a special `tls.mode` Custom Resource option, which can be set to `disabled`, `allowTLS`, `preferTLS`, or `requireTLS` values.

New Features

- K8SPSMDB-1000: Users who store backups on Azure Blob Storage can now use private endpoints
- K8SPSMDB-1055: The `kubectl get psmdb-backup` command now shows latest restorable time to make it easier to pick a point-in-time recovery target
- K8SPSMDB-491: It is now possible to specify the existing cert-manager issuer which should be used by the Operator
- K8SPSMDB-733: It is now possible to resize Persistent Volume Claims by patching the PerconaServerMongoDB custom resource: change `persistentVolumeClaim.resources.requests.storage` and let the Operator do the scaling

Improvements

- K8SPSMDB-1004: Exposing replica set with split-horizon DNS allows to specify URIs with non-standard port numbers, which are particularly useful with the NodePort service type
- K8SPSMDB-1013: MongoDB 7.0 is now supported.
- K8SPSMDB-1015: Information about backup and restore operations is now included in the Operator's logs
- K8SPSMDB-951, K8SPSMDB-979 and K8SPSMDB-1021: The Operator now allows setting custom configuration for Percona Backup for MongoDB through the set of new Custom Resource options under `backup.configuration.backupOptions`, `backup.configuration.restoreOptions`, and `backup.storages.s3.retryer` subsections
- K8SPSMDB-1029: Mongod is now run in quiet mode by default to reduce the amount of log messages
- K8SPSMDB-1032: It is now possible to define TCP port for mongos Service when it is exposed through a NodePort (thanks to Mike Devresse for contribution)
- K8SPSMDB-1062: The Operator now sets appProtocol to `mongo` for Service objects, which is useful for service mesh implementations (thanks to Søren Mathiasen for contribution)
- K8SPSMDB-732: Integration of the Operator with OpenLDAP can now be secured by using TLS connections
- K8SPSMDB-755: New `allowInvalidCertificates` option allows to enable or disable bypassing MongoDB Shell checks for the certificates presented by the mongod/mongos instance, useful for self-signed certificates
- K8SPSMDB-948: Officially certified images for ARM architecture are now available for the Operator, as well as Percona Server for MongoDB and Percona Backup for MongoDB
- K8SPSMDB-993: To avoid backup fail on clusters where Percona Backup for MongoDB resync process takes too long, the Operator now checks, if there is still a resync operation working, with exponentially increasing interval and total wait time until failure equal to 8715 seconds
- K8SPSMDB-995: The Operator now allows storing key for backups server-side AWS KMS encryption in a Secret configurable with the `secrets.sse` Custom Resource option
- K8SPSMDB-780: Removing `allowUnsafeConfigurations` Custom Resource option in favor of fine-grained safety control in the `unsafeFlags` subsection
- K8SPSMDB-1042: Helm chart for Percona Server for MongoDB now accepts replica set options as the map argument instead of the array one used in previous releases; this simplifies how arguments are specified in the command line and allows to specify only part of the replica set parameters, relying on the default values for the other part. **Take this change into account** if you are installing database via helm and want to use set of custom options from previous releases

Bugs Fixed

- K8SPSMDB-1011: Fix a bug where custom logins for system users stopped working after deleting and recreating back the users Secret (thanks for Patrick Wolleb for report)
- K8SPSMDB-1014: Fix a bug that certificate rotation was bringing the sharded MongoDB cluster down for clusters originally created with the Operator version prior to 1.15.0 (thanks to Stiliyan Stefanov for reporting)
- K8SPSMDB-1018: Fix a bug where MongoDB container startup would fail if the MongoDB image being used contained the numactl package
- K8SPSMDB-1024: Fix a bug where environment variable wasn't properly updated in the Percona Backup for MongoDB container entry script (thanks to Rockawear for contribution)
- K8SPSMDB-1035: Fixed a bug where the empty `secretName` field was not allowed for backup jobs that might not need it when accessing AWS S3 buckets based on IAM roles (thanks to Sergey Zelenov for contribution)
- K8SPSMDB-1036: Fix a bug due to which restoring backup to a new cluster was broken by incompatibility with Percona Backup for MongoDB 2.3.0
- K8SPSMDB-1038: Fix a bug where mongos Services were deleted if the cluster was set to paused state
- K8SPSMDB-1039: Fix a bug which prevented deleting PMM agent from the PMM Server inventory on Pod termination
- K8SPSMDB-1058: A minor missing privileges issue caused flooding MongoDB logs with "Checking authorization failed" errors
- K8SPSMDB-1070: Fix a bug where panic was happening in `delete-psmdb-pods-in-order` finalizer if the cluster was deleted prior to creating Pods
- K8SPSMDB-940: Fix a bug due to which the Operator didn't allow to set serviceAccount for mongos Pods
- K8SPSMDB-985: Fix a bug where `pbmPod` key in backup object was only showing one replica/pod

Deprecation and removal

- Starting from now, `allowUnsafeConfigurations` Custom Resource option is deprecated in favor of a number of options under the `unsafeFlags` subsection. Setting `allowUnsafeConfigurations` won't have any effect; upgrading existing clusters with `allowUnsafeConfigurations=true` will cause everything under unsafeFlags set to true and TLS functionality disabled

- MongoDB 4.4 support in the Operator has reached its end-of-life. Starting from now Percona will not provide officially certified images for it. Make sure that you have a supported MongoDB version before upgrading the Operator to 1.16.0. You can use major version upgrade functionality.

Supported Platforms

The Operator was developed and tested with Percona Server for MongoDB 5.0.26-22, 6.0.15-12, and 7.0.8-5. Other options may also work but have not been tested. The Operator also uses Percona Backup for MongoDB 2.4.1.

The following platforms were tested and are officially supported by the Operator 1.16.0:

- Google Kubernetes Engine (GKE) 1.26-1.29
- Amazon Elastic Container Service for Kubernetes (EKS) 1.26-1.29
- OpenShift Container Platform 4.12.56 - 4.15.11
- Azure Kubernetes Service (AKS) 1.27-1.29
- Minikube 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 the backward compatibility offered by Kubernetes itself.

Percona Operator for MongoDB 1.15.0

- **Date**

  October 9, 2023

- **Installation**

  Installing Percona Operator for MongoDB

Release Highlights

Physical Backups now support Point-in-time Recovery (in tech preview)

In the previous 1.14.0 release we added support for Physical Backups and Restores to significantly reduce Recovery Time Objective (RTO .)), especially for big data sets. But the problem with losing data between backups - in other words Recovery Point Objective (RPO) - for physical backups was not solved. With this release users can greatly reduce RPO by leveraging the Point-in-time Recovery feature in the Operators. Under the hood we store logical oplogs along with physical backups into the object storage. Read more about this feature in our documentation.

Encrypted backups with Server Side Encryption (SSE)

Backups stored on S3 compatible storage can now be encrypted with Server Side Encryption (SSE) to pass certain compliance or security requirements. Users can leverage integration with AWS KMS or just encrypt/decrypt backups with AES-256 encryption algorithm. It is important to remember that Operator does not store keys and users can choose which key storage to use.

New Features

- K8SPSMDB-227 The new `topologySpreadConstraints` Custom Resource option allows to use Pod Topology Spread Constraints to achieve even distribution of Pods across the Kubernetes cluster
- K8SPSMDB-792 and K8SPSMDB-974 The new “sleep infinity” mode available for replset and config server containers allows running the Pod without starting mongod useful to examine a problematic Pod that is constantly restarting
- K8SPSMDB-801 It is now possible to delete a backup with its PITR data on retention period or with `delete-backup` finalizer (there were no PITR files deletion in previous versions)
- K8SPSMDB-926 Point-in-time recovery is now supported with physical backups to significantly reduce Recovery Point Objective (RPO)
- K8SPSMDB-961 The new `sharding.balancer.enabled` Custom Resource option allows to disable Load Balancer on a cross-site replication managed cluster

Improvements

- K8SPSMDB-662 Restoring a backup with point-in-time recovery can now be easily done to a latest available position by setting `pitr.type` PerconaServerMongoDBRestore Custom Resource option to `latest`
- K8SPSMDB-774 The Transport encryption documentation now includes details on updating TLS certificates
- K8SPSMDB-807 A custom name for a Replica Set config server instead of the default `cfg` one can be set in the custom configuration, which can be useful for migration purposes
- K8SPSMDB-814 and K8SPSMDB-927 The new `terminationGracePeriodSeconds` Custom Resource option allows to set termination period for Replica Set containers, useful to cleanly shutdown clusters with big data sets
- K8SPSMDB-850 Server Side Encryption for backups with for S3 and S3-compatible storage is now supported (thanks to Mert Gönül for contribution)
- K8SPSMDB-903 The backup destination URI now includes bucket/container name, allowing the user to specify the full path to the backup as an easy to read string
- K8SPSMDB-924 The token associated with the operator’s ServiceAccount is no longer printed in the log when a scheduled backup is running; this improves security and avoids logging uninformative elements
- K8SPSMDB-938 Configuring Kubernetes host aliases is now possible for replica set, config server, and mongos Pods
- K8SPSMDB-946 The psmdb-backup object now includes the name of the Pod that made the backup, to save users from searching for the correct Pod to examine the Percona Backup for MongoDB logs (previously it was necessary to check replica set Pods one by one until logs were found)

- K8SPSMDB-976 The Operator now does not start backups if storages or credentials are not set, avoiding fruitless attempts to configure Percona Backup for MongoDB and cluster state repeatedly changing between ready and error
- K8SPSMDB-929 Using split-horizon DNS for the external access to MongoDB Replica Set Pods of the exposed cluster is now possible

Bugs Fixed

- K8SPSMDB-913 Fix a bug due to which restoring a backup on a cluster with mongos exposed via LoabBalancer resulted in recreating mongos Service with a new IP address
- K8SPSMDB-956 Fix a bug that certificate rotation was bringing the sharded MongoDB cluster down (thanks to Stiliyan for reporting)
- K8SPSMDB-854 Backup stucks after cluster was exposed
- K8SPSMDB-977 The out of memory problem could cause cluster got stuck in the "initializing" state at reconciliation
- K8SPSMDB-778 Fix a bug due to which the Operator did not delete arbiter instances during replica set deletion
- K8SPSMDB-791 Fix a bug which prevented setting `LoadBalancerSourceRanges` Custom Resource option when `replsets.expose.exposeType` is set to `Loadbalancer`
- K8SPSMDB-813 Fix a bug due to which secure connection was not used for MongoDB Liveness check (thanks to t-yrka for contribution)
- K8SPSMDB-818 Fix a bug where `clusterMonitor` user had not enough permissions for PMM monitoring with `--enable-all-collectors` flag turned on
- K8SPSMDB-872 The Operator didn't prevent attempts to restore a backup with "error" status, which could cause the cluster got stuck in the "initializing" state
- K8SPSMDB-876 Fix a bug due to which `delete-psmdb-pods-in-order` finalizer, intended to shutdown primary Pod last, affected only shards and did not affect config replica set
- K8SPSMDB-911 Fix a bug where connection string with credentials was included in the backup-agent container logs
- K8SPSMDB-958 Fix insufficient permissions issue that didn't allow to monitor mongos instances with Percona Monitoring and Management (PMM)
- K8SPSMDB-962 Fix a memory leak due to which the Operator's Pod continually increased both CPU and memory usage in cluster-wide mode (with an unmanaged cluster)
- K8SPSMDB-968 Fix a bug due to which the endpoints list returned by `kubectl get psmdb` command contained fully qualified domain names (FQDN) instead of IP addresses when the replset was exposed as a LoadBalancer and the clusterServiceDNSMode was set to Internal

Deprecation and removal

- K8SPSMDB-883 The `spec.mongod` section deprecated in the Operator version 1.12.0 is finally removed from the Custom Resource configuration. If you have encryption disabled using the deprecated `mongod.security.enableEncryption` option, you need to set encryption disabled with custom configuration before removing `mongod` section (and before upgrade):

```
spec:
  ...
  replsets:
    - name: rs0
      ...
      configuration: |
        security:
          enableEncryption: false
        ...
```

Supported Platforms

The Operator was developed and tested with Percona Server for MongoDB 4.4.24, 5.0.20, and 6.0.9. Other options may also work but have not been tested. The Operator also uses Percona Backup for MongoDB 2.3.0.

The following platforms were tested and are officially supported by the Operator 1.15.0:

- Google Kubernetes Engine (GKE) 1.24-1.28
- Amazon Elastic Container Service for Kubernetes (EKS) 1.24-1.28
- OpenShift Container Platform 4.11 - 4.13
- Azure Kubernetes Service (AKS) 1.25-1.28

- Minikube 1.31.2 (based on Kubernetes 1.28)

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 the backward compatibility offered by Kubernetes itself.

Percona Operator for MongoDB 1.14.0

- **Date**

  March 13, 2023

- **Installation**

  Installing Percona Operator for MongoDB

Release Highlights

- Backups and Restores are critical for business continuity. With this release you can significantly reduce your Recovery Time Objective (RTO) with Physical backups support in the Operator. The feature is now in technical preview.
- MongoDB 6.0 comes with a variety of improvements and new features. It is now fully supported by the Operator. See our documentation to learn how to upgrade.

New Features

- K8SPSMDB-713 Physical backups are now supported by the Operator to recover big data sets faster
- K8SPSMDB-737 MongoDB 6.0 is now officially supported in addition to 4.x and 5.x versions. Read more about version 6 in our blog post
- K8SPSMDB-824 New `ignoreAnnotations` and `ignoreLabels` Custom Resource options allow to list specific annotations and labels for Kubernetes Service objects, which the Operator should ignore (useful with various Kubernetes flavors which add annotations to the objects managed by the Operator)

Improvements

- K8SPSMDB-658 The Operator log messages appearing during the pause/unpause of the cluster were improved to more clearly indicate this event
- K8SPSMDB-708 The new `initContainerSecurityContext` option allows to configure securityContext for the container which can be used instead of the official image during the initial Operator installation
- K8SPSMDB-721 The backup subsystem was improved so that database is not crashing in case if the backup agent is not able to connect to MongoDB (e.g. due to misconfigured password)
- K8SPSMDB-758 The ServiceMesh fully qualified domain names (FQDNs) for config servers are now prioritized if DNSMode is set to ServiceMesh (thanks to Jo Lyshoel for contribution)
- K8SPSMDB-793 It is now possible to set annotations and labels for Persistent Volume Claims for better integration with Cloud Native tools
- K8SPSMDB-803 The Operator now does not attempt to start Percona Monitoring and Management (PMM) client sidecar if the corresponding secret does not contain the `pmmserver` or `pmmserverkey` key
- K8SPSMDB-817 Adding external nodes to the cluster is now allowed even when the replica set is not exposed. This unblocks the creation of complex multi-cluster topologies
- K8SPSMDB-844 Update the RuntimeClass API version to `v1` from the `v1beta1`, which was already deprecated since the Kubernetes version 1.22
- K8SPSMDB-848 Remove formatted strings from log messages to avoid confronting with structured logging based on key-value pairs
- K8SPSMDB-882 Percona Server for MongoDB Helm chart now persists data by default instead of deleting Persistent Volumes after the cluster deletion
- CLOUD-768 Helm charts now use random passwords generated by the Operator by default instead of providing pre-configured passwords specified in the values file
- K8SPSMDB-853 To improve the operator we capture anonymous telemetry and usage data. In this release we add more data points to it
- K8SPSMDB-867 The Operator now configures replset members using local fully-qualified domain names (FQDN) resolvable and available only from inside the cluster instead of using IP addresses; the old behavior can be restored by setting the `clusterServiceDNSMode` option to `External`

Bugs Fixed

- K8SPSMDB-784 Fix a bug due to which the `enableEncryption` MongoDB configuration option was always activated when using psmdb-db Helm Chart
- K8SPSMDB-796 Fix a bug due to which backup failed if replica set was exposed
- K8SPSMDB-854 Fix a bug due to which backup got stuck after the cluster was exposed

- K8SPSMDB-471 Fix a bug due to which in case of scheduled backups with error status `delete-backup` finalizer didn't allow to delete the appropriate failed resources and the Kubernetes namespace (thanks to Aliaksandr Karavai for reporting)
- K8SPSMDB-674 Fix a bug that caused the Operator not deleting unneeded Services after the replica set exposing is turned off
- K8SPSMDB-742 Fix a bug that caused the updates of the `sharding.mongos.expose.serviceAnnotations` option to be silently rejected
- K8SPSMDB-766 and K8SPSMDB-767 Fix a bug where the combination of `delete-psmdb-pods-in-order` and `delete-psmdb-pvc` finalizers was not working
- K8SPSMDB-770 We now mention the namespace name in the log message to ease debugging when the cluster-wide mode is used
- K8SPSMDB-797 Fix the backup/restore documentation not clearly mentioning that user should specify the bucket for the S3 storage
- K8SPSMDB-820 Fix a bug which prevented the parallel backup jobs execution for different MongoDB clusters in the cluster-wide mode
- K8SPSMDB-823 Fix a bug where backups were not working in case of ReplicaSet exposed with NodePort
- K8SPSMDB-836 Fix backups being incorrectly marked as error while still being in starting status
- K8SPSMDB-841 Fix a bug which turned the cluster into unready status after switching from the LoadBalancer expose to ClusterIP
- K8SPSMDB-843 Fix a bug which made the cluster unable to start if it was recreated with the same Custom Resource after delete without deleting PVCs and Secrets
- K8SPSMDB-846 Fix a bug due to which scaling the replica set down to 1 instance caused the last Pod to remain Secondary instead of becoming Primary
- K8SPSMDB-866 Fix the bug due to which the Operator was continuously flooding the log with error messages if the PMM server credentials were missing

Known Issues and Limitations

- K8SPSMDB-875 Physical backups cannot be restored on the clusters with arbiter, non-voting, or delayed members due to current Percona Backup for MongoDB limitations
- K8SPSMDB-846 After switching the cluster to unsafe mode by setting allowUnsafeConfig: true, it is not possible to switch back into safe mode. The user can still scale the cluster safely, but the flag is ignored

Supported Platforms

The Operator was developed and tested with Percona Server for MongoDB 4.4.18, 5.0.14, and 6.0.4. Other options may also work but have not been tested.

The following platforms were tested and are officially supported by the Operator 1.14.0:

- Google Kubernetes Engine (GKE) 1.22 - 1.25
- Amazon Elastic Container Service for Kubernetes (EKS) 1.22 - 1.24
- OpenShift Container Platform 4.10 - 4.12
- Azure Kubernetes Service (AKS) 1.23 - 1.25
- Minikube 1.29

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 the backward compatibility offered by Kubernetes itself.

Percona Operator for MongoDB 1.13.0

- **Date**

  September 15, 2022

- **Installation**

  Installing Percona Operator for MongoDB

Release Highlights

- Azure Kubernetes Service (AKS) is now officially supported platform, so developers and vendors of the solutions based on the Azure platform can take advantage of the official support from Percona or just use officially certified Percona Operator for MongoDB images
- Starting from now, the Operator can be installed in multi-namespace (so-called “cluster-wide”) mode, when a single Operator can be given a list of namespaces in which to manage Percona Server for MongoDB clusters

New Features

- K8SPSMDB-203 Support for the cluster-wide operator mode allowing one Operator to watch for Percona Server for MongoDB Custom Resources in several namespaces
- K8SPSMDB-287 Support for the HashiCorp Vault for encryption keys as a universal, secure and reliable way to store and distribute secrets without depending on the operating system, platform or cloud provider
- K8SPSMDB-704 Support for the Azure Kubernetes Service (AKS)

Improvements

- K8SPSMDB-515 Allow setting requireTLS mode for MongoDB through the Operator to enforce security by restricting each MongoDB server to use TLS/SSL encrypted connections only
- K8SPSMDB-636 An additional `databaseAdmin` user was added to the list of system users which are automatically created by the Operator. This user is intended to provision databases, collections and perform data modifications
- K8SPSMDB-699 Disable automated upgrade by default to prevent an unplanned downtime for user applications and to provide defaults more focused on strict user’s control over the cluster
- K8SPSMDB-725 Configuring the log structuring and leveling is now supported using the `LOG_STRUCTURED` and `LOG_LEVEL` environment variables. This reduces the information overload in logs, still leaving the possibility of getting more details when needed, for example, for debugging
- K8SPSMDB-719 Details about using sharding, Hashicorp Vault and cluster-wide mode were added to telemetry
- K8SPSMDB-715 Starting from now, the Opearator changed its API version to v1 instead of having a separate API version for each release. Three last API version are supported in addition to `v1`, which substantially reduces the size of Custom Resource Definition to prevent reaching the etcd limit
- K8SPSMDB-709 Make it possible to use API Key to authorize within Percona Monitoring and Management Server as a more convenient and modern alternative password-based authentication
- K8SPSMDB-707 Allow to set Service labels for replica set, config servers and mongos in Custom Resource to enable various integrations with cloud providers or service meshes

Bugs Fixed

- K8SPSMDB-702 Fix a bug which resulted in always using the `force` option when reconfiguring MongoDB member, which is normally recommended only for special scenarios such as crash recovery
- K8SPSMDB-730 Fix a bug due to which point-in-time recovery was enabled and consequently disabled when setting Percona Backup for MongoDB compression options without checking whether it was enabled in the Custom Resource
- K8SPSMDB-660 Fix a bug due to which a successful backup could be erroneously marked as failed due to exceeding the start deadline in case of big number of nodes, especially on sharded clusters
- K8SPSMDB-686 Fix a bug that prevented downscaling sharded MongoDB cluster to a non-sharded replica set variant
- K8SPSMDB-691 Fix a bug that produced an error in the Operator log in case of the empty SSL Secret name in Custom Resource

- K8SPSMDB-696 Fix a bug that prevented removing additional annotations previously added under the `spec.replsets.annotations` field
- K8SPSMDB-724 Fix a bug which caused the delete-backup finalizer not working causing backups being not deleted from buckets
- K8SPSMDB-746 Fix a bug due to which the Operator was unable to initialize a three-member replica set with a primary-secondary-arbiter (PSA) architecture
- K8SPSMDB-762 Fix a bug due to which the Operator was running the replSetReconfig MongoDB command at every reconciliation if arbiter was enabled

Deprecation, Rename and Removal

- K8SPSMDB-690 CCustom Resource options under the sharding.mongos.auditLog subsection, deprecated since the Operator version 1.9.0 in favor of using replsets.configuration, were finally removed and cannot be used with the Operator
- K8SPSMDB-709 Password-based authorization to Percona Monitoring and Management Server is now deprecated and will be removed in future releases in favor of a token-based one. Password-based authorization was used by the Operator before this release to provide MongoDB monitoring, but now using the API Key is the recommended authorization method

Supported Platforms

The Operator was developed and tested with Percona Server for MongoDB 4.2.22, 4.4.8, 4.4.10, 4.4.13, 4.4.16, 5.0.2, 5.0.4, and 5.0.11. Other options may also work but have not been tested.

The following platforms were tested and are officially supported by the Operator 1.13.0:

- Google Kubernetes Engine (GKE) 1.21 - 1.23
- Amazon Elastic Container Service for Kubernetes (EKS) 1.21 - 1.23
- OpenShift Container Platform 4.10 - 4.11
- Azure Kubernetes Service (AKS) 1.22 - 1.24
- Minikube 1.26

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 the backward compatibility offered by Kubernetes itself.

Percona Operator for MongoDB 1.12.0

- **Date**

  May 5, 2022
- **Installation**

  Installing Percona Operator for MongoDB

Release Highlights

- With this release, the Operator turns to a simplified naming convention and changes its official name to **Percona Operator for MongoDB**
- The Operator is able now to use the Amazon Web Services feature of authenticating applications running on EC2 instances based on Identity and Access Management (IAM) roles assigned to the instance; this makes it possible to configure S3 backup on AWS without using IAM keys saved in Secrets
- This release brings support for the Multi Cluster Services (MCS). This allows users to deploy MongoDB with Percona Operator across multiple Kubernetes clusters using MCS, which extends the reach of the Service object beyond one cluster, so one Service can be used across multiple clusters. It can be used to provide disaster recovery or perform a migration for MongoDB clusters.
- The OpenAPI schema is now generated for the Operator , which allows Kubernetes to perform Custom Resource validation and saves user from occasionally applying `deploy/cr.yaml` with syntax typos

New Features

- K8SPSMDB-185: Allow using AWS EC2 instances for backups with IAM roles assigned to the instance instead of using stored IAM credentials (Thanks to Oleksii for reporting this issue)
- K8SPSMDB-625: Integrate the Operator with Multi Cluster Services (MCS)
- K8SPSMDB-668: Adding support for enabling replication over a service mesh (Thanks to Jo Lyshoel for contribution)

Improvements

- K8SPSMDB-473: Allow to skip TLS verification for backup storage, useful for self-hosted S3-compatible storage with a self-issued certificate
- K8SPSMDB-644: Make `cacheSizeRatio` parameter available as a custom value in psmdb-db-1.11.0 helm chart (Thanks to Richard CARRE for reporting this issue)
- K8SPSMDB-574: Allow user to choose the validity duration of the external certificate for cert manager
- K8SPSMDB-634: Support point-in-time recovery compression levels for backups (Thanks to Damiano Albani for reporting this issue)
- K8SPSMDB-570: The Operator documentation now includes a How-To on using Percona Server for MongoDB with LDAP authentication and authorization
- K8SPSMDB-537: PMM container does not cause the crash of the whole database Pod if pmm-agent is not working properly
- K8SPSMDB-684: Generate OpenAPI schema for and validate Custom Resource

Bugs Fixed

- K8SPSMDB-597: Fix a bug in the Operator helm chart which caused deleting the watched Namespace on uninstall (Thanks to Andrei Nistor for reporting this issue)
- K8SPSMDB-640: Fix a regression which prevented labels from being applied to Pods after the Custom Resource change
- K8SPSMDB-583: Fix a bug which caused backup crashing if `spec.mongod.net.port` not set or set to zero
- K8SPSMDB-540 and K8SPSMDB-563: Fix a bug which could cause a cluster crash when reducing the configured Replicaset size between deletion and re-creation of the cluster
- K8SPSMDB-608: Fix a bug due to which the password of backup user was printed in backup agent logs (Thanks to Antoine Ozenne for reporting this issue)
- K8SPSMDB-599: A new mongos.expose.servicePerPod option allows deploying a separate ClusterIP Service for each mongos instance, which prevents the failure of a multi-threaded transaction executed with the same driver instance and ended up on a different mongos. Starting from this release, mongos is deployed by StatefulSet instead of Deployment object
- K8SPSMDB-656: Fix a bug which caused cluster name being not displayed in the backup Custom Resource output with `psmdbCluster` set in the backup spec

- K8SPSMDB-653: Fix a bug due to which `spec.ImagePullPolicy` options from `deploy/cr.yaml` wasn't applied to backup and pmm-client images
- K8SPSMDB-632: Fix a bug which caused the Operator to perform Smart Update on the initial deployment
- K8SPSMDB-624: Fix a bug due to which the Operator didn't grant enough permissions to the Cluster Monitor user necessary for Percona Monitoring and Management (PMM) (Thanks to Richard CARRE for reporting this issue)
- K8SPSMDB-618: Improve security and meet compliance requirements by building MongoDB Operator based on Red Hat Universal Base Image (UBI) 8 instead of UBI 7
- K8SPSMDB-602: Fix a thread leak in a mongod container of the Replica Set Pods, which occurred when setting `setFCV` flag to `true` in Custom Resource
- K8SPSMDB-560: Fix a bug due to which `serviceName` tag was not set to all members in the Replica Set
- K8SPSMDB-533: Fix a bug due to which setting password with a special character for a system user was breaking the cluster

Known Issues

- K8SPSMDB-686: The Operator versions 1.11.0 and 1.12.0 can not be downscaled from a sharding to non-sharding/Replica Set configuration on Google Kubernetes Engine (GKE) 1.19-1.21 (GKE 1.22 is not affected)

Deprecation, Rename and Removal

- K8SPSMDB-596: The `spec.mongod` section is removed from the Custom Resource configuration. Starting from now, mongod options should be passed to Replica Sets using `spec.replsets.[].configuration` key, except the following 3 options:
  - `mongod.security.encryptionKeySecret` key was left in a deprecated state in favor of the new `spec.secrets.encryptionKey` option
  - `mongod.storage.wiredTiger.engineConfig.cacheSizeRatio` and `mongod.storage.inMemory.engineConfig.inMemorySizeRatio` options are now only available from the `replsets.storage` section

Before the upgrade, please ensure that you have moved all custom MongoDB parameters to proper places!

- K8SPSMDB-228: The `spec.psmdbCluster` option in the example on-demand backup configuration file `backup/backup.yaml` was renamed to `spec.clusterName` (`psmdbCluster` will be valid till 1.15 version)

Supported Platforms

The following platforms were tested and are officially supported by the Operator 1.12.0:

- OpenShift 4.7 - 4.10
- Google Kubernetes Engine (GKE) 1.19 - 1.22
- Amazon Elastic Container Service for Kubernetes (EKS) 1.19 - 1.22
- Minikube 1.23

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 the backward compatibility offered by Kubernetes itself.

Percona Distribution for MongoDB Operator 1.11.0

- **Date**

  December 21, 2021

- **Installation**

  For installation please refer to [the documentation page]

Release Highlights

- In addition to S3-compatible storage, you can now configure backups to use Microsoft Azure Blob storage. This feature makes the Operator fully compatible with Azure Cloud.
- Custom sidecar containers allow users to customize Percona Distribution for MongoDB and other Operator components without changing the container images. In this release, we enable even more customization, by allowing users to mount volumes into the sidecar containers.

New Features

- K8SPSMDB-513: Add support of Microsoft Azure Blob storage for backups

Improvements

- K8SPSMDB-422: It is now possible to set annotations to backup cron jobs (Thanks to Aliaksandr Karavai for contribution)
- K8SPSMDB-534: mongos readiness probe now avoids running listDatabases command for all databases in the cluster to avoid unneeded delays on clusters with an extremely large amount of databases
- K8SPSMDB-527: Timeout parameters for liveness and readiness probes can be customized to avoid false-positives for heavy-loaded clusters
- K8SPSMDB-520: Mount volumes into sidecar containers to enable customization
- K8SPSMDB-463: Update backup status as error if it's not started for a long time
- K8SPSMDB-388: New `backup.pitr.oplogSpanMin` option controls how often oplogs are uploaded to the cloud storage

Bugs Fixed

- K8SPSMDB-603: Fixed a bug where the Operator checked the presence of CPU limit and not memory limit when deciding whether to set the size of cache memory for WiredTiger
- K8SPSMDB-511 and K8SPSMDB-558: Fixed a bug where Operator changed NodePort port every 20 seconds for a Replica Set service (Thanks to Rajshekar Reddy for reporting this issue)
- K8SPSMDB-608: Fix a bug that resulted in printing the password of backup user the in backup agent logs (Thanks to Antoine Ozenne for reporting this issue)
- K8SPSMDB-592: Fixed a bug where helm chart was incorrectly setting the `serviceAnnotations` and `loadBalancerSourceRanges` for mongos exposure
- K8SPSMDB-568: Fixed a bug where upgrading to MongoDB 5.0 failed when using the `upgradeOptions:apply` option

Supported Platforms

The following platforms were tested and are officially supported by the Operator 1.11.0:

- OpenShift 4.7 - 4.9
- Google Kubernetes Engine (GKE) 1.19 - 1.22
- Amazon Elastic Container Service for Kubernetes (EKS) 1.18 - 1.22
- Minikube 1.22

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 the backward compatibility offered by Kubernetes itself.

Percona Distribution for MongoDB Operator 1.10.0

- **Date**

  September 30, 2021

- **Installation**

  For installation please refer to the documentation page

Release Highlights

- Starting from this release, the Operator implements as a technical preview the possibility to include non-voting replica set members into the cluster, which do not participate in the primary election process. This feature enables users to deploy non-voting members with the Operator through a Custom Resource object without manual configuration.
- The technical preview of the cross-site replication feature allows users to add external replica set nodes into the cluster managed by the Operator, including scenarios when one of the clusters is outside of the Kubernetes environment. External nodes can be run by another Operator or can be regular MongoDB deployment. The feature is intended for the following use cases:
  - provide migrations of your regular MongoDB database to the Percona Server for MongoDB cluster under the Operator control, or carry on backward migration,
  - deploy cross-regional clusters for Disaster Recovery.

New Features

- K8SPSMDB-479: Allow users to add non-voting members to MongoDB replica, needed to have more than 7 nodes or to create a node in the edge location
- K8SPSMDB-265: Cross region replication feature simplifies the migrations and enables Disaster Recovery capabilities for MongoDB on Kubernetes

Improvements

- K8SPSMDB-537: PMM container should not cause the crash of the whole database Pod if pmm-agent is not working properly
- K8SPSMDB-517: Users can now run Percona Server for MongoDB 5 with the Operator. Version 5 support is added as a technical preview and is not recommended for Production.
- K8SPSMDB-490: Add validation for the Custom Resource name so that cluster name and replica set name do not exceed 51 characters in total

Bugs Fixed

- K8SPSMDB-504: Fixed a race condition that could prevent the cluster with LoadBalancer-exposed replica set members from becoming ready
- K8SPSMDB-470: Fix a bug where ServiceAnnotation and LoadBalancerSourceRanges fields didn't propagate to Kubernetes service (Thanks to Aliaksandr Karavai for reporting this issue)
- K8SPSMDB-531: Fix compatibility issues between Percona Kubernetes Operator for MongoDB and Calico (Thanks to Mykola Kruliv for reporting this issue)
- K8SPSMDB-514: Fix a bug where backup cronJob created by the Operator did not include resources limits and requests, which prevented it to run in the namespaces with resource quotas (Thanks to George Asenov for reporting this issue)
- K8SPSMDB-512: Fix a bug where configuring getLastErrorModes in the replica set causes the Operator to fail to reconcile (Thanks to Adam Watson for contribution)
- K8SPSMDB-553: Fix a bug where wrong S3 credentials caused backup to keep running despite the actual failure
- K8SPSMDB-496: Fix a bug where Pods did not restart if custom MongoDB config was updated with a secret or a configmap

Supported Platforms

The following platforms were tested and are officially supported by the Operator 1.10.0:

- OpenShift 4.6 - 4.8

- Google Kubernetes Engine (GKE) 1.17 - 1.21
- Amazon Elastic Container Service for Kubernetes (EKS) 1.16 - 1.21
- Minikube 1.22

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 the backward compatibility offered by Kubernetes itself.

Percona Distribution for MongoDB Operator 1.9.0

- **Date**

  June 29, 2021

- **Installation**

  For installation please refer to the documentation page

Release Highlights

- Starting from this release, the Operator changes its official name to **Percona Distribution for MongoDB Operator**. This new name emphasizes graduate changes which incorporated a collection of Percona's solutions to run and operate MongoDB Server, available separately as Percona Distribution for MongoDB.
- It is now possible to restore backups from S3-compatible storage to a new Kubernetes-based environment with no existing Backup Custom Resources
- You can now customize Percona Server for MongoDB by storing custom configuration for Replica Set, mongos, and Config Server instances in ConfigMaps or in Secrets

New Features

- K8SPSMDB-276: Restore backups to a new Kubernetes-based environment with no existing Backup Custom Resource
- K8SPSMDB-444, K8SPSMDB-445: Allow storing custom configuration in ConfigMaps and Secrets

Improvements

- K8SPSMDB-365: Unblock backups even if just a single Replica Set node is available by setting `allowUnsafeConfigurations` flag to true
- K8SPSMDB-453: It is now possible to see the overall progress of the provisioning of MongoDB cluster resources and dependent components in Custom Resource status
- K8SPSMDB-451, K8SPSMDB-398: MongoDB cluster resource statuses in Custom Resource output (e.g. returned by `kubectl get psmdb` command) have been improved and now provide more precise reporting
- K8SPSMDB-425: Remove `mongos.expose.enabled` option from Custom Resource and always expose mongos (with the ClusterIP exposeType by default)
- K8SPSMDB-421: Secret object containing system users passwords is now deleted along with the Cluster if delete-psmdb-pvc finalizer is enabled
- K8SPSMDB-411: Added options to specify custom memory and CPU requirements for Arbiter instances
- K8SPSMDB-329: Reduced the number of various etcd and k8s object updates from the operator to minimize the pressure on the Kubernetes cluster

Bugs Fixed

- K8SPSMDB-437: Fixed a bug where Labels were not set on Persistent Volume Claim objects when set on the respective Pods
- K8SPSMDB-435: Fixed a bug that prevented adding custom Labels to mongos Pods
- K8SPSMDB-423: Fixed a bug where unpause of a cluster did not work when `replsets.expose = LoadBalancer` because of provisioning new Load Balancers with different names (Thanks to Aliaksandr Karavai for reporting this issue)
- K8SPSMDB-494: When upgrading MongoDB clusters with Smart Update, the statuses reported in Custom Resource are now reflecting the real state
- K8SPSMDB-489: Fixed a bug where the status of successful backups could be set to error in case of a cluster crash
- K8SPSMDB-462: Fixed a bug where psmdb-backup object could not be deleted if the backup was not successful
- K8SPSMDB-456: Fixed a bug where Smart Update was not upgrading a MongoDB deployment with a replica set consisting of one node
- K8SPSMDB-455: Fixed a bug that prevented major version downgrade to a specific version number when `upgradeOptions.setFCV` Custom Resource option was not updated to the new version
- K8SPSMDB-485: Fixed TLS documentation that referenced incorrect Secrets names from the cr.yaml configuration file

Deprecation and Removal

- We are simplifying the way the user can customize MongoDB components such as mongod and mongos. It is now possible to set custom configuration through ConfigMaps and Secrets Kubernetes resources. The following options will be deprecated in Percona Distribution for MongoDB Operator v1.9.0+, and completely removed in v1.12.0+:
    - `sharding.mongos.auditLog.\*`
    - `mongod.security.redactClientLogData`
    - `mongod.security.\*`
    - `mongod.setParameter.\*`
    - `mongod.storage.\*`
    - `mongod.operationProfiling.mode`
    - `mongod.auditLog.\*`
- The mongos.expose.enabled option has been completely removed from the Custom Resource as it was causing confusion for the users

Percona Kubernetes Operator for Percona Server for MongoDB 1.8.0

- **Date**

  May 6, 2021

- **Installation**

  Installing Percona Kubernetes Operator for Percona Server for MongoDB

Release Highlights

- The support for Point-in-time recovery added in this release. Users can now recover to a specific date and time from operations logs stored on S3
- It is now possible to perform a major version upgrade for MongoDB (for example, upgrade 4.2 version to 4.4) with no manual steps

New Features

- K8SPSMDB-387: Add support for point-in-time recovery to recover to a specific date and time
- K8SPSMDB-284: Add support for automated major version MongoDB upgrades

Improvements

- K8SPSMDB-436: The imagePullPolicy option in the `deploy/cr.yaml` configuration file now is applied to init container as well
- K8SPSMDB-400: Simplify secret change logic to avoid Pod restarts when user changes the credentials
- K8SPSMDB-381: Get credentials directly from Secrets instead of the environment variables when initializing the Replica Set
- K8SPSMDB-352: Restrict running run less than 5 Pods of Replica Sets with enabled arbiter unless the `allowUnsafeConfigurations` option is set to true
- K8SPSMDB-332: Restrict running less than 3 Pods of Config Servers unless the `allowUnsafeConfigurations` option is set to true
- K8SPSMDB-331: Restrict running less than 3 mongos Pods unless the `allowUnsafeConfigurations` option is set to true

Bugs Fixed

- K8SPSMDB-384: Fix a bug due to which mongos Pods were failing readiness probes for some period of time during the cluster initialization
- K8SPSMDB-434: Fix a bug due to which nil pointer dereference error was occurring when switching the `sharding.enabled` option from false to true (thanks to srteam2020 for contributing)
- K8SPSMDB-430: Fix a bug due to which a stale apiserver could trigger undesired StatefulSet and PVC deletion when recreating the cluster with the same name (thanks to srteam2020 for contributing)
- K8SPSMDB-428: Fix a bug which caused mongos to fail in case of the empty name field in configsvrReplSet section of the Custom Resource
- K8SPSMDB-418: Fix a bug due to which `serviceAnnotations` changes in the `deploy/cr.yaml` file were not applied to the running cluster
- K8SPSMDB-364: Fix a bug where liveness probe of a mongo container was always failing if the userAdmin password contained special characters
- K8SPSMDB-43: Fix a bug due to which renaming Replica Set in the Custom Resource caused creating new Replica Set without deleting the old one

Percona Kubernetes Operator for Percona Server for MongoDB 1.7.0

- **Date**

  March 8, 2021

- **Installation**

  [Installing Percona Kubernetes Operator for Percona Server for MongoDB]

Release Highlights

- This release brings full support for the Percona Server for MongoDB Sharding. Sharding allows you to scale databases horizontally, distributing data across multiple MongoDB Pods, and so it is extremely useful for large data sets. By default of the `deploy/cr.yaml` configuration file contains only one replica set, but when you turn sharding on, you can add more replica sets with different names to the `replsets` section.
- It is now possible to clean up Persistent Volume Claims automatically after the cluster deletion event. This feature is off by default. Particularly it is useful to avoid leftovers in testing environments, where the cluster can be re-created and deleted many times. Support for custom sidecar containers. The Operator makes it possible now to deploy additional (*sidecar*) containers to the Pod. This feature can be useful to run debugging tools or some specific monitoring solutions, etc. The sidecar container can be added to replsets, sharding.configsvrReplSet, and sharding.mongos sections of the `deploy/cr.yaml` configuration file.

New Features

- K8SPSMDB-121: Add support for sharding to scale MongoDB cluster horizontally
- K8SPSMDB-294: Support for custom sidecar container to extend the Operator capabilities
- K8SPSMDB-260: Persistent Volume Claims can now be automatically removed after MongoDB cluster deletion

Improvements

- K8SPSMDB-335: Operator can now automatically remove old backups from S3 if retention period is set
- K8SPSMDB-330: Add support for runtimeClassName Kubernetes feature for selecting the container runtime
- K8SPSMDB-306: It is now possible to explicitly set the version of MongoDB for newly provisioned clusters. Before that, all new clusters were started with the latest MongoDB version if Version Service was enabled
- K8SPSMDB-370: Fix confusing log messages about no backup / restore found which were caused by Percona Backup for MongoDB waiting for the backup metadata
- K8SPSMDB-342: MongoDB container liveness probe will now use TLS to follow best practices and remove noisy log messages from mongod log

Bugs Fixed

- K8SPSMDB-346: Fix a bug which prevented adding/removing labels to Pods without downtime
- K8SPSMDB-366: Fix a bug which prevented enabling Percona Monitoring and Management (PMM) due to incorrect request for the recommended PMM Client image version to the Version Service
- K8SPSMDB-402: running multiple replica sets without sharding enabled should be prohibited
- K8SPSMDB-382: Fix a bug which caused mongos process to fail when using `allowUnsafeConfigurations=true`
- K8SPSMDB-362: Fix a bug due to which changing secrets in a single-shard mode caused mongos Pods to fail

Percona Kubernetes Operator for Percona Server for MongoDB 1.6.0

- **Date**

  December 22, 2020

- **Installation**

  Installing Percona Kubernetes Operator for Percona Server for MongoDB

New Features

- K8SPSMDB-273: Add support for `mongos` service to expose a single shard of a MongoDB cluster through one entry point instead of provisioning a load-balancer per replica set node. In the following release, we will add support for multiple shards.
- K8SPSMDB-282: Official support for Percona Monitoring and Management (PMM) v.2

> **Note**
>
> Monitoring with PMM v.1 configured according to the unofficial instruction will not work after the upgrade. Please switch to PMM v.2.

Improvements

- K8SPSMDB-258: Add support for Percona Server for MongoDB version 4.4
- K8SPSMDB-319: Show Endpoint in the `kubectl get psmdb` command output to connect to a MongoDB cluster easily
- K8SPSMDB-257: Store the Operator version as a `crVersion` field in the `deploy/cr.yaml` configuration file
- K8SPSMDB-266: Use plain-text passwords instead of base64-encoded ones when creating System Users secrets for simplicity

Bugs Fixed

- K8SPSMDB-268: Fix a bug affecting the support of TLS certificates issued by cert-manager, due to which proper rights were not set for the role-based access control, and Kubernetes versions newer than 1.15 required other certificate issuing sources
- K8SPSMDB-261: Fix a bug due to which cluster pause/resume functionality didn't work in previous releases
- K8SPSMDB-292: Fix a bug due to which not all clusters managed by the Operator were upgraded by the automatic update

Removal

- The MMAPv1 storage engine is no longer supported for all MongoDB versions starting from this version of the Operator. MMAPv1 was already deprecated by MongoDB for a long time. WiredTiger is the default storage engine since MongoDB 3.2, and MMAPv1 was completely removed in MongoDB 4.2.

> **Note**
>
> Upgrade of the Operator from 1.5.0 to 1.6.0 will fail if MMAPv1 is used, but MongoDB cluster will continue to run. It is recommended to migrate your clusters to WiredTiger engine before the upgrade.

Percona Kubernetes Operator for Percona Server for MongoDB 1.5.0

- **Date**

  September 7, 2020

- **Installation**

  Installing Percona Kubernetes Operator for Percona Server for MongoDB

New Features

- K8SPSMDB-233: Automatic management of system users for MongoDB on password rotation via Secret
- K8SPSMDB-226: Official Helm chart for the Operator
- K8SPSMDB-199: Support multiple PSMDB minor versions by the Operator
- K8SPSMDB-198: Fully Automate Minor Version Updates (Smart Update)

Improvements

- K8SPSMDB-192: The ability to set the mongod cursorTimeoutMillis parameter in YAML (Thanks to user xprt64 for the contribution)
- K8SPSMDB-234: OpenShift 4.5 support
- K8SPSMDB-197: Additional certificate SANs useful for reverse DNS lookups (Thanks to user phin1x for the contribution)
- K8SPSMDB-190: Direct API quering with "curl" instead of using "kubectl" tool in scheduled backup jobs (Thanks to user phin1x for the contribution)
- K8SPSMDB-133: A special Percona Server for MongoDB debug image which avoids restarting on fail and contains additional tools useful for debugging
- CLOUD-556: Kubernetes 1.17 / Google Kubernetes Engine 1.17 support

Bugs Fixed

- K8SPSMDB-213: Installation instruction not reflecting recent changes in git tags (Thanks to user geraintj for reporting this issue)
- K8SPSMDB-210: Backup documentation not reflecting changes in Percona Backup for MongoDB
- K8SPSMDB-180: Replset and cluster having "ready" status set before mongo initialization and replicasets configuration finished
- K8SPSMDB-179: The "error" cluster status instead of the "initializing" one during the replset initialization
- CLOUD-531: Wrong usage of `strings.TrimLeft` when processing apiVersion

Percona Kubernetes Operator for Percona Server for MongoDB 1.4.0

- **Date**

  March 31, 2020

- **Installation**

  Installing Percona Kubernetes Operator for PSMDB

New Features

- K8SPSMDB-89: Amazon Elastic Container Service for Kubernetes (EKS) was added to the list of the officially supported platforms
- K8SPSMDB-113: Percona Server for MongoDB 4.2 is now supported
- OpenShift Container Platform 4.3 is now supported

Improvements

- K8SPSMDB-79: The health check algorithm improvements have increased the overall stability of the Operator
- K8SPSMDB-176: The Operator was updated to use Percona Backup for MongoDB version 1.2
- K8SPSMDB-153: Now the user can adjust securityContext, replacing the automatically generated securityContext with the customized one
- K8SPSMDB-175: Operator now updates observedGeneration status message to allow better monitoring of the cluster rollout or backups/restore process

Bugs Fixed

- K8SPSMDB-182: Setting the `updateStrategy: OnDelete` didn't work if was not specified from scratch in CR
- K8SPSMDB-174: The inability to update or delete existing CRD was possible because of too large records in etcd, resulting in "request is too large" errors. Only 20 last status changes are now stored in etcd to avoid this problem.

Help us improve our software quality by reporting any bugs you encounter using our bug tracking system.

Percona Kubernetes Operator for Percona Server for MongoDB 1.3.0

Percona announces the *Percona Kubernetes Operator for Percona Server for MongoDB* 1.3.0 release on December 11, 2019. This release is now the current GA release in the 1.3 series. Install the Kubernetes Operator for Percona Server for MongoDB by following the instructions.

The Operator simplifies the deployment and management of the Percona Server for MongoDB in Kubernetes-based environments. It extends the Kubernetes API with a new custom resource for deploying, configuring and managing the application through the whole life cycle.

The Operator source code is available in our Github repository. All of Percona's software is open-source and free.

New Features and Improvements

- CLOUD-415: Non-default cluster domain can now be specified with the new `ClusterServiceDNSSuffix` Operator option.
- CLOUD-395: The Percona Server for MongoDB images size decrease by 42% was achieved by removing unnecessary dependencies and modules to reduce the cluster deployment time.
- CLOUD-390: Helm chart for Percona Monitoring and Management (PMM) 2.0 have been provided.

Percona Server for MongoDB is an enhanced, open source and highly-scalable database that is a fully-compatible, drop-in replacement for MongoDB Community Edition. It supports MongoDB protocols and drivers. Percona Server for MongoDB extends MongoDB Community Edition functionality by including the Percona Memory Engine, as well as several enterprise-grade features. It requires no changes to MongoDB applications or code.

Help us improve our software quality by reporting any bugs you encounter using our bug tracking system.

Percona Kubernetes Operator for Percona Server for MongoDB 1.2.0

Percona announces the *Percona Kubernetes Operator for Percona Server for MongoDB* 1.2.0 release on September 20, 2019. This release is now the current GA release in the 1.2 series. Install the Kubernetes Operator for Percona Server for MongoDB by following the instructions.

The Operator simplifies the deployment and management of the Percona Server for MongoDB in Kubernetes-based environments. It extends the Kubernetes API with a new custom resource for deploying, configuring and managing the application through the whole life cycle.

The Operator source code is available in our Github repository. All of Percona's software is open-source and free.

New Features and Improvements

- A Service Broker was implemented for the Operator, allowing a user to deploy Percona XtraDB Cluster on the OpenShift Platform, configuring it with a standard GUI, following the Open Service Broker API.
- Now the Operator supports Percona Monitoring and Management 2, which means being able to detect and register to PMM Server of both 1.x and 2.0 versions.
- Data-at-rest encryption is now enabled by default unless `EnableEncryption=false` is explicitly specified in the `deploy/cr.yaml` configuration file.
- Now it is possible to set the `schedulerName` option in the operator parameters. This allows using storage which depends on a custom scheduler, or a cloud provider which optimizes scheduling to run workloads in a cost-effective way.
- The resource constraint values were refined for all containers to eliminate the possibility of an out of memory error.

Fixed Bugs

- Oscillations of the cluster status between "initializing" and "ready" took place after an update.
- The Operator was removing other cron jobs in case of the enabled backups without defined tasks (contributed by Marcel Heers).

Percona Server for MongoDB is an enhanced, open source and highly-scalable database that is a fully-compatible, drop-in replacement for MongoDB Community Edition. It supports MongoDB protocols and drivers. Percona Server for MongoDB extends MongoDB Community Edition functionality by including the Percona Memory Engine, as well as several enterprise-grade features. It requires no changes to MongoDB applications or code.

Help us improve our software quality by reporting any bugs you encounter using our bug tracking system.

Percona Kubernetes Operator for Percona Server for MongoDB 1.1.0

Percona announces the general availability of *Percona Kubernetes Operator for Percona Server for MongoDB* 1.1.0 on July 15, 2019. This release is now the current GA release in the 1.1 series. [Install the Kubernetes Operator for Percona Server for MongoDB by following the instructions](). Please see the [GA release announcement]().

The Operator simplifies the deployment and management of the [Percona Server for MongoDB]() in Kubernetes-based environments. It extends the Kubernetes API with a new custom resource for deploying, configuring and managing the application through the whole life cycle.

The Operator source code is available [in our Github repository](). All of Percona's software is open-source and free.

New Features and Improvements

- Now the Percona Kubernetes Operator [allows upgrading]() Percona Server for MongoDB to newer versions, either in semi-automatic or in manual mode.
- Also, two modes are implemented for updating the Percona Server for MongoDB `mongod.conf` configuration file: in *automatic configuration update* mode Percona Server for MongoDB Pods are immediately re-created to populate changed options from the Operator YAML file, while in *manual mode* changes are held until Percona Server for MongoDB Pods are re-created manually.
- [Percona Server for MongoDB data-at-rest encryption]() is now supported by the Operator to ensure that encrypted data files cannot be decrypted by anyone except those with the decryption key.
- A separate service account is now used by the Operator's containers which need special privileges, and all other Pods run on default service account with limited permissions.
- [User secrets]() are now generated automatically if don't exist: this feature especially helps reduce work in repeated development environment testing and reduces the chance of accidentally pushing predefined development passwords to production environments.
- The Operator [is now able to generate TLS certificates itself]() which removes the need in manual certificate generation.
- The list of officially supported platforms now includes the [Minikube](), which provides an easy way to test the Operator locally on your own machine before deploying it on a cloud.
- Also, Google Kubernetes Engine 1.14 and OpenShift Platform 4.1 are now supported.

[Percona Server for MongoDB]() is an enhanced, open source and highly-scalable database that is a fully-compatible, drop-in replacement for MongoDB Community Edition. It supports MongoDB protocols and drivers. Percona Server for MongoDB extends MongoDB Community Edition functionality by including the Percona Memory Engine, as well as several enterprise-grade features. It requires no changes to MongoDB applications or code.

Help us improve our software quality by reporting any bugs you encounter using [our bug tracking system]().

Percona Kubernetes Operator for Percona Server for MongoDB 1.0.0

Percona announces the general availability of *Percona Kubernetes Operator for Percona Server for MongoDB* 1.0.0 on May 29, 2019. This release is now the current GA release in the 1.0 series. Install the Kubernetes Operator for Percona Server for MongoDB by following the instructions. Please see the GA release announcement. All of Percona's software is open-source and free.

The Percona Kubernetes Operator for Percona Server for MongoDB automates the lifecycle of your Percona Server for MongoDB environment. The Operator can be used to create a Percona Server for MongoDB replica set, or scale an existing replica set.

The Operator creates a Percona Server for MongoDB replica set with the needed settings and provides a consistent Percona Server for MongoDB instance. The Percona Kubernetes Operators are based on best practices for configuration and setup of the Percona Server for MongoDB.

The Kubernetes Operators provide a consistent way to package, deploy, manage, and perform a backup and a restore for a Kubernetes application. Operators deliver automation advantages in cloud-native applications and may save time while providing a consistent environment.

The advantages are the following:

- Deploy a Percona Server for MongoDB environment with no single point of failure and environment can span multiple availability zones (AZs).
- Deployment takes about six minutes with the default configuration.
- Modify the Percona Server for MongoDB size parameter to add or remove Percona Server for MongoDB replica set members
- Integrate with Percona Monitoring and Management (PMM) to seamlessly monitor your Percona Server for MongoDB
- Automate backups or perform on-demand backups as needed with support for performing an automatic restore
- Supports using Cloud storage with S3-compatible APIs for backups
- Automate the recovery from failure of a Percona Server for MongoDB replica set member
- TLS is enabled by default for replication and client traffic using Cert-Manager
- Access private registries to enhance security
- Supports advanced Kubernetes features such as pod disruption budgets, node selector, constraints, tolerations, priority classes, and affinity/anti-affinity
- You can use either PersistentVolumeClaims or local storage with hostPath to store your database
- Supports a replica set Arbiter member
- Supports Percona Server for MongoDB versions 3.6 and 4.0

Installation

Installation is performed by following the documentation installation instructions for Kubernetes and OpenShift.