# Percona Operator for PostgreSQL documentation

2.2.0 (June 30, 2023)

*Percona Technical Documentation Team*

# Table of contents

# 1. Percona Operator for PostgreSQL

Kubernetes have added a way to manage containerized systems, including database clusters. This management is achieved by controllers, declared in configuration files. These controllers provide automation with the ability to create objects, such as a container or a group of containers called pods, to listen for an specific event and then perform a task.

This automation adds a level of complexity to the container-based architecture and stateful applications, such as a database. 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 PostgreSQL is based on best practices for configuration and setup of a Percona Distribution for PostgreSQL cluster. The benefits of the Operator are many, but saving time and delivering a consistent and vetted environment is key.

# 2. Features

- System Requirements
- Design and architecture
- Comparison with other solutions

# 3. Quickstart

- Install with Helm
- Install with kubectl

# 4. Installation

- Install on Minikube
- Install on Google Kubernetes Engine (GKE)
- Install on Amazon Elastic Kubernetes Service (AWS EKS)
- Generic Kubernetes installation

# 5. Configuration

- Application and system users
- Exposing the cluster
- Changing PostgreSQL Options
- Anti-affinity and tolerations
- Telemetry

### Contact Us

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

---

Last update: 2023-06-30

# 10. Features

## 10.1 System Requirements

The Operator is validated for deployment on Kubernetes, GKE and EKS clusters. The Operator is cloud native and storage agnostic, working with a wide variety of storage classes, hostPath, and NFS.

### 10.1.1 Officially supported platforms

The Operator was developed and tested with PostgreSQL versions 12.14, 13.10, 14.7, and 15.2. Other options may also work but have not been tested. The Operator 2.2.0 provides connection pooling based on pgBouncer 1.18.0 and high-availability implementation based on Patroni 3.0.1.

The following platforms were tested and are officially supported by the Operator 2.2.0:

- Google Kubernetes Engine (GKE) 1.23 - 1.26
- Amazon Elastic Container Service for Kubernetes (EKS) 1.23 - 1.27
- Minikube 1.30.1 (based on Kubernetes 1.27)

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

**CONTACT US**

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

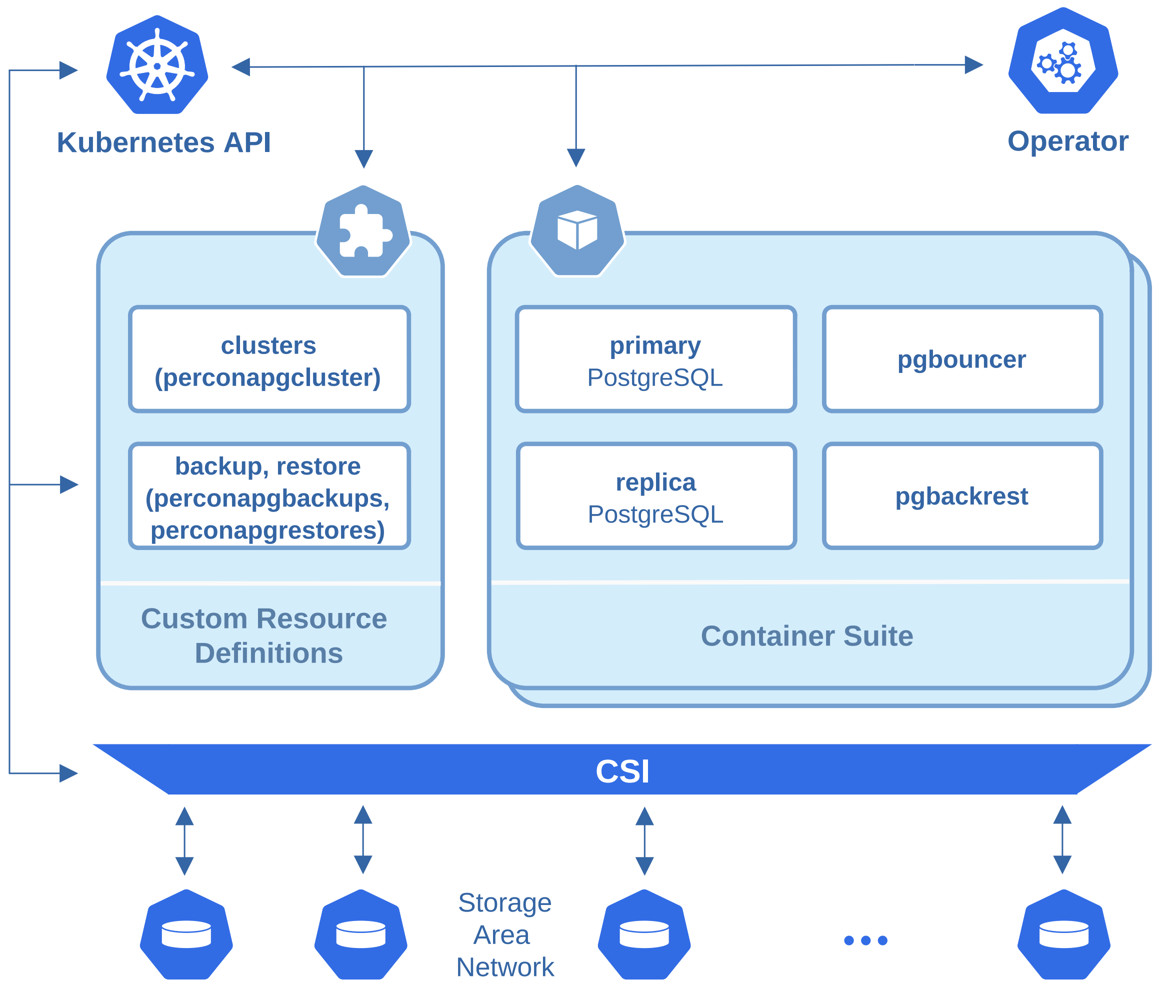
To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-06-30

## 10.2 Design overview

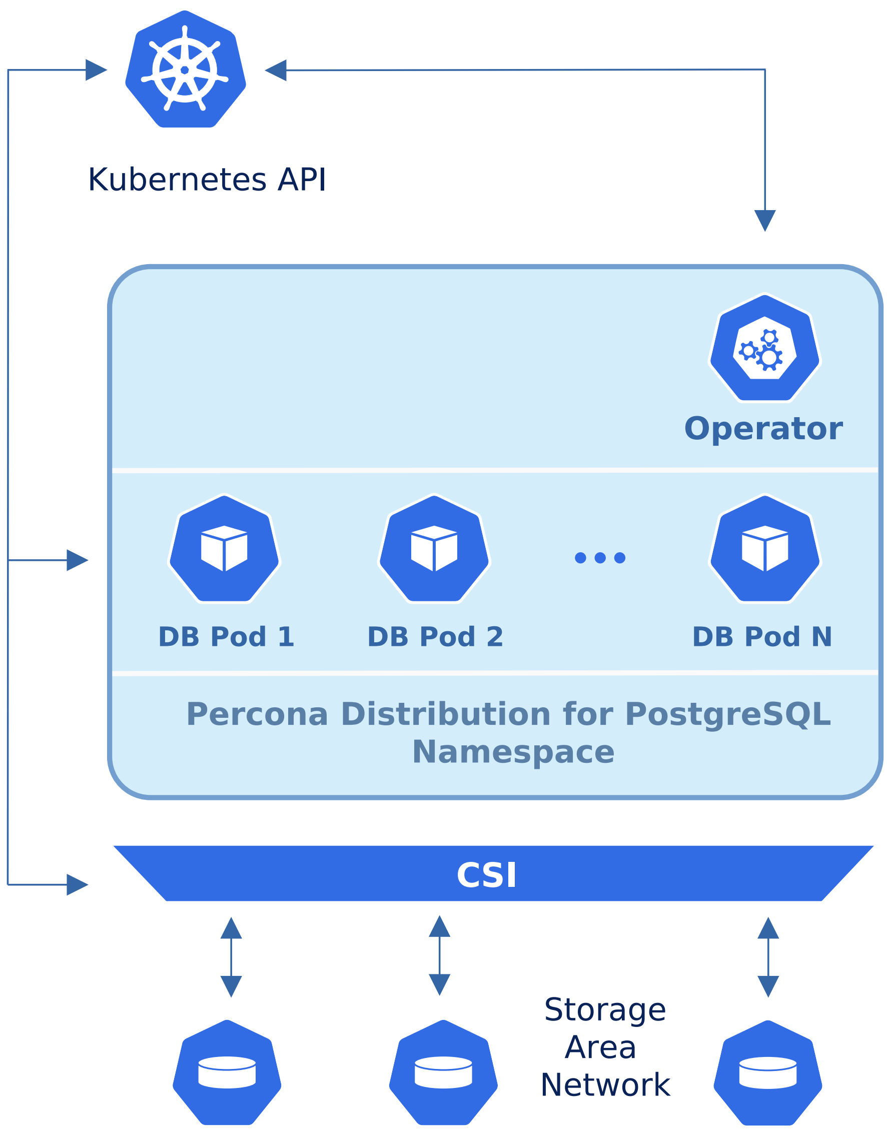
The Percona Operator for PostgreSQL automates and simplifies deploying and managing open source PostgreSQL clusters on Kubernetes. The Operator is based on CrunchyData's PostgreSQL Operator.



PostgreSQL containers deployed with the Operator include the following components:

- The PostgreSQL database management system, including:
  - PostgreSQL Additional Supplied Modules,
  - pgAudit PostgreSQL auditing extension,
  - PostgreSQL set_user Extension Module,
  - wal2json output plugin,
- The pgBackRest Backup & Restore utility,
- The pgBouncer connection pooler for PostgreSQL,
- The PostgreSQL high-availability implementation based on the Patroni template,
- the pg_stat_monitor PostgreSQL Query Performance Monitoring utility,
- LLVM (for JIT compilation).

To provide high availability the Operator involves node affinity to run PostgreSQL Cluster instances on separate worker nodes if possible. If some node fails, the Pod with it is automatically re-created on another node.



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 Operator functionality extends the Kubernetes API with Custom Resources Definitions. These CRDs provide extensions to the Kubernetes API, and, in the case of the Operator, allow you to perform actions such as creating a PostgreSQL Cluster, updating PostgreSQL Cluster resource allocations, adding additional utilities to a PostgreSQL cluster, e.g. pgBouncer for connection pooling and more.

When a new Custom Resource 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 proper Percona PostgreSQL Cluster operation.

Following CRDs are created while the Operator installation:

- `perconapgclusters` stores information required to manage a PostgreSQL cluster. This includes things like the cluster name, what storage and resource classes to use, which version of PostgreSQL to run, information about how to maintain a high-availability cluster, etc.
- `perconapgbackups` and `perconapgrestores` are in charge for making backups and restore them.

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-04-14

## 10.3 Compare various solutions to deploy PostgreSQL in Kubernetes

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

- Crunchy Data PostgreSQL Operator (PGO)
- CloudNative PG from Enterprise DB
- Stackgres from OnGres
- Zalando Postgres Operator
- Percona Operator for PostgreSQL

### 10.3.1 Generic

| **Feature/ Product** | **Percona Operator for PostgreSQL** | **Stackgres** | **CrunchyData** | **CloudNativePG (EDB)** | **Zalando** |
|---|---|---|---|---|---|
| Open-source license | Apache 2.0 | AGPL 3 | Apache 2.0, but images are under Developer Program | Apache 2.0 | MIT |
| PostgreSQL versions | 12 - 15 | 12-15 | 12, 13, 14 | 11 - 15 | 11 - 14 |
| Kubernetes conformance | Various versions are tested | Various versions are tested | Various versions are tested | Various versions are tested | AWS EKS |

### 10.3.2 Maintenance

| **Feature/ Product** | **Percona Operator for PostgreSQL** | **Stackgres** | **CrunchyData** | **CloudNativePG (EDB)** | **Zalando** |
|---|---|---|---|---|---|
| Operator upgrade | ✅ | ✅ | ✅ | ✅ | ✅ |
| Database upgrade | Automated and safe | Automated and safe | Manual | Manual | Manual |
| Compute scaling | Horizontal and vertical | Horizontal and vertical | Horizontal and vertical | Horizontal and vertical | Horizontal and vertical |
| Storage scaling | Manual | Manual | Manual | Manual | Manual, automated for AWS EBS |

### 10.3.3 PostgreSQL topologies

| Feature/ Product | Percona Operator for PostgreSQL | Stackgres | CrunchyData | CloudNativePG (EDB) | Zalando |
|---|---|---|---|---|---|
| Warm standby | ✅ | ✅ | ✅ | ✅ | ✅ |
| Hot standby | ✅ | ✅ | ✅ | ✅ | ✅ |
| Connection pooling | ✅ | ✅ | ✅ | ✅ | ✅ |
| Delayed replica | 🚫 | 🚫 | 🚫 | 🚫 | 🚫 |

### 10.3.4 Backups

| Feature/ Product | Percona Operator for PostgreSQL | Stackgres | CrunchyData | CloudNativePG (EDB) | Zalando |
|---|---|---|---|---|---|
| Scheduled backups | ✅ | ✅ | ✅ | ✅ | ✅ |
| WAL archiving | ✅ | ✅ | ✅ | ✅ | ✅ |
| PITR | ✅ | ✅ | ✅ | ✅ | ✅ |
| GCS | ✅ | ✅ | ✅ | ✅ | ✅ |
| S3 | ✅ | ✅ | ✅ | ✅ | ✅ |
| Azure | ✅ | ✅ | ✅ | ✅ | ✅ |

### 10.3.5 Monitoring

| Feature/ Product | Percona Operator for PostgreSQL | Stackgres | CrunchyData | CloudNativePG (EDB) | Zalando |
|---|---|---|---|---|---|
| Solution | Percona Monitoring and Management and sidecars | Exposing metrics in Prometheus format | Prometheus stack and pgMonitor | Exposing metrics in Prometheus format | Sidecars |

## 10.3.6 Miscellaneous

| Feature/ Product | Percona Operator for PostgreSQL | Stackgres | CrunchyData | CloudNativePG (EDB) | Zalando |
|---|---|---|---|---|---|
| Customize PostgreSQL configuration | ✓ | ✓ | ✓ | ✓ | ✓ |
| Helm | ✓ | ✓ | ✓ | ✓ | ✓ |
| Transport encryption | ✓ | ✓ | ✓ | ✓ | ✓ |
| Data-at-rest encryption | Through storage class | Through storage class | Through storage class | Through storage class | Through storage class |
| Create users/roles | ✓ | ✓ | ✓ | ✓ | limited |

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-05-17

# 11. Quickstart

## 11.1 Install Percona Distribution for PostgreSQL using Helm

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

### 11.1.1 Pre-requisites

Install Helm following its official installation instructions.

**Note**

Helm v3 is needed to run the following steps.

### 11.1.2 Installation

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. Install the Percona Operator for PostgreSQL:

```
$ helm install my-operator percona/pg-operator
```

The `my-operator` 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).

**Note**

If nothing explicitly specified, `helm install` command will work with the `default` namespace and the latest version of the Helm chart.

- To use different namespace, provide its name with the following additional parameter: `--namespace my-namespace`.
- To use different Helm chart version, provide it as follows: `--version 2.2.0`

3. Install PostgreSQL:

```
$ helm install my-db percona/pg-db
```

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

### 11.1.3 Installing Percona Distribution for PostgreSQL with customized parameters

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

The following example will deploy a PostgreSQL 14 based cluster in the `my-namespace` namespace, with enabled Percona Monitoring and Management (PMM):

```
$ helm install my-db percona/pg-db --version 2.2.0 --namespace my-namespace \
  --set postgresVersion=14 \
  --set pmm.enabled=true
```

Last update: 2023-06-30

## 11.2 Install Percona Distribution for PostgreSQL using kubectl

The kubectl command line utility is a tool used before anything else to interact with Kubernetes and containerized applications running on it. Users can run kubectl to deploy applications, manage cluster resources, check logs, etc.

### 11.2.1 Pre-requisites

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

1. The **Git** distributed version control system. You can install it following the official installation instructions.
2. The **kubectl** tool to manage and deploy applications on Kubernetes, included in most Kubernetes distributions. Install it, if not present, following the official installation instructions.

### 11.2.2 Install the Operator and Percona Distribution for PostgreSQL

The following steps are needed to deploy the Operator and Percona Distribution for PostgreSQL in your Kubernetes environment:

1. Create the Kubernetes namespace for your cluster if needed (for example, let's name it `postgres-operator` ):

   ```
   $ kubectl create namespace postgres-operator
   ```

   **Expected output**

   ```
   namespace/postgres-operator was created
   ```

   **Note**

   To use different namespace, specify other name instead of `postgres-operator` in the above command, and modify the `-n postgres-operator` parameter with it in the following two steps. You can also omit this parameter completely to deploy everything in the `default` namespace.

2. Deploy the Operator using the following command:

   ```
   $ kubectl apply --server-side -f https://raw.githubusercontent.com/percona/percona-
   postgresql-operator/v2.2.0/deploy/bundle.yaml -n postgres-operator
   ```

   **Expected output**

   ```
   customresourcedefinition.apiextensions.k8s.io/perconapgbackups.pgv2.percona.com serverside-
   applied
   customresourcedefinition.apiextensions.k8s.io/perconapgclusters.pgv2.percona.com serverside-
   applied
   customresourcedefinition.apiextensions.k8s.io/perconapgrestores.pgv2.percona.com serverside-
   applied
   customresourcedefinition.apiextensions.k8s.io/postgresclusters.postgres-
   operator.crunchydata.com serverside-applied
   serviceaccount/percona-postgresql-operator serverside-applied
   role.rbac.authorization.k8s.io/percona-postgresql-operator serverside-applied
   rolebinding.rbac.authorization.k8s.io/service-account-percona-postgresql-operator serverside-
   applied
   deployment.apps/percona-postgresql-operator serverside-applied
   ```

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

3. The Operator has been started, and you can deploy your Percona Distribution for PostgreSQL cluster:

   ```
   $ kubectl apply -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/
   v2.2.0/deploy/cr.yaml -n postgres-operator
   ```

   **Expected output**

   ```
   perconapgcluster.pgv2.percona.com/cluster1 created
   ```

**Note**

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

```
$ git clone -b v2.2.0 https://github.com/percona/percona-postgresql-operator
```

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

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

The creation process may take some time. The process is over when both Operator and replica set Pods have reached their Running status:

```
$ kubectl get pg
```

**Expected output**

```
NAME       ENDPOINT                                STATUS   POSTGRES   PGBOUNCER   AGE
cluster1   cluster1-pgbouncer.postgres-operator.svc   ready    3          3           143m
```

### 11.2.3 Verifying the cluster operation

When creation process is over, `kubectl get pg` command will show you the cluster status as `ready`, and you can try to connect to the cluster.

1. During the installation, the Operator has generated several secrets, including the one with password for default PostgreSQL user. This default user has the login name same as the the cluster.

   Use `kubectl get secrets` command to see the list of Secrets objects. The Secrets object you are interested in is named as `<cluster_name>-pguser-<cluster_name>` (substitute `<cluster_name>` with the name of your Percona Distribution for PostgreSQL Cluster). The default variant will be `cluster1-pguser-cluster1`. You can use the following command to get the password of this user:

   ```
   $ kubectl get secret <cluster_name>-<user_name>-<cluster_name> --
   template='{{.data.password | base64decode}}{{"\n"}}'
   ```

2. Run a container with `psql` tool and connect its console output to your terminal. The following command will do this, naming the new Pod `pg-client`:

   ```
   $ kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-postgresql:
   15 --restart=Never -- bash -il
   [postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer.postgres-
   operator.svc -p 5432 -U cluster1 cluster1
   ```

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

   This command will connect you as a `cluster1` user to a `cluster1` database via the PostgreSQL interactive terminal.

   ```
   psql (15)
   SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256,
   compression: off)
   Type "help" for help.
   pgdb=>
   ```

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-06-29

# 12. Installation

## 12.1 Install Percona Distribution for PostgreSQL on Minikube

Installing the Percona Operator for PostgreSQL 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 deploy the Operator and Percona Distribution for PostgreSQL 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-
   postgresql-operator/v2.2.0/deploy/bundle.yaml
   ```

   **Expected output**

   ```
   customresourcedefinition.apiextensions.k8s.io/perconapgbackups.pgv2.percona.com serverside-
   applied
   customresourcedefinition.apiextensions.k8s.io/perconapgclusters.pgv2.percona.com serverside-
   applied
   customresourcedefinition.apiextensions.k8s.io/perconapgrestores.pgv2.percona.com serverside-
   applied
   customresourcedefinition.apiextensions.k8s.io/postgresclusters.postgres-
   operator.crunchydata.com serverside-applied
   serviceaccount/percona-postgresql-operator serverside-applied
   role.rbac.authorization.k8s.io/percona-postgresql-operator serverside-applied
   rolebinding.rbac.authorization.k8s.io/service-account-percona-postgresql-operator serverside-
   applied
   deployment.apps/percona-postgresql-operator serverside-applied
   ```

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

3. Deploy Percona Distribution for PostgreSQL:

   ```
   $ kubectl apply -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/
   v2.2.0/deploy/cr.yaml
   ```

   **Expected output**

   ```
   perconapgcluster.pgv2.percona.com/cluster1 created
   ```

**Note**

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

```
$ git clone -b v2.2.0 https://github.com/percona/percona-postgresql-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 Pods have reached their Running status:

```
$ kubectl get pg
```

**Expected output**

```
NAME       ENDPOINT                                 STATUS   POSTGRES   PGBOUNCER   AGE
cluster1   cluster1-pgbouncer.postgres-operator.svc ready    3          3           143m
```

### 12.1.1 Verifying the cluster operation

When creation process is over, you can try to connect to the cluster.

1. During the installation, the Operator has generated several secrets, including the one with password for default PostgreSQL user. This default user has the login name same as the the cluster.

   Use `kubectl get secrets` command to see the list of Secrets objects. The Secrets object you are interested in is named as `<cluster_name>-pguser-<cluster_name>` (substitute `<cluster_name>` with the name of your Percona Distribution for PostgreSQL Cluster). The default variant will be `cluster1-pguser-cluster1`. You can use the following command to get the password of this user:

   ```
   $ kubectl get secret <cluster_name>-<user_name>-<cluster_name> --
   template='{{.data.password | base64decode}}{{"\n"}}'
   ```

2. Run a container with `psql` tool and connect its console output to your terminal. The following command will do this, naming the new Pod `pg-client`:

   ```
   $ kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-postgresql:
   15 --restart=Never -- bash -il
   [postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer.postgres-
   operator.svc -p 5432 -U cluster1 cluster1
   ```

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

   This command will connect you as a `cluster1` user to a `cluster1` database via the PostgreSQL interactive terminal.

   ```
   psql (15)
   SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256,
   compression: off)
   Type "help" for help.
   pgdb=>
   ```

Last update: 2023-06-28

## 12.2 Install Percona Distribution for PostgreSQL on Google Kubernetes Engine (GKE)

Following steps will allow you to install the Operator and use it to manage Percona Distribution for PostgreSQL with the Google Kubernetes Engine. The document assumes some experience with Google Kubernetes Engine (GKE). For more information on the GKE, see the Kubernetes Engine Quickstart.

### 12.2.1 Prerequisites

All commands from this installation 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
```

### 12.2.2 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 `cluster-1`:

```
$ gcloud container clusters create cluster-1 --project <project name> --zone us-central1-a
--cluster-version  --machine-type n1-standard-4 --num-nodes=3
```

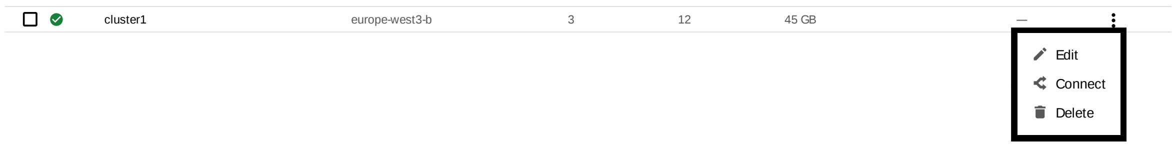
**Note**

You must edit the above command and other command-line statements to replace the `<project name>` placeholder with your project name. 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 4 vCPUs and 45 GB memory.

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:



| | | | | | |
|---|---|---|---|---|---|
| cluster1 | europe-west3-b | 3 | 12 | 45 GB | — |

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 cluster-1 --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
```

### 12.2.3 Install the Operator and deploy your PostgreSQL cluster

1. First of all, use the following `git clone` command to download the correct branch of the percona-postgresql-operator repository:

```
$ git clone -b v2.2.0 https://github.com/percona/percona-postgresql-operator
$ cd percona-postgresql-operator
```

2. Create the Kubernetes namespace for your cluster if needed (for example, let's name it `postgres-operator` ):

```
$ kubectl create namespace postgres-operator
```

**Expected output**

```
namespace/postgres-operator was created
```

**Note**

To use different namespace, specify other name instead of `postgres-operator` in the above command, and modify the `-n postgres-operator` parameter with it in the following two steps. You can also omit this parameter completely to deploy everything in the `default` namespace.

3. Deploy the Operator using the following command:

```
$ kubectl apply --server-side -f deploy/bundle.yaml -n postgres-operator
```

**Expected output**

```
customresourcedefinition.apiextensions.k8s.io/perconapgbackups.pgv2.percona.com serverside-
applied
customresourcedefinition.apiextensions.k8s.io/perconapgclusters.pgv2.percona.com serverside-
applied
customresourcedefinition.apiextensions.k8s.io/perconapgrestores.pgv2.percona.com serverside-
applied
customresourcedefinition.apiextensions.k8s.io/postgresclusters.postgres-
operator.crunchydata.com serverside-applied
serviceaccount/percona-postgresql-operator serverside-applied
role.rbac.authorization.k8s.io/percona-postgresql-operator serverside-applied
rolebinding.rbac.authorization.k8s.io/service-account-percona-postgresql-operator serverside-
applied
deployment.apps/percona-postgresql-operator serverside-applied
```

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

4. Deploy Percona Distribution for PostgreSQL:

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

**Expected output**

```
perconapgcluster.pgv2.percona.com/cluster1 created
```

Creation process will take some time. The process is over when both Operator and PostgreSQL Pods have reached their Running status:

```
$ kubectl get pg
```

**Expected output**

```
NAME       ENDPOINT                               STATUS   POSTGRES   PGBOUNCER   AGE
cluster1   cluster1-pgbouncer.postgres-operator.svc   ready    3          3           143m
```

**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 | Pods | Namespace | Cluster |
|---|---|---|---|---|---|
| cluster1-backup-7hsq | ✓ OK | Job | 0/1 | pg-opertor | cluster1 |
| cluster1-instance1-mntz | ✓ OK | Stateful Set | 1/1 | pg-opertor | cluster1 |
| cluster1-pgbouncer | ✓ OK | Deployment | 1/1 | pg-opertor | cluster1 |
| cluster1-repo-host | ✓ OK | Stateful Set | 1/1 | pg-opertor | cluster1 |
| cluster1-repo1-full | ✓ OK | Cron Job | 0/0 | pg-opertor | cluster1 |
| percona-postgresql-operator | ✓ OK | Deployment | 1/1 | pg-opertor | cluster1 |

### 12.2.4 Verifying the cluster operation

When creation process is over, `kubectl get pg` command will show you the cluster status as `ready`, and you can try to connect to the cluster.

1. During the installation, the Operator has generated several secrets, including the one with password for default PostgreSQL user. This default user has the login name same as the the cluster.

   Use `kubectl get secrets` command to see the list of Secrets objects. The Secrets object you are interested in is named as `<cluster_name>-pguser-<cluster_name>` (substitute `<cluster_name>` with the name of your Percona Distribution for PostgreSQL Cluster). The default variant will be `cluster1-pguser-cluster1`. You can use the following command to get the password of this user:

   ```
   $ kubectl get secret <cluster_name>-<user_name>-<cluster_name> --
   template='{{.data.password | base64decode}}{{"\n"}}'
   ```

2. Run a container with `psql` tool and connect its console output to your terminal. The following command will do this, naming the new Pod `pg-client`:

   ```
   $ kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-postgresql:
   15 --restart=Never -- bash -il
   [postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer.postgres-
   operator.svc -p 5432 -U cluster1 cluster1
   ```

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

   This command will connect you as a `cluster1` user to a `cluster1` database via the PostgreSQL interactive terminal.

   ```
   psql (15)
   SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256,
   compression: off)
   Type "help" for help.
   pgdb=>
   ```

### 12.2.5 Removing the GKE cluster

There are several ways that you can delete the cluster.

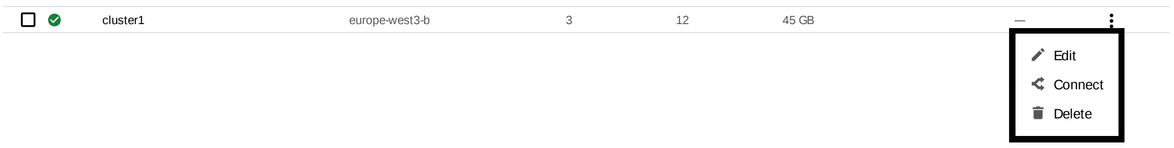
You can clean up the cluster with the `gcloud` command as follows:

```
$ gcloud container clusters delete <cluster name>
```

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!

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only ”ask me anything” sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-06-29

## 12.3 Install Percona Distribution for PostgreSQL on Amazon Elastic Kubernetes Service (EKS)

This guide shows you how to deploy Percona Operator for PostgreSQL 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.

### 12.3.1 Prerequisites

#### Software installation

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.

#### Creating 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.

    **Note**

    CSI driver is needed for the Operator to work propely, and is not included by default starting from the Amazon EKS version 1.22. Therefore sers with existing EKS cluster based on the version 1.22 or earlier need to install CSI driver before updating the EKS cluster to 1.23 or above.

### 12.3.2 Install the Operator and Percona Distribution for PostgreSQL

The following steps are needed to deploy the Operator and Percona Distribution for PostgreSQL in your Kubernetes environment:

1. Create the Kubernetes namespace for your cluster if needed (for example, let's name it `postgres-operator` ):

```
$ kubectl create namespace postgres-operator
```

**Expected output**

```
namespace/postgres-operator was created
```

**Note**

To use different namespace, specify other name instead of `postgres-operator` in the above command, and modify the `-n postgres-operator` parameter with it in the following two steps. You can also omit this parameter completely to deploy everything in the `default` namespace.

2. Deploy the Operator using the following command:

```
$ kubectl apply --server-side -f https://raw.githubusercontent.com/percona/percona-
postgresql-operator/v2.2.0/deploy/bundle.yaml -n postgres-operator
```

**Expected output**

```
customresourcedefinition.apiextensions.k8s.io/perconapgbackups.pgv2.percona.com serverside-
applied
customresourcedefinition.apiextensions.k8s.io/perconapgclusters.pgv2.percona.com serverside-
applied
customresourcedefinition.apiextensions.k8s.io/perconapgrestores.pgv2.percona.com serverside-
applied
customresourcedefinition.apiextensions.k8s.io/postgresclusters.postgres-
operator.crunchydata.com serverside-applied
serviceaccount/percona-postgresql-operator serverside-applied
role.rbac.authorization.k8s.io/percona-postgresql-operator serverside-applied
rolebinding.rbac.authorization.k8s.io/service-account-percona-postgresql-operator serverside-
applied
deployment.apps/percona-postgresql-operator serverside-applied
```

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

3. The operator has been started, and you can deploy your Percona Distribution for PostgreSQL cluster:

```
$ kubectl apply -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/
v2.2.0/deploy/cr.yaml -n postgres-operator
```

**Expected output**

```
perconapgcluster.pgv2.percona.com/cluster1 created
```

**Note**

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

```
$ git clone -b v2.2.0 https://github.com/percona/percona-postgresql-operator
```

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

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

The creation process may take some time. The process is over when both Operator and replica set Pods have reached their Running status:

```
$ kubectl get pg
```

**Expected output**

```
NAME       ENDPOINT                                STATUS   POSTGRES   PGBOUNCER   AGE
cluster1   cluster1-pgbouncer.postgres-operator.svc   ready    3          3           143m
```

### 12.3.3 Verifying the cluster operation

When creation process is over, `kubectl get pg` command will show you the cluster status as `ready`, and you can try to connect to the cluster.

1. During the installation, the Operator has generated several secrets, including the one with password for default PostgreSQL user. This default user has the login name same as the the cluster.

   Use `kubectl get secrets` command to see the list of Secrets objects. The Secrets object you are interested in is named as `<cluster_name>-pguser-<cluster_name>` (substitute `<cluster_name>` with the name of your Percona Distribution for PostgreSQL Cluster). The default variant will be `cluster1-pguser-cluster1`. You can use the following command to get the password of this user:

   ```
   $ kubectl get secret <cluster_name>-<user_name>-<cluster_name> --
   template='{{.data.password | base64decode}}{{"\n"}}'
   ```

2. Run a container with `psql` tool and connect its console output to your terminal. The following command will do this, naming the new Pod `pg-client`:

   ```
   $ kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-postgresql:
   15 --restart=Never -- bash -il
   [postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer.postgres-
   operator.svc -p 5432 -U cluster1 cluster1
   ```

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

   This command will connect you as a `cluster1` user to a `cluster1` database via the PostgreSQL interactive terminal.

   ```
   psql (15)
   SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256,
   compression: off)
   Type "help" for help.
   pgdb=>
   ```

### 12.3.4 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!

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

---

Last update: 2023-06-28

## 12.4 Install Percona Distribution for PostgreSQL on Kubernetes

Following steps will allow you to install the Operator and use it to manage Percona Distribution for PostgreSQL in a Kubernetes-based environment.

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

```
$ git clone -b v2.2.0 https://github.com/percona/percona-postgresql-operator
$ cd percona-postgresql-operator
```

**Note**

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

2. Create the Kubernetes namespace for your cluster if needed (for example, let's name it `postgres-operator` ):

```
$ kubectl create namespace postgres-operator
```

**Expected output**

```
namespace/postgres-operator was created
```

**Note**

To use different namespace, specify other name instead of `postgres-operator` in the above command, and modify the `-n postgres-operator` parameter with it in the following two steps. You can also omit this parameter completely to deploy everything in the `default` namespace.

3. Deploy the Operator using the following command:

```
$ kubectl apply --server-side  -f deploy/bundle.yaml -n postgres-operator
```

4. After the Operator is started Percona Distribution for PostgreSQL can be created at any time with the following command:

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

Creation process will take some time. The process is over when both Operator and replica set Pods have reached their Running status:

```
$ kubectl get pg
```

**Expected output**

```
NAME       ENDPOINT                                 STATUS   POSTGRES   PGBOUNCER   AGE
cluster1   cluster1-pgbouncer.postgres-operator.svc ready    3          3           143m
```

### 12.4.1 Verifying the cluster operation

When creation process is over, `kubectl get pg` command will show you the cluster status as `ready`, and you can try to connect to the cluster.

1. During the installation, the Operator has generated several secrets, including the one with password for default PostgreSQL user. This default user has the login name same as the the cluster.

   Use `kubectl get secrets` command to see the list of Secrets objects. The Secrets object you are interested in is named as `<cluster_name>-pguser-<cluster_name>` (substitute `<cluster_name>` with the name of your Percona Distribution for PostgreSQL Cluster). The default variant will be `cluster1-pguser-cluster1`. You can use the following command to get the password of this user:

   ```
   $ kubectl get secret <cluster_name>-<user_name>-<cluster_name> --
   template='{{.data.password | base64decode}}{{"\n"}}'
   ```

2. Run a container with `psql` tool and connect its console output to your terminal. The following command will do this, naming the new Pod `pg-client`:

   ```
   $ kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-postgresql:
   15 --restart=Never -- bash -il
   [postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer.postgres-
   operator.svc -p 5432 -U cluster1 cluster1
   ```

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

   This command will connect you as a `cluster1` user to a `cluster1` database via the PostgreSQL interactive terminal.

   ```
   psql (15)
   SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256,
   compression: off)
   Type "help" for help.
   pgdb=>
   ```

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-06-29

# 13. Configuration

## 13.1 Users

Operator provides a feature to manage users and databases in your PostgreSQL cluster. This document describes this feature, defaults and ways to fine tune your users.

### 13.1.1 Defaults

When you create a PostgreSQL cluster with the Operator and do not specify any additional users or databases, the Operator will do the following:

- Create a database that matches the name of your PostgreSQL cluster.
- Create an unprivileged PostgreSQL user with the name of the cluster. This user has access to the database created in the previous step.
- Create a Secret with the login credentials and connection details for the PostgreSQL user which is in relation to the database. This is stored in a Secret named `<clusterName>-pguser-<clusterName>`. These credentials include:
  - `user`: The name of the user account.
  - `password`: The password for the user account.
  - `dbname`: The name of the database that the user has access to by default.
  - `host`: The name of the host of the database. This references the Service of the primary PostgreSQL instance.
  - `port`: The port that the database is listening on.
  - `uri`: A PostgreSQL connection URI that provides all the information for logging into the PostgreSQL database via pgBouncer
  - `jdbc-uri`: A PostgreSQL JDBC connection URI that provides all the information for logging into the PostgreSQL database via the JDBC driver.

As an example, using our `cluster1` PostgreSQL cluster, we would see the following created:

- A database named `cluster1`.
- A PostgreSQL user named `cluster1`.
- A Secret named `cluster1-pguser-cluster1` that contains the user credentials and connection information.

### 13.1.2 Custom Users and Databases

Users and databases can be customized in `spec.users` section in the Custom Resource. Section can be changed at the cluster creation time and adjusted over time. Note the following:

- If `spec.users` is set during the cluster creation, the Operator will not create any default users or databases except for PostgreSQL. If you want additional databases, you will need to specify them.
- For each user added in `spec.users`, the Operator will create a Secret of the `<clusterName>-pguser-<userName>` format (such default Secret naming can be altered for the user with the `spec.users.secretName` option). This Secret will contain the user credentials.
- If no databases are specified, `dbname` and `uri` will not be present in the Secret.
- If at least one option under the `spec.users.databases` is specified, the first database in the list will be populated into the connection credentials.
- The Operator does not automatically drop users in case of removed Custom Resource options to prevent accidental data loss.
- Similarly, to prevent accidental data loss Operator does not automatically drop databases (see how to actually drop a database here).
- Role attributes are not automatically dropped if you remove them. You need to set the inverse attribute to actually drop them (e.g. `NOSUPERUSER`).
- The special `postgres` user can be added as one of the custom users; however, the privileges of this user cannot be adjusted.

#### Creating a New User

Change `PerconaPGCluster` Custom Resource (e.g. by editing your YAML manifest in the `deploy/cr.yaml` configuration file):

```yaml
...
spec:
  users:
    - name: perconapg
```

Apply the changes (e.g. with the usual `kubctl apply -f deploy/cr.yaml’ command) will create the new user:

- The user will only be able to connect to the default `postgres` database.
- The credentials of this user are populated in the `<clusterName>-pguser-perconapg` secret. There are no connection credentials.
- The user is unprivileged.

The following example shows how to create a new `pgtest` database and let `perconapg` user access it. The appropriate Custom Resource fragment will look as follows:

```yaml
...
spec:
  users:
    - name: perconapg
      databases:
        - pgtest
```

If you inspect the `<clusterName>-pguser-perconapg` Secret after applying the changes, you will see `dbname` and `uri` options populated there, and the database is created as well.

### Adjusting privileges

You can set role privileges by using the standard role attributes that PostgreSQL provides and adding them to the `spec.users.options` subsection in the Custom Resource. The following example will make the `perconapg` a superuser. You can add the following to the spec in your `deploy/cr.yaml`:

```yaml
...
spec:
  users:
    - name: perconapg
      databases:
        - pgtest
      options: "SUPERUSER"
```

Apply changes with the usual `kubctl apply -f deploy/cr.yaml' command.

To actually revoke the superuser privilege afterwards, you will need to do and apply the following change:

```yaml
...
spec:
  users:
    - name: perconapg
      databases:
        - pgtest
      options: "NOSUPERUSER"
```

If you want to add multiple privileges, you can use a space-separated list as follows:

```yaml
...
spec:
  users:
    - name: perconapg
      databases:
        - pgtest
      options: "CREATEDB CREATEROLE"
```

### `postgres` User

By default, the Operator does not create the `postgres` user. You can create it by applying the following change to your Custom Resource:

```yaml
...
spec:
  users:
    - name: postgres
```

This will create a Secret named `<clusterName>-pguser-postgres` that contains the credentials of the `postgres` account.

### Deleting users and databases

The Operator does not delete users and databases automatically. After you remove the user from the Custom Resource, it will continue to exist in your cluster. To remove a user and all of its objects, as a superuser you will need to run `DROP OWNED` in each database the user has objects in, and `DROP ROLE` in your PostgreSQL cluster.

```
DROP OWNED BY perconapg;
DROP ROLE perconapg;
```

For databases, you should run the `DROP DATABASE` command as a superuser:

```
DROP DATABASE pgtest;
```

#### Managing user passwords

If you want to rotate user's password, just remove the old password in the correspondent Secret: the Operator will immediately generate a new password and save it to the appropriate Secret. You can remove the old password with the `kubectl patch secret` command:

```
$ kubectl patch secret <clusterName>-pguser-<userName> -p '{"data":{"password":""}}'
```

Also, you can set a custom password for the user. Do it as follows:

```
$ kubectl patch secret <clusterName>-pguser-<userName> -p '{"stringData":
{"password":"<custom_password>", "verifier":""}}'
```

#### Superuser and pgBouncer

For security reasons we do not allow superusers to connect to cluster through pgBouncer by default. You can connect through `primary` service (read more in exposure documentation).

Otherwise you can use the proxy.pgBouncer.exposeSuperusers Custom Resource option to enable superusers connection via pgBouncer.

Last update: 2023-06-29

## 13.2 Exposing cluster

The Operator provides entry points for accessing the database by client applications. The database cluster is exposed with regular Kubernetes Service objects configured by the Operator.

This document describes the usage of Custom Resource manifest options to expose the clusters deployed with the Operator.

### 13.2.1 PgBouncer

We recommend exposing the cluster through PgBouncer, which is enabled by default. You can disable pgBouncer by setting `proxy.pgBouncer.replicas` to 0.

The following example deploys two pgBouncer nodes exposed through a LoadBalancer Service object:

```
proxy:
  pgBouncer:
    replicas: 2
    image: percona/percona-postgresql-operator:2.2.0-ppg14-pgbouncer
    expose:
      type: LoadBalancer
```

The Service will be called `<clusterName>-pgbouncer`:

```
$ kubectl get service
```

**Expected output**

```
NAME                 TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)          AGE
cluster1-pgbouncer   LoadBalancer   10.88.8.48     34.133.38.186   5432:30601/TCP   20m
```

You can connect to the database using the External IP of the load balancer and port `5432`.

If your application runs inside the Kubernetes cluster as well, you might want to use the Cluster IP Service type in `proxy.pgBouncer.expose.type`, which is the default. In this case to connect to the database use the internal domain name - `cluster1-pgbouncer.<namespace>.svc.cluster.local`.

### 13.2.2 Exposing the cluster without PgBouncer

You can connect to the cluster without a proxy. For that use `<clusterName>-ha` Service object:

```
$ kubectl get service
```

**Expected output**

```
NAME                 TYPE         CLUSTER-IP     EXTERNAL-IP    PORT(S)    AGE
cluster1-ha          ClusterIP    10.88.8.121    <none>         5432/TCP   115s
```

This service points to the active primary. In case of failover to the replica node, will change the endpoint automatically.

To change the Service type, use `expose.type` in the Custom Resource manifest. For example, the following manifest will expose this service through a load balancer:

```yaml
spec:
...
  expose:
    type: LoadBalancer
```

Last update: 2023-05-04

## 13.3 Changing PostgreSQL Options

Despite the Operator's ability to configure PostgreSQL and the large number of Custom Resource options, there may be situations where you need to pass specific options directly to your cluster's PostgreSQL instances. For this purpose, you can use the PostgreSQL dynamic configuration method provided by Patroni. You can pass PostgreSQL options to Patroni through the Operator Custom Resource, updating it with `deploy/cr.yaml` configuration file).

Custom PostgreSQL configuration options should be included into the `patroni.dynamicConfiguration.postgresql.parameters` subsection as follows:

```yaml
...
patroni:
  dynamicConfiguration:
    postgresql:
      parameters:
        max_parallel_workers: 2
        max_worker_processes: 2
        shared_buffers: 1GB
        work_mem: 2MB
```

Please note that configuration changes will be automatically applied to the running instances as soon as you apply Custom Resource changes in a usual way, running the `kubectl apply -f deploy/cr.yaml` command.

You can apply custom configuration in this way for both new and existing clusters.

Normally, options should be applied to PostgreSQL instances dynamically without restart, except the options with the postmaster context. Changing options which have `context=postmaster` will cause Patroni to initiate restart of all PostgreSQL instances, one by one. You can check the context of a specific option using the `SELECT name, context FROM pg_settings;` query to to see if the change should cause a restart or not.

**Note**

The Operator passes options to Patroni without validation, so there is a theoretical possibility of the cluster malfunction caused by wrongly configured PostgreSQL instances. Also, this configuration method is used for PostgreSQL options only and cannot be applied to change other Patroni dynamic configuration options. It means that options in the `parameters` subsection under `patroni.dynamicConfiguration.postgresql` will be applied, and everything else in `patroni.dynamicConfiguration.postgresql` will be ignored.

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-06-30

## 13.4 Binding Percona Distribution for PostgreSQL components to Specific Kubernetes/OpenShift Nodes

The operator does good job automatically assigning new Pods to nodes with sufficient resources to achieve balanced distribution across the cluster. Still there are situations when it is worth to ensure that pods will land on specific nodes: for example, to get speed advantages of the SSD equipped machine, or to reduce network costs choosing nodes in a same availability zone.

Appropriate sections of the deploy/cr.yaml file (such as `proxy.pgBouncer`) contain keys which can be used to do this, depending on what is the best for a particular situation.

### 13.4.1 Affinity and anti-affinity

Affinity makes Pod eligible (or not eligible - so called "anti-affinity") to be scheduled on the node which already has Pods with specific labels, or has specific labels itself (so called "Node affinity"). Particularly, Pod anti-affinity is good to reduce costs making sure several Pods with intensive data exchange will occupy the same availability zone or even the same node - or, on the contrary, to make them land on different nodes or even different availability zones for the high availability and balancing purposes. Node affinity is useful to assign PostgreSQL instances to specific Kubernetes Nodes (ones with specific hardware, zone, etc.).

Pod anti-affinity is controlled by the `affinity.podAntiAffinity` subsection, which can be put into `proxy.pgBouncer` and `backups.pgbackrest.repoHost` sections of the `deploy/cr.yaml` configuration file.

`podAntiAffinity` allows you to use standard Kubernetes affinity constraints of any complexity:

```
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 1
      podAffinityTerm:
        labelSelector:
          matchLabels:
            postgres-operator.crunchydata.com/cluster: keycloakdb
            postgres-operator.crunchydata.com/role: pgbouncer
        topologyKey: kubernetes.io/hostname
```

You can see the explanation of these affinity options in Kubernetes documentation.

### 13.4.2 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 `proxy.pgBouncer` and `backups.pgbackrest.repoHost` sections of the `deploy/cr.yaml` configuration file as follows:

```
topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: my-node-label
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        postgres-operator.crunchydata.com/instance-set: instance1
```

You can see the explanation of these affinity options in Kubernetes documentation.

### 13.4.3 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 latter variant also requires a `value` the key is equal to). Moreover, toleration should have a specified `effect`, which may be a self-explanatory `NoSchedule`, less strict `PreferNoSchedule`, or `NoExecute`. The last variant means that if a *taint* with `NoExecute` is assigned to node, then any Pod not tolerating this *taint* will be removed from the node, immediately or after the `tolerationSeconds` interval, like in the following example.

You can use `instances.tolerations` and `backups.pgbackrest.jobs.tolerations` subsections in the `deploy/cr.yaml` configuration file as follows:

```yaml
tolerations:
- effect: NoSchedule
  key: role
  operator: Equal
  value: connection-poolers
```

The Kubernetes Taints and Toleratins contains more examples on this topic.

Last update: 2022-12-30

## 13.5 Transport Layer Security (TLS)

The Percona Operator for PostgreSQL uses Transport Layer Security (TLS) cryptographic protocol for the following types of communication:

- Internal - communication between PostgreSQL instances in the cluster
- External - communication between the client application and the cluster

The internal certificate is also used as an authorization method for PostgreSQL Replica instances.

TLS security can be configured in several ways:

- the Operator can generate long-term certificates automatically at cluster creation time,
- the Operator can use a specifically installed *cert-manager*, which will automatically generate and renew short-term TLS certificates,
- you can also generate certificates manually.

The following subsections explain how to configure TLS security with the Operator yourself, as well as how to temporarily disable it if needed.

### 13.5.1 Install and use the *cert-manager*

#### About the *cert-manager*

The cert-manager is a Kubernetes certificate management controller which 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.

Self-signed issuer allows you to deploy and use the Operator without creating a cluster issuer separately.

#### Installation of 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 create namespace cert-manager
$ kubectl label namespace cert-manager certmanager.k8s.io/disable-validation=true
$ kubectl apply -f https://github.com/jetstack/cert-manager/releases/download/v1.8.0/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 cert-manager to create certificates. Whenever you check certificates for expiration, you will find that they are valid and short-term.

### 13.5.2 Allow the Operator to generate certificates automatically

The Operator is able to generate long-term certificates automatically and turn on encryption at cluster creation time, if there are no certificate secrets available. Just deploy your cluster as usual, with the `kubectl apply -f deploy/cr.yaml` command, and certificates will be generated.

### 13.5.3 Check connectivity to the cluster

You can check TLS communication with use of the `psql`, the standard interactive terminal-based frontend to PostgreSQL. The following command will spawn a new `pg-client` container, which includes needed command and can be used for the check (use your real cluster name instead of the `<cluster-name>` placeholder):

```
$ cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pg-client
spec:
  replicas: 1
  selector:
    matchLabels:
      name: pg-client
  template:
    metadata:
      labels:
        name: pg-client
    spec:
      containers:
        - name: pg-client
          image: perconalab/percona-distribution-postgresql:15
          imagePullPolicy: Always
          command:
          - sleep
          args:
          - "100500"
          volumeMounts:
            - name: ca
              mountPath: "/tmp/tls"
      volumes:
      - name: ca
        secret:
          secretName: <cluster_name>-ssl-ca
          items:
          - key: ca.crt
            path: ca.crt
            mode: 0777
EOF
```

Now get shell access to the newly created container, and launch the PostgreSQL interactive terminal to check connectivity over the encrypted channel (please use real cluster-name, PostgreSQL user login and password):

```
$ kubectl exec -it deployment/pg-client -- bash -il
[postgres@pg-client /]$ PGSSLMODE=verify-ca PGSSLROOTCERT=/tmp/tls/ca.crt psql postgres://
<postgresql-user>:<postgresql-password>@<cluster-name>-
pgbouncer.<namespace>.svc.cluster.local
```

Now you should see the prompt of PostgreSQL interactive terminal:

```
$ psql (15)
Type "help" for help.
pgdb=>
```

### 13.5.4 Generate certificates manually

To use custom TLS certificates for a Postgres cluster, you will need to create a Secret in the Namespace of your cluster that contains the TLS key (`tls.key`), TLS certificate (`tls.crt`) and the CA certificate (`ca.crt`) to use. The Secret should contain the following values:

```
data:
  ca.crt: <value>
  tls.crt: <value>
  tls.key: <value>
```

You should generate certificates twice: one set is for external communications, and another set is for internal ones. A secret created for the external use must be added to the `secrets.customTLSSecret.name` field of your Custom Resource. A certificate generated for internal communications must be added to the `secrets.customReplicationTLSSecret.name` field.

For example, if you have files named `ca.crt`, `hippo.key`, and `hippo.crt` stored on your local machine, you could run the following command:

```
$ kubectl create secret generic -n postgres-operator hippo.tls \
  --from-file=ca.crt=ca.crt \
  --from-file=tls.key=hippo.key \
  --from-file=tls.crt=hippo.crt
```

Now you can add the custom TLS Secret name to the `secrets.customTLSSecret.name` field in your Rustom Resource:

```
secrets:
  customTLSSecret:
    name: hippo.tls
```

Don't forget to apply changes as usual:

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

### 13.5.5 Check your certificates for expiration

1. First, check the necessary secrets names (`cluster1-cluster-cert` and `cluster1-replication-cert` by default):

   ```
   $ kubectl get secrets
   ```

   You will have the following response:

   ```
   NAME                            TYPE     DATA   AGE
   cluster1-cluster-cert           Opaque   3      11m
   ...
   cluster1-replication-cert       Opaque   3      11m
   ...
   ```

2. Now use the following command to find out the certificates validity dates, substituting Secrets names if necessary:

   ```
   $ {
     kubectl get secret/cluster1-replication-cert -o jsonpath='{.data.tls\.crt}' | base64 --
   decode | openssl x509 -noout -dates
     kubectl get secret/cluster1-cluster-cert -o jsonpath='{.data.ca\.crt}' | base64 --
   decode | openssl x509 -noout -dates
     }
   ```

   The resulting output will be self-explanatory:

   ```
   notBefore=Jun 28 10:20:19 2023 GMT
   notAfter=Jun 27 11:20:19 2024 GMT
   notBefore=Jun 28 10:20:18 2023 GMT
   notAfter=Jun 25 11:20:18 2033 GMT
   ```

### 13.5.6 Keep certificates after deleting the cluster

In case of cluster deletion, objects, created for SSL (Secret, certificate, and issuer) are not deleted by default.

If the user wants the cleanup of objects created for SSL, there is a finalizers.percona.com/delete-ssl Custom Resource option, which can be set in `deploy/cr.yaml`: if this finalizer is set, the Operator will delete Secret, certificate and issuer after the cluster deletion event.

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-06-29

## 13.6 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
- PostgreSQL version
- PgBackRest version
- Was the Operator deployed with Helm
- Are sidecar containers used
- Are backups used

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/pg-db --version 2.2.0 --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 `disable_telemetry` key and set it to `true`:

```
...
disable_telemetry: "true"
...
```

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-05-03

# 14. Management

## 14.1 Upgrade from version 1 to version 2

### 14.1.1 Upgrade from the Operator version 1.x to version 2.x

The Operator version 2.x has a lot of differences compared to the version 2.x. This makes upgrading from version 1.x to version 2.x quite different from a normal upgrade. In fact, you have to migrate the cluster from version 1.x to version 2.x.

There are several ways to do such version 1.x to version 2.x upgrade. Choose the method based on your downtime preference and roll back strategy:

| | **Pros** | **Cons** |
|---|---|---|
| Data Volumes migration - re-use the volumes that were created by the Operator version 1.x | The simplest method | - Requires downtime<br>- Impossible to roll back |
| Backup and restore - take the backup with the Operator version 1.x and restore it to the cluster deployed by the Operator version 2.x | Allows you to quickly test version 2.x | Provides significant downtime in case of migration |
| Replication - replicate the data from the Operator version 1.x cluster to the standby cluster deployed by the Operator version 2.x | - Quick test of v2 cluster<br>- Minimal downtime during upgrade | Requires significant computing resources to run two clusters in parallel |

**Contact Us**

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-06-30

### 14.1.2 Upgrade using data volumes

**Prerequisites:**

The following conditions should be met for the Volumes-based migration:

- You have a version 1.x cluster with `spec.keepData: true` in the Custom Resource
- You have both Operators deployed and allow them to control resources in the same namespace
- Old and new clusters must be of the same PostgreSQL major version

This migration method has two limitations. First of all, this migration method introduces a downtime. Also, you can only reverse such migration by restoring the old cluster from the backup. See other migration methods if you need lower downtime and a roll back plan.

#### Prepare version 1.x cluster for the migration

1. Remove all Replicas from the cluster, keeping only primary running. It is required to assure that Volume of the primary PVC does not change. The `deploy/cr.yaml` configuration file should have it as follows:

```
...
pgReplicas:
    hotStandby:
      size: 0
```

2. Apply the Custom Resource in a usual way:

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

3. When all Replicas are gone, proceed with removing the cluster. Double check that `spec.keepData` is in place, otherwise the Operator will delete the volumes!

```
$ kubectl delete perconapgcluster cluster1
```

4. Find PVC for the Primary and `pgBackRest`:

```
$ kubectl get pvc --selector=pg-cluster=cluster1 -n pgo
```

**Expected output**

```
NAME                 STATUS   VOLUME                                     CAPACITY   ACCESS
MODES   STORAGECLASS   AGE
cluster1             Bound    pvc-940cdc23-cd4c-4f62-ac3a-dc69850042b0   1Gi
RWO            standard-rwo   57m
cluster1-pgbr-repo   Bound    pvc-afb00490-5a45-45cb-a1cb-10af8e48bb13   1Gi
RWO            standard-rwo   57m
```

A third PVC used to store write-ahead logs (WAL) may also be present if external WAL volumes were enabled for the cluster.

5. Permissions for `pgBackRest` repo folders are managed differently in version 1 and version 2. We need to change the ownership of the `backrest` folder on the Persistent Volume to avoid errors during migration. Running a `chown` command within a container fixes this problem. You can use the following manifest to execute it:

```yaml title="chown-pod.yaml" apiVersion: v1 kind: Pod metadata: name: chown-pod spec: volumes: - name: backrestrepo persistentVolumeClaim: claimName: cluster1-pgbr-repo containers: - name: task-pv-container image: ubuntu command: - chown - -R - 26:26 - /backrestrepo/cluster1-backrest-shared-repo volumeMounts: - mountPath: "/backrestrepo" name: backrestrepo

```
Apply it as follows:

```{.bash data-prompt="$"}
$ kubectl apply -f chown-pod.yaml -n pgo
```

#### Execute the migration to version 2.x

The old cluster is shut down, and Volumes are ready to be used to provision the new cluster managed by the Operator version 2.x.

1. Install the Operator version 2 (if not done yet). Pick your favorite method from our documentaion.
2. Run the following command to show the names of PVC belonging to the old cluster:

```
$ kubectl get pvc --selector=pg-cluster=cluster1 -n pgo
```

**Expected output**

```
NAME                 STATUS   VOLUME                                     CAPACITY   ACCESS
MODES   STORAGECLASS   AGE
cluster1             Bound    pvc-db9bf618-04d5-4807-948d-e32e81098575   1Gi
RWO            standard-rwo   87m
cluster1-pgbr-repo   Bound    pvc-37d93aa9-bf02-4295-bbbc-c1f834ed6045   1Gi
RWO            standard-rwo   87m
```

3. Now edit the Custom Resource manifest ( `deploy/cr.yaml` configuration file) of the version 2.x cluster: add fields to the `dataSource.volumes` subsection, pointing to the PVCs of the version 1.x cluster:

```
...
dataSource:
  volumes:
      pgDataVolume:
        pvcName: cluster1
        directory: cluster1
      pgBackRestVolume:
        pvcName: cluster1-pgbr-repo
        directory: cluster1-backrest-shared-repo
```

4. Do not forget to set the proper PostgreSQL major version. It must be the same version that was used in version 1 cluster. You can set the version in the corresponding `image` sections and `postgresVersion`. The following example sets version 14:

```
spec:
  image: percona/percona-postgresql-operator:2.2.0-ppg14-postgres
  postgresVersion: 14
  proxy:
    pgBouncer:
      image: percona/percona-postgresql-operator:2.2.0-ppg14-pgbouncer
  backups:
    pgbackrest:
      image: percona/percona-postgresql-operator:2.2.0-ppg14-pgbackrest
```

5. Apply the manifest:

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

The new cluster will be provisioned shortly using the volume of the version 1.x cluster. You should remove the `spec.datasource.volumes` section from your manifest.

**Contact Us**

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only ”ask me anything” sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

---

Last update: 2023-06-30

### 14.1.3 Upgrade using backup and restore

This method allows you to migrate from the version 1.x to version 2.x cluster by restoring (actually creating) a new version 2.x PostgreSQL cluster using a backup from the version 1.x cluster.

**Note**

To make sure that all transactions are captured in the backup, you need to stop the old cluster. This bringы downtime to the application.

#### Prepare the backup

1. Create the backup on the version 1.x cluster, following the official guide for manual (on-demand) backups. This involves preparing the manifest in YAML and applying it in the ususal way:

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

2. Pause or delete the version 1.x cluster to ensure that you have the latest data.

   **Warning**

   Before deleting the cluster, make sure that the spec.keepBackups Custom Resource option is set to `true`. When it's set, local backups will be kept after the cluster deletion, so you can proceed with deleting your cluster as follows:

   ```
   $ kubectl delete perconapgcluster cluster1
   ```

#### Restore the backup as a version 2.x cluster

##### Restore from S3 / Google Cloud Storage for backups repository

1. To restore from the S3 or Google Cloud Storage for backups (GCS) repository, you should first configure the `spec.backups.pgbackrest.repos` subsection in your version 2.x cluster Custom Resource to point to the backup storage system. Just follow the repository documentation instruction for S3 or GCS. For example, for GCS you can define the repository similar to the following:

```
spec:
  backups:
    pgbackrest:
      repos:
      - name: repo1
        gcs:
          bucket: MY-BUCKET
          region: us-central1
```

2. Create and configure any required Secrets or desired custom pgBackrest configuration as described in the backup documentation for te Operator version 2.x.
3. Set the repository path in the `backups.pgbackrest.global` subsection. By default it is `/backrestrepo/&lt;clusterName>-backrest-shared-repo`:

```
  spec:
  backups:
    pgbackrest:
      global:
        repo1: /backrestrepo/cluster1-backrest-shared-repo
```

4. Set the `spec.dataSource` option to create the version 2.x cluster from the specific repository:

```
spec:
  dataSource:
    postgresCluster:
      repoName: repo1
```

   You can also provide other pgBackRest restore options, e.g. if you wish to restore to a specific point-in-time (PITR).
5. Create the version 2.x cluster:

```
$ kubectl apply -f cr.yaml
```

Contact Us

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-06-30

### 14.1.4 Migrate using Standby

This method allows you to migrate from version 1,x to version 2.x by creating a new version 2.x PostgreSQL cluster in a "standby" mode, mirroring the version 1.x cluster to it continuously. This method can provide minimal downtime, but requires additional computing resources to run two clusters in parallel.

This method only works if the version 1.x cluster uses Amazon S3 or S3-compatible storage, or Google Cloud storage (GCS) for backups. For more information on standby clusters, please refer to [this article].

#### Migrate to version 2

There is no need to perform any additional configuration on version 1.x cluster, you will only need to configure the version 2.x one.

1. Configure `spec.backups.pgbackrest.repos` Custom Resource option to point to the backup storage system. For example, for GCS, the repository would be defined similar to the following:

```
spec:
  backups:
    pgbackrest:
      repos:
      - name: repo1
        gcs:
          bucket: MY-BUCKET
          region: us-central1
```

2. Create and configure any required secrets or desired custom pgBackrest configuration as described in the backup documentation for the version 2.x.
3. Set the repository path in `backups.pgbackrest.global` section of the Custom Resource configuration file. By default it will be `/backrestrepo/&lt;clusterName>-backrest-shared-repo`:

```
      spec:
      backups:
        pgbackrest:
          global:
            repo1: /backrestrepo/cluster1-backrest-shared-repo
```

4. Enable the standby mode in `spec.standby` and point to the repository:

```
spec:
  standby:
    enabled: true
    repoName: repo1
```

5. Create the version 2.x cluster:

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

### Promote version 2.x cluster

Once the standby cluster is up and running, you can promote it.

1. Delete version 1.x cluster, but ensure that `spec.keepBackups` is set to `true`.

```
$ kubectl delete perconapgcluster cluster1
```

2. Promote version 2.x cluster by disabling the standby mode:

```
spec:
  standby:
    enabled: false
```

You can use version 2.x cluster now. Also the 2.x version is now managing the object storage with backups, so you should not start your old cluster.

### Create the replication user

Right after disabling standby, run the following SQL commands as a PostgreSQL superuser. For example, you can login as the `postgres` user, or exec into the Pod and use `psql`:

- add the managed replication user

```
CREATE ROLE _crunchyrepl WITH LOGIN REPLICATION;
```

- allow for the replication user to execute the functions required as part of "rewinding"

```
GRANT EXECUTE ON function pg_catalog.pg_ls_dir(text, boolean, boolean) TO _crunchyrepl;
GRANT EXECUTE ON function pg_catalog.pg_stat_file(text, boolean) TO _crunchyrepl;
GRANT EXECUTE ON function pg_catalog.pg_read_binary_file(text) TO _crunchyrepl;
GRANT EXECUTE ON function pg_catalog.pg_read_binary_file(text, bigint, bigint, boolean) TO
_crunchyrepl;
```

The above step will be automated in upcoming releases.

#### Contact Us

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-06-30

## 14.2 Providing Backups

The Operator allows doing backups in two ways. *Scheduled backups* are configured in the deploy/cr.yaml file to be executed automatically in proper time. *On-demand backups* can be done manually at any moment.

The Operator uses the open source pgBackRest backup and restore utility.

#### Backup repositories

A special *pgBackRest repository* is created by the Operator along with creating a new PostgreSQL cluster to facilitate the usage of the pgBackRest features in it (you can notice additional `repo-host` Pod after the cluster creation).

The Operator can use the following variants of cloud storage outside the Kubernetes cluster to keep PostgreSQL backups:

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

It is also possible to store backups in Kubernetes, just on a Persistent Volume attached to the pgBackRest Pod.

Each pgBackRest repository consists of the following Kubernetes objects:

- A Deployment,
- A Secret that contains information that is specific to the PostgreSQL cluster (e.g. SSH keys, AWS S3 keys, etc.),
- A Pod with a number of supporting scripts,
- A Service.

You can have up to 4 pgBackRest repositories named as `repo1`, `repo2`, `repo3`, and `repo4`.

### 14.2.1 Backup types

The PostgreSQL Operator supports three types of pgBackRest backups:

- `full`: A full backup of all the contents of the PostgreSQL cluster,
- `differential`: A backup of only the files that have changed since the last full backup,
- `incremental`: A backup of only the files that have changed since the last full or differential backup. Incremental backup is the default choice.

### 14.2.2 Backup retention

The Operator also supports setting pgBackRest retention policies for full and differential backups. When a full backup expires according to the retention policy, pgBackRest cleans up all the files related to this backup and to write-ahead log. So, expiring of a full backup with some incremental backups based on it results in expiring all these incremental backups.

Backup retention can be controlled by the following pgBackRest options:

- `--<repo name>-retention-full` how much full backups to retain,
- `--<repo name>-retention-diff` how much differential backups to retain.

Backup retention type can be either `count` (the number of backups to keep) or `time` (the number of days a backup should be kept for).

You can set both backups type and retention policy for each of 4 repositories as follows.

```
backups:
    pgbackrest:
...
      global:
        repo1-retention-full: "14"
        repo1-retention-full-type: time
        ...
```

### 14.2.3 Backup storage

You should configure backup storage for your repositories in the `backups.pgbackrest.repos` section of the `deploy/cr.yaml` configuration file.

#### Configuring the S3-compatible backup storage

In order to use S3-compatible storage for backups you need to provide some S3-related information, such as proper S3 bucket name, endpoint, etc. This information can be passed to pgBackRest via the following `deploy/cr.yaml` options in the `backups.pgbackrest.repos` subsection:

- `bucket` specifies the AWS S3 bucket that should be utilized, for example `my-postgresql-backups-example`,
- `endpoint` specifies the S3 endpoint that should be utilized, for example `s3.amazonaws.com`,
- `region` specifies the AWS S3 region that should be utilized, for example `us-east-1`.

You also need to supply pgBackRest with base64-encoded AWS S3 key and AWS S3 key secret stored along with other sensitive information in Kubernetes Secrets.

1. Put your AWS S3 key and AWS S3 key secret into the base64-encoded pgBackRest configuration with your pgBackRest repository name. In case of the `repo1` repository it can be done as follows:

   **in Linux**

   ```
   $ cat <<EOF | base64 --wrap=0
   [global]
   repo1-s3-key=<YOUR_AWS_S3_KEY>
   repo1-s3-key-secret=<YOUR_AWS_S3_KEY_SECRET>
   EOF
   ```

   **in macOS**

   ```
   $ cat <<EOF | base64
   [global]
   repo1-s3-key=<YOUR_AWS_S3_KEY>
   repo1-s3-key-secret=<YOUR_AWS_S3_KEY_SECRET>
   EOF
   ```

2. Create the Secret configuration file with the resulted base64-encoded string as the following `cluster1-pgbackrest-secrets.yaml` example:

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: cluster1-pgbackrest-secrets
   type: Opaque
   data:
     s3.conf: <base64-encoded-configuration-contents>
   ```

   **Note**

   This Secret can store credentials for several repositories presented as separate data keys.

   When done, create the Secrets object from this yaml file:

   ```
   $ kubectl apply -f cluster1-pgbackrest-secrets.yaml
   ```

3. Update your `deploy/cr.yaml` configuration with the S3 credentials Secret in the `backups.pgbackrest.configuration` subsection, and put all other S3 related information into the options of one of your repositories in the `backups.pgbackrest.repos` subsection. For example, the S3 storage for the `repo1` repository would look as follows.

   ```
   ...
   backups:
     pgbackrest:
       ...
       configuration:
         - secret:
             name: cluster1-pgbackrest-secrets
       ...
       repos:
       - name: repo1
         s3:
           bucket: "<YOUR_AWS_S3_BUCKET_NAME>"
           endpoint: "<YOUR_AWS_S3_ENDPOINT>"
           region: "<YOUR_AWS_S3_REGION>"
   ```

4. Finally, create or update the cluster:

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

#### Configuring Google Cloud Storage for backups

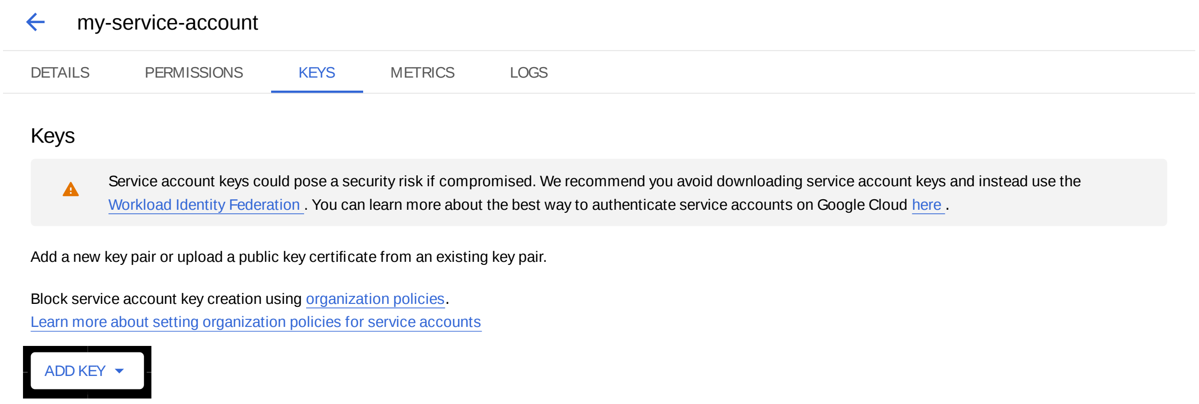
You can configure Google Cloud Storage as an object store for backups similarly to S3 storage.

In order to use Google Cloud Storage (GCS) for backups you need to provide a proper GCS bucket name. Bucket name can be passed to `pgBackRest` via the `gcs.bucket` key in the `backups.pgbackrest.repos` subsection of `deploy/cr.yaml`.

The Operator will also need your service account key to access storage.

1. Create your service account key following the official Google Cloud instructions.
2. Export this key from your Google Cloud account.

   You can find your key in the Google Cloud console (select *IAM & Admin* → *Service Accounts* in the left menu panel, then click your account and open the *KEYS* tab):

   Keys

   Service account keys could pose a security risk if compromised. We recommend you avoid downloading service account keys and instead use the Workload Identity Federation . You can learn more about the best way to authenticate service accounts on Google Cloud here .

   Add a new key pair or upload a public key certificate from an existing key pair.

   Block service account key creation using organization policies.
   Learn more about setting organization policies for service accounts

   Click the *ADD KEY* button, chose *Create new key* and chose *JSON* as a key type. These actions will result in downloading a file in JSON format with your new private key and related information.
3. Now you should create the Kubernetes Secret using base64-encoded versions of two files: the file containing the private key you have just downloaded, and the special `gcs.conf` configuration file.

   The content of the `gcs.conf` file depends on the repository name. In case of the `repo1` repository, it looks as follows:

   ```
   [global]
   repo1-gcs-key=/etc/pgbackrest/conf.d/gcs-key.json
   ```

   You can encode a text file with the `base64 --wrap=0 <filename>` command (or just `base64 <filename>` in case of Apple macOS). When done, create the following yaml file with your cluster name and base64-encoded files contents as the following `cluster1-pgbackrest-secrets.yaml` example:

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: cluster1-pgbackrest-secrets
   type: Opaque
   data:
     gcs-key.json: <base64-encoded-json-file-contents>
     gcs.conf: <base64-encoded-conf-file-contents>
   ```

   **Note**

   This Secret can store credentials for several repositories presented as separate data keys.

   Create the Secrets object from this YAML file:

   ```
   $ kubectl apply -f cluster1-pgbackrest-secrets.yaml
   ```
4. Update your `deploy/cr.yaml` configuration with your GCS credentials Secret in the `backups.pgbackrest.configuration` subsection, and put GCS bucket name into the `bucket` option of one of your repositories in the `backups.pgbackrest.repos` subsection. For example, GCS storage for the `repo3` repository would look as follows.

```
...
backups:
  pgbackrest:
    ...
    configuration:
      - secret:
          name: cluster1-pgbackrest-secrets
    ...
    repos:
    - name: repo3
      gcs:
        bucket: "<YOUR_GCS_BUCKET_NAME>"
```

5. Finally, create or update the cluster:

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

#### Configuring Azure Blob Storage for backups (tech preview)

In order to use Microsoft Azure Blob Storage for backups you need to provide a proper Azure container name. It can be passed to `pgBackRest` via the `azure.container` key in the `backups.pgbackrest.repos` subsection of `deploy/cr.yaml`.

The Operator will also need a Kubernetes Secret with your Azure Storage credentials to access the storage.

1. Put your Azure storage account name and key into the base64-encoded pgBackRest configuration with your pgBackRest repository name. In case of the `repo1` repository it can be done as follows:

   **in Linux**

   ```
   $ cat <<EOF | base64 --wrap=0
   [global]
   repo1-azure-account=<AZURE_STORAGE_ACCOUNT_NAME>
   repo1-azure-key=<AZURE_STORAGE_ACCOUNT_KEY>
   EOF
   ```

   **in macOS**

   ```
   $ cat <<EOF | base64
   [global]
   repo1-azure-account=<AZURE_STORAGE_ACCOUNT_NAME>
   repo1-azure-key=<AZURE_STORAGE_ACCOUNT_KEY>
   EOF
   ```

2. Create the Secret configuration file with the resulted base64-encoded string as the following `cluster1-pgbackrest-secrets.yaml` example:

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: cluster1-pgbackrest-secrets
   type: Opaque
   data:
     azure.conf: <base64-encoded-configuration-contents>
   ```

   **Note**

   This Secret can store credentials for several repositories presented as separate data keys.

   When done, create the Secrets object from this yaml file:

   ```
   $ kubectl apply -f cluster1-pgbackrest-secrets.yaml
   ```

3. Update your `deploy/cr.yaml` configuration with the Azure Storage credentials Secret in the `backups.pgbackrest.configuration` subsection, and put Azure container name into the options of one of your repositories in the `backups.pgbackrest.repos` subsection. For example, the Azure storage for the `repo1` repository would look as follows.

   ```
   ...
   backups:
     pgbackrest:
       ...
       configuration:
         - secret:
             name: cluster1-pgbackrest-secrets
       ...
       repos:
       - name: repo1
         azure:
           container: "<YOUR_AZURE_CONTAINER>"
   ```

4. Finally, create or update the cluster:

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

## 14.2.4 Scheduling backups

Backups schedule is defined on per-repository basis in the `backups.pgbackrest.repos` subsection of the `deploy/cr.yaml` file. You can supply each repository with a `schedules.<backup type>` key equal to an actual schedule specified in crontab format.

Here is an example of `deploy/cr.yaml` which uses `repo1` repository for backups:

```
...
backups:
  pgbackrest:
  ...
        repos:
        - name: repo1
          schedules:
            full: "0 0 * * 6"
            differential: "0 1 * * 1-6"
          ...
```

The schedule is specified in crontab format as explained in Custom Resource options.

## 14.2.5 Making on-demand backup

To make an on-demand backup, the user should use a backup configuration file. The example of the backup configuration file is deploy/backup.yaml:

```
apiVersion: pg.percona.com/v2beta1
kind: PerconaPGBackup
metadata:
  name: backup1
spec:
  pgCluster: cluster1
  repoName: repo1
#  options:
#  - --type=full
```

Fill it with the proper repository name to be used for this backup, and any needed pgBackRest command line options.

When the backup options are configured, execute the actual backup command:

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

## 14.2.6 Restore the cluster from a previously saved backup

The Operator supports the ability to perform a full restore on a PostgreSQL cluster as well as a point-in-time-recovery. There are two types of ways to restore a cluster:

- restore to a new cluster using the dataSource.postgresCluster subsection,
- restore in-place, to an existing cluster (note that this is destructive) using the backups.restore subsection.

#### Restore to an existing PostgreSQL cluster

To restore the previously saved backup the user should use a *backup restore* configuration file. The example of the backup configuration file is deploy/restore.yaml:

```
apiVersion: pg.percona.com/v2beta1
kind: PerconaPGRestore
metadata:
  name: restore1
spec:
  pgCluster: cluster1
  repoName: repo1
  options:
  - --type=time
  - --target="2022-11-30 15:12:11+03"
```

The following keys are the most important ones:

- `pgCluster` specifies the name of your cluster,
- `repoName` specifies the name of one of the 4 pgBackRest repositories, already configured in the `backups.pgbackrest.repos` subsection,
- `options` passes through any pgBackRest command line options.

The actual restoration process can be started as follows:

```
$ kubectl apply -f deploy/restore.yaml
```

#### Restore the cluster with point-in-time recovery

Point-in-time recovery functionality allows users to revert the database back to a state before an unwanted change had occurred.

**Note**

For this feature to work, the Operator initiates a full backup immediately after the cluster creation, to use it as a basis for point-in-time recovery when needed (this backup is not listed in the output of the `kubectl get pg-backup` command).

You can set up a point-in-time recovery using the normal restore command of pgBackRest with few additional `spec.options` fields in `deploy/restore.yaml`:

- set `--type` option to `time`,
- set `--target` to a specific time you would like to restore to. You can use the typical string formatted as `<YYYY-MM-DD HH:MM:DD>`, optionally followed by a timezone offset: `"2021-04-16 15:13:32+00"` (`+00` in the above example means just UTC),
- optional `--set` argument allows you to choose the backup which will be the starting point for point-in-time recovery (look through the available backups with the `kubectl get pg-backup` command to find out the proper backup name). This option must be specified if the target is one or more backups away from the current moment.

After setting these options in the *backup restore* configuration file, follow the standard restore instructions.

**Note**

Make sure you have a backup that is older than your desired point in time. You obviously can't restore from a time where you do not have a backup. All relevant write-ahead log files must be successfully pushed before you make the restore.

#### Restore to a new PostgreSQL cluster

Restoring to a new PostgreSQL cluster allows you to take a backup and create a new PostgreSQL cluster that can run alongside an existing one. There are several scenarios where using this technique is helpful:

- Creating a copy of a PostgreSQL cluster that can be used for other purposes. Another way of putting this is *creating a clone*.
- Restore to a point-in-time and inspect the state of the data without affecting the current cluster.

To create a new PostgreSQL cluster from either the active one, or a former cluster whose pgBackRest repository still exists, use the dataSource.postgresCluster subsection options. The content of this subsection should copy the `backups` keys of the original cluster - ones needed to carry on the restore:

- `dataSource.postgresCluster.clusterName` should contain the new cluster name,
- `dataSource.postgresCluster.options` allow you to set the needed pgBackRest command line options,
- `dataSource.postgresCluster.repoName` should contain the name of the pgBackRest repository, while the actual storage configuration keys for this repository should be placed into `dataSource.pgbackrest.repo` subsection,
- `dataSource.pgbackrest.configuration.secret.name` should contain the name of a Kubernetes Secret with credentials needed to access cloud storage, if any.

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-06-30

## 14.3 High availability and scaling

One of the great advantages brought by Kubernetes and the OpenShift platform is the ease of an application scaling. Scaling an application results in adding resources or Pods and scheduling them to available Kubernetes nodes.

Scaling can be vertical and horizontal. Vertical scaling adds more compute or storage resources to PostgreSQL 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.

### 14.3.1 Vertical scaling

There are multiple components that Operator deploys and manages: PostgreSQL instances, pgBouncer connection pooler, etc. To add or reduce CPU or Memory you need to edit corresponding sections in the Custom Resource. We follow the structure for requests and limits that Kubernetes provides.

To add more resources to your PostgreSQL instances edit the following section in the Custom Resource:

```
spec:
...
  instances:
  - name: instance1
    replicas: 3
    resources:
      limits:
        cpu: 2.0
        memory: 4Gi
```

Use our reference documentation for the Custom Resource options for more details about other components.

### 14.3.2 High availability

Percona Operator allows you to deploy highly-available PostgreSQL clusters. There are two ways how to control replicas in your HA cluster:

1. Through changing `spec.instances.replicas` value
2. By adding new entry into `spec.instances`

### 14.3.3 Using `spec.instances.replicas`

For example, you have the following Custom Resource manifest:

```
spec:
...
  instances:
    - name: instance1
      replicas: 2
```

This will provision a cluster with two nodes - one Primary and one Replica. Add the node by changing the manifest...

```
spec:
...
```

```
  instances:
    - name: instance1
      replicas: 3
```

…and applying the Custom Resource:

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

The Operator will provision a new replica node. It will be ready and available once data is synchronized from Primary.

### 14.3.4 Using `spec.instances`

Each instance's entry has its own set of parameters, like resources, storage configuration, sidecars, etc. When you add a new entry into instances, this creates replica PostgreSQL nodes, but with a new set of parameters. This can be useful in various cases:

- Test or migrate to new hardware
- Blue-green deployment of a new configuration
- Try out new versions of your sidecar containers

For example, you have the following Custom Resource manifest:

```
spec:
...
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        storageClassName: old-ssd
        accessModes:
        - ReadWriteOnce
        resources:
          requests:
            storage: 100Gi
```

Now you have a goal to migrate to new disks, which are coming with the `new-ssd` storage class. You can create a new instance entry. This will instruct the Operator to create additional nodes with the new configuration keeping your existing nodes intact.

```
spec:
...
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        storageClassName: old-ssd
        accessModes:
        - ReadWriteOnce
        resources:
          requests:
            storage: 100Gi
    - name: instance2
      replicas: 2
      dataVolumeClaimSpec:
        storageClassName: new-ssd
        accessModes:
        - ReadWriteOnce
```

```
          resources:
            requests:
              storage: 100Gi
```

Last update: 2023-06-22

## 14.4 Monitoring

Percona Monitoring and Management (PMM) provides an excellent solution to monitor Percona Distribution for PostgreSQL.

**Note**

Only PMM 2.x versions are supported by the Operator.

PMM is a client/server application. PMM Client runs on each node with the database you wish to monitor: it collects needed metrics and sends gathered data to PMM Server. As a user, you connect to PMM Server to see database metrics on a number of dashboards.

That’s why PMM Server and PMM Client need to be installed separately.

### 14.4.1 Installing the PMM Server

PMM Server runs as a *Docker image*, a *virtual appliance*, or on an *AWS instance*. Please refer to the official PMM documentation for the installation instructions.

### 14.4.2 Installing the PMM Client

The following steps are needed for the PMM client installation in your Kubernetes-based environment:

1. The PMM client installation is initiated by updating the `pmm` section in the deploy/cr.yaml file.
    - set `pmm.enabled=true`
    - set the `pmm.serverHost` key to your PMM Server hostname or IP address (it should be resolvable and reachable from within your cluster)
    - authorize PMM Client within PMM Server: acquire the API Key from your PMM Server and set `PMM_SERVER_KEY` in the deploy/secrets.yaml secrets file to this obtained API Key value. Keep in mind that you need an API Key with the "Admin" role. The API Key won't be rotated automatically.

**Note**

Alternatively, you can query your PMM Server installation for the API Key using `curl` and `jq` utils as follows:

```
$ 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)
```

Don't forget to use your real PMM Server login, password, and host address (same as in `pmm.serverHost` key) instead of the `<login>:<password>@<server_host>` placeholders.

**Info**

You use `deploy/secrets.yaml` file to *create* Secrets Object. The file contains all values for each key/value pair in a convenient plain text format. But the resulting Secrets contain passwords stored as base64-encoded strings. If you want to *update* password field, you'll need to encode the value into base64 format. To do this, you can run `echo -n "password" | base64 --wrap=0` (or just `echo -n "password" | base64` in case of Apple macOS) in your local shell to get valid values. For example, setting the PMM Server user's password to `new_password` in the `cluster1-pmm-secret` object can be done with the following command:

in Linux

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

in macOS

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

When done, apply the edited `deploy/cr.yaml` file:

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

2. 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
$ kubectl logs cluster1-7b7f7898d5-7f5pz -c pmm-client
```

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only ”ask me anything” sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-06-30

## 14.5 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.

### 14.5.1 Adding a sidecar container

You can add sidecar containers to PostgreSQL instance and pgBouncer Pods. Just use `sidecars` subsection in the `instances` or `proxy.pgBouncer` Custom Resource section in the `deploy/cr.yaml` configuration file. In this subsection, you should specify at least the name and image of your container, and possibly a command to run:

```
spec:
  instances:
    ....
    sidecars:
    - image: busybox
      command: ["/bin/sh"]
      args: ["-c", "while true; do echo echo $(date -u) 'test' >> /dev/null; sleep 5; done"]
      name: my-sidecar-1
    ....
```

Apply your modifications as usual:

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

**Note**

More options suitable for the `sidecars` subsection can be found in the Custom Resource options reference.

Running `kubectl describe` command for the appropriate Pod can bring you the information about the newly created container:

```
$ kubectl describe pod cluster1-instance1
```

**Expected output**

```
Name:            cluster1-instance1-n8v4-0
....
Containers:
....
my-sidecar-1:
  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)
....
```

## 14.5.2 Getting shell access to a sidecar container

You can login to your sidecar container as follows:

```
$ kubectl exec -it cluster1-instance1n8v4-0 -c my-sidecar-1 -- sh
/ #
```

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only ”ask me anything” sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-05-03

## 14.6 Pause/resume PostgreSQL Cluster

There may be external situations when it is needed to pause your 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 paused just revert the `spec.pause` key to `false`.

**Note**

There is an option also to put the cluster into a standby (read-only) mode instead of completely shutting it down. This is done by a special `spec.standby` key, which should be set to `true` for read-only state or should be set to `false` for normal cluster operation:

```
spec:
  .......
  standby: false
```

Last update: 2023-05-03

# 15. How to

## 15.1 How to deploy a standby cluster for Disaster Recovery

Disaster recovery is not optional for businesses operating in the digital age. With the ever-increasing reliance on data, system outages or data loss can be catastrophic, causing significant business disruptions and financial losses.

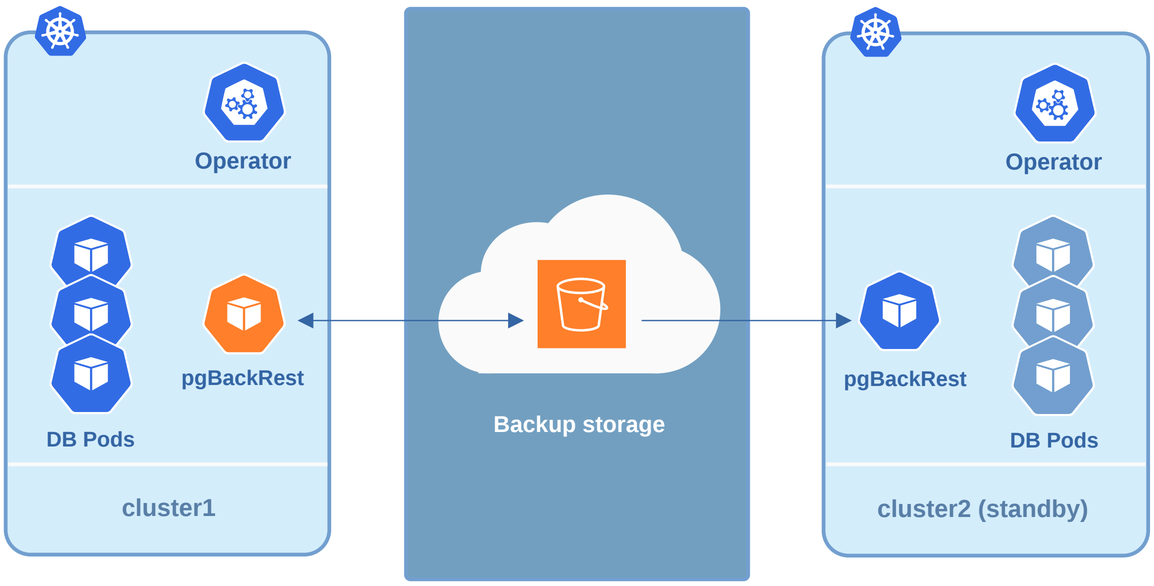
With multi-cloud or multi-regional PostgreSQL deployments, the complexity of managing disaster recovery only increases. This is where the Percona Operators come in, providing a solution to streamline disaster recovery for PostgreSQL clusters running on Kubernetes. With the Percona Operators, businesses can manage multi-cloud or hybrid-cloud PostgreSQL deployments with ease, ensuring that critical data is always available and secure, no matter what happens.

### 15.1.1 Solution overview

Operators automate routine tasks and remove toil. For standby, the Percona Operator for PostgreSQL version 2 provides the following options:

1. pgBackrest repo based standby
2. Streaming replication
3. Combination of (1) and (2)

This document describes the pgBackRest repo-based standby as the simplest one. The following is the architecture diagram:



1. This solution describes two Kubernetes clusters in different regions, clouds or running in hybrid mode (on-premises and cloud). One cluster is Main and the other is Disaster Recovery (DR)
2. Each cluster includes the following components:
   a. Percona Operator
   b. PostgreSQL cluster
   c. pgBackrest
   d. pgBouncer
3. pgBackrest on the Main site streams backups and Write Ahead Logs (WALs) to the object storage
4. pgBackrest on the DR site takes these backups and streams them to the standby cluster

### 15.1.2 Deploy disaster recovery for PostgreSQL on Kubernetes

**Configure Main site**

1. Deploy the Operator using your favorite method. Once installed, configure the Custom Resource manifest, so that pgBackrest starts using the Object Storage of your choice. Skip this step if you already have it configured.
2. Configure the `backups.pgbackrest.repos` section by adding the necessary configuration. The below example is for Google Cloud Storage (GCS):

```yaml
spec:
  backups:
    configuration:
      - secret:
          name: main-pgbackrest-secrets
    pgbackrest:
      repos:
      - name: repo1
        gcs:
          bucket: MY-BUCKET
```

   The `main-pgbackrest-secrets` value contains the keys for GCS. Read more about the configuration in the backup and restore tutorial.

3. Once configured, apply the custom resource:

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

   **Expected output**

   ```
   perconapgcluster.pg.percona.com/standby created
   ```

   The backups should appear in the object storage. By default pgBackrest puts them into the pgbackrest folder.

#### Configure DR site

The configuration of the disaster recovery site is similar to that of the Main site, with the only difference in standby settings.

The following manifest has `standby.enabled` set to `true` and points to the `repoName` where backups are (GCS in our case):

```
metadata:
  name: standby
spec:
...
  backups:
    configuration:
      - secret:
          name: standby-pgbackrest-secrets
    pgbackrest:
      repos:
      - name: repo1
        gcs:
          bucket: MY-BUCKET
  standby:
    enabled: true
    repoName: repo1
```

Deploy the standby cluster by applying the manifest:

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

**Expected output**

```
perconapgcluster.pg.percona.com/standby created
```

### 15.1.3 Failover

In case of the Main site failure or in other cases, you can promote the standby cluster. The promotion effectively allows writing to the cluster. This creates a net effect of pushing Write Ahead Logs (WALs) to the pgBackrest repository. It might create a split-brain situation where two primary instances attempt to write to the same repository. To avoid this, make sure the primary cluster is either deleted or shut down before trying to promote the standby cluster.

Once the primary is down or inactive, promote the standby through changing the corresponding section:

```
spec:
  standby:
    enabled: false
```

Now you can start writing to the cluster.

#### Split brain

There might be a case, where your old primary comes up and starts writing to the repository. To recover from this situation, do the following:

1. Keep only one primary with the latest data running
2. Stop the writes on the other one
3. Take the new full backup from the primary and upload it to the repo

#### Automate the failover

Automated failover consists of multiple steps and is outside of the Operator's scope. There are a few steps that you can take to reduce the Recovery Time Objective (RTO). To detect the failover we recommend having the 3rd site to monitor both DR and Main sites. In this case you can be sure that Main really failed and it is not a network split situation.

Another aspect of automation is to switch the traffic for the application from Main to Standby after promotion. It can be done through various Kubernetes configurations and heavily depends on how your networking and application are designed. The following options are quite common:

1. Global Load Balancer - various clouds and vendors provide their solutions
2. Multi Cluster Services or MCS - available on most of the public clouds
3. Federation or other multi-cluster solutions

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-06-30

## 15.2 Percona Operator for PostgreSQL single-namespace and multi-namespace deployment

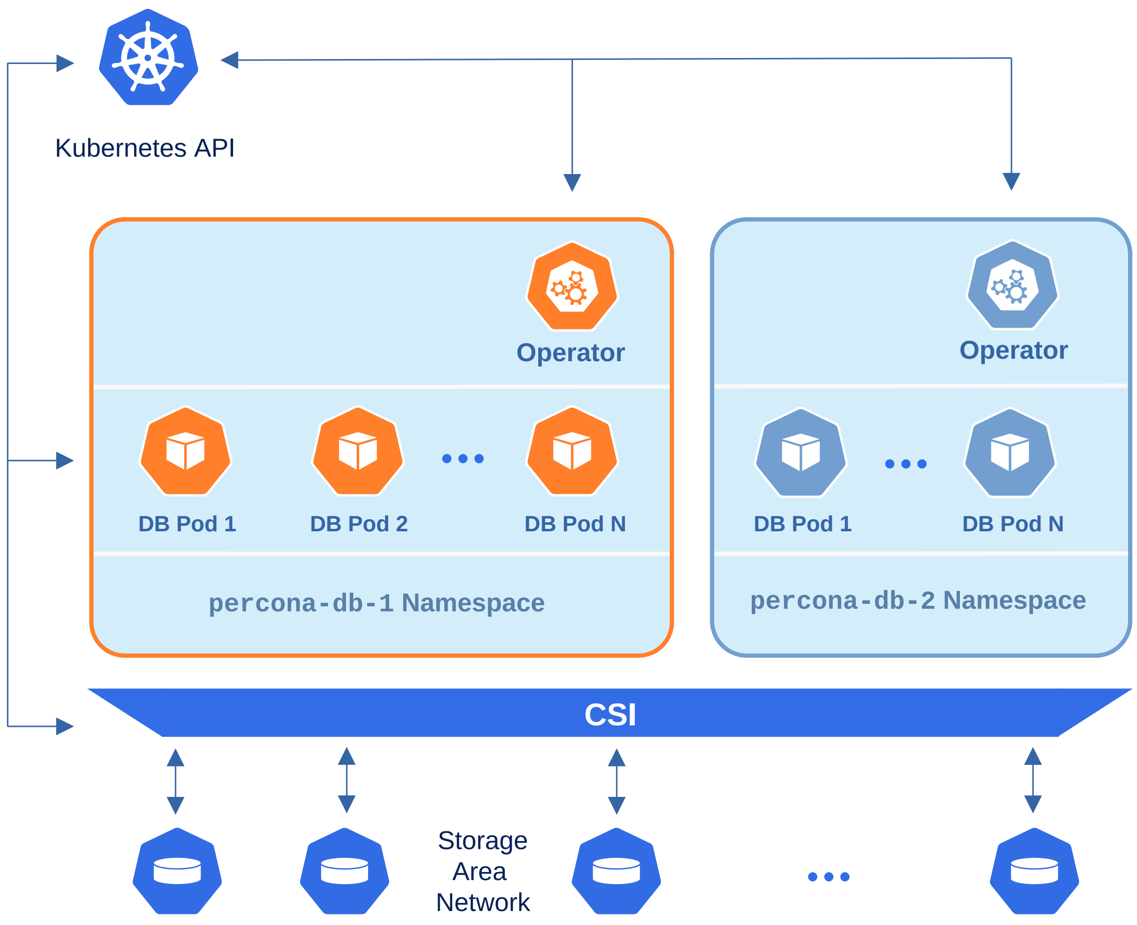
There are two design patterns that you can choose from when deploying Percona Operator for PostgreSQL and PostgreSQL clusters in Kubernetes:

- Namespace-scope - one Operator per Kubernetes namespace,
- Cluster-wide - one Operator can manage clusters in multiple namespaces.

This how-to explains how to configure Percona Operator for PostgreSQL for each scenario.

### 15.2.1 Namespace-scope

By default, Percona Operator for PostgreSQL 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:



Normally this is a recommended approach, as isolation minimizes impact in case of various failure scenarios. This is the default configuration of our Operator.

Let’s say you have a Namespace in your Kubernetes cluster called `percona-db-1`.

1. Create your `percona-db-1` namespace (if it doesn’t yet exist) as follows:

```
$ kubectl create namespace percona-db-1
```

2. Deploy the Operator:

```
$ kubectl apply -f deploy/operator.yaml -n percona-db-1
```

3. Once Operator is up and running, deploy the database cluster itself:

```
$ kubectl apply -f deploy/cr.yaml -n percona-db-1
```

You can deploy multiple clusters in this namespace.

#### Add more namespaces

What if there is a need to deploy clusters in another namespace? The solution for namespace-scope deployment is to have more than one Operator. We will use the `percona-db-2` namespace as an example.

1. Create your `percona-db-2` namespace (if it doesn’t yet exist) as follows:

```
$ kubectl create namespace percona-db-2
```

2. Deploy the Operator:

```
$ kubectl apply -f deploy/operator.yaml -n percona-db-2
```

3. Once Operator is up and running deploy the database cluster itself:

```
$ kubectl apply -f deploy/cr.yaml -n percona-db-2
```

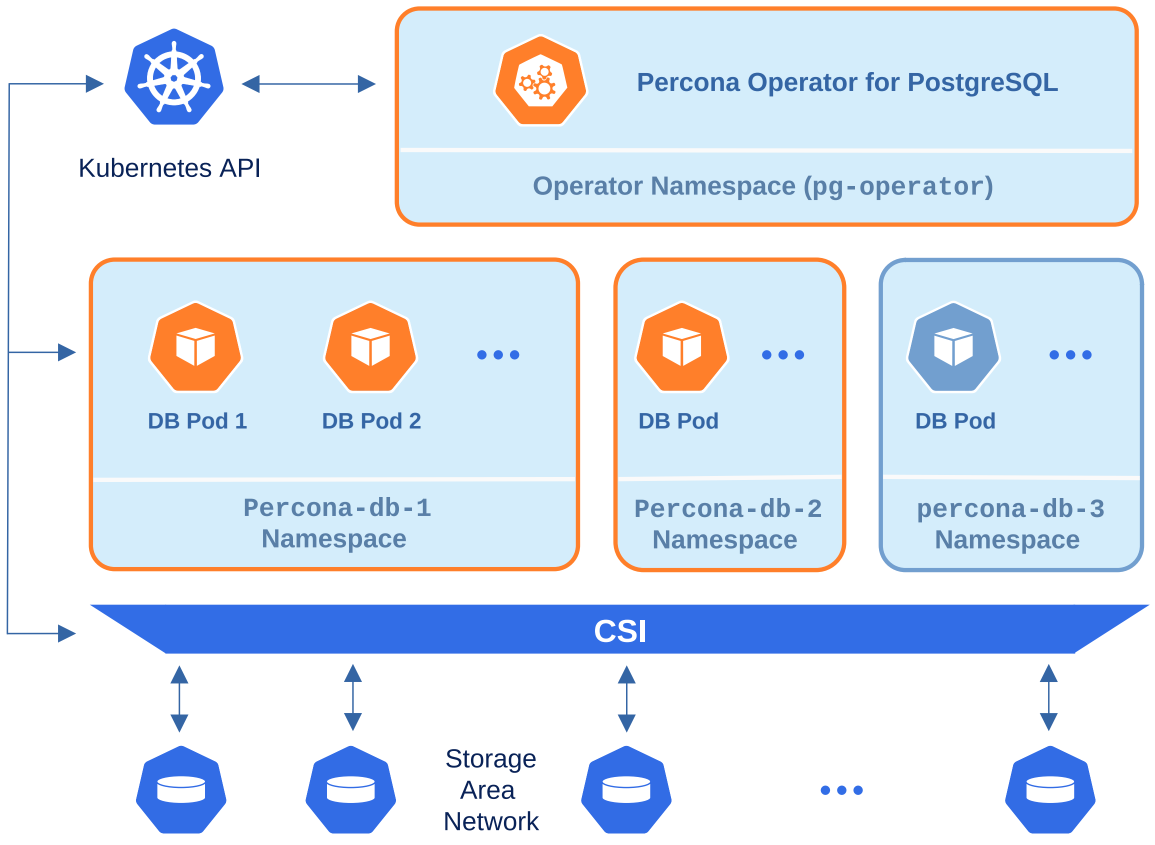
**Note**

Cluster names may be the same in different namespaces.

### 15.2.2 Install the Operator cluster-wide

Sometimes it is more convenient to have one Operator watching for Percona Distribution for PostgreSQL custom resources in several namespaces.

We recommend running Percona Operator for PostgreSQL in a traditional way, limited to a specific namespace, to limit the blast radius. But it is possible to run it in so-called *cluster-wide* mode, one Operator watching several namespaces, if needed:



To use the Operator in such cluster-wide mode, you should install it with a different set of configuration YAML files, which are available in the deploy folder and have filenames with a special `cw-` prefix: e.g. `deploy/cw-bundle.yaml`.

While using this cluster-wide versions of configuration files, you should set the following information there:

- `subjects.namespace` option should contain the namespace which will host the Operator,
- `WATCH_NAMESPACE` key-value pair in the `env` section should have `value` equal to a comma-separated list of the namespaces to be watched by the Operator, *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. Clone `percona-postgresql-operator` repository:

```
$ git clone -b v2.2.0 https://github.com/percona/percona-postgresql-operator
$ cd percona-postgresql-operator
```

2. Let's suppose that Operator's namespace should be the `pg-operator` one. Create it as follows:

```
$ kubectl create namespace pg-operator
```

3. Edit the `deploy/cw-bundle.yaml` configuration file to make sure it contains proper namespace name for the Operator:

```
...
subjects:
- kind: ServiceAccount
  name: percona-postgresql-operator
  namespace: pg-operator
...
```

4. Apply the `deploy/cw-bundle.yaml` file with the following command:

```
$ kubectl apply -f deploy/cw-bundle.yaml -n pg-operator
```

   Right now the operator deployed in cluster-wide mode will monitor all namespaces in the cluster, either already existing or newly created ones.

5. Create the namespace you have chosen for the cluster, if needed. let's call it `percona-db-1` for example:

```
$ kubectl create namespace percona-db-1
```

6. Deploy the cluster in the namespace of your choice:

```
$ kubectl apply -f deploy/cr.yaml -n percona-db-1
```

### 15.2.3 Verifying the cluster operation

When creation process is over, you can try to connect to the cluster.

1. During the installation, the Operator has generated several secrets, including the one with password for default PostgreSQL user. This default user has the login name same as the the cluster.

   Use `kubectl get secrets` command to see the list of Secrets objects. The Secrets object you are interested in is named as `<cluster_name>-pguser-<cluster_name>` (substitute `<cluster_name>` with the name of your Percona Distribution for PostgreSQL Cluster). The default variant will be `cluster1-pguser-cluster1`. You can use the following command to get the password of this user:

   ```
   $ kubectl get secret <cluster_name>-<user_name>-<cluster_name> --
   template='{{.data.password | base64decode}}{{"\n"}}'
   ```

2. Run a container with `psql` tool and connect its console output to your terminal. The following command will do this, naming the new Pod `pg-client`:

   ```
   $ kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-postgresql:
   15 --restart=Never -- bash -il
   [postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer.postgres-
   operator.svc -p 5432 -U cluster1 cluster1
   ```

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

   This command will connect you as a `cluster1` user to a `cluster1` database via the PostgreSQL interactive terminal.

   ```
   psql (15)
   SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256,
   compression: off)
   Type "help" for help.
   pgdb=>
   ```

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only ”ask me anything” sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-06-29

# 16. Troubleshooting

## 16.1 Initial troubleshooting

Percona Operator for PostgreSQL uses Custom Resources to manage options for the various components of the cluster.

- `PerconaPGCluster` Custom Resource with Percona PostgreSQL Cluster options (it has handy `pg` shortname also),
- `PerconaPGBackup` and `PerconaPGRestore` Custom Resources contain options for Percona XtraBackup used to backup Percona XtraDB Cluster and to restore it from backups ( `pg-backup` and `pg-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 pg
```

**Expected output**

```
NAME       ENDPOINT                         STATUS   POSTGRES   PGBOUNCER   AGE
cluster1   cluster1-pgbouncer.default.svc   ready    3          3           33d
```

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**

```
perconapgbackups          pg-backup    pg.percona.com/v2beta1           true
PerconaPGBackup
perconapgclusters         pg           pg.percona.com/v2beta1           true
PerconaPGCluster
perconapgrestores         pg-restore   pg.percona.com/v2beta1           true
PerconaPGRestore
```

### 16.1.1 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
cluster1-backup-4vwt-p5d9j                       0/1     Completed   0          97m
cluster1-instance1-b5mr-0                        4/4     Running     0          99m
cluster1-instance1-b8p7-0                        4/4     Running     0          99m
cluster1-instance1-w7q2-0                        4/4     Running     0          99m
cluster1-pgbouncer-79bbf55c45-62xlk              2/2     Running     0          99m
cluster1-pgbouncer-79bbf55c45-9g4cb              2/2     Running     0          99m
cluster1-pgbouncer-79bbf55c45-9nrmd              2/2     Running     0          99m
cluster1-repo-host-0                             2/2     Running     0          99m
percona-postgresql-operator-79cd8586f5-2qzcs     1/1     Running     0          120m
```

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, `cluster1-repo-host-0` container has all two containers ready (2/2). For an application to work properly, all containers of the Pod should be ready.
- `STATUS` indicates the current status of the Pod. The Pod should be in a `Running` state to confirm that the application is working as expected. You can find out other possible states in the official Kubernetes documentation.
- `RESTARTS` indicates how many times containers of Pod were restarted. This is impacted by the Container Restart Policy. In an ideal world, the restart count would be zero, meaning no issues from the beginning. If the restart count exceeds zero, it may be reasonable to check why it happens.
- `AGE` : Indicates how long the Pod is running. Any abnormality in this value needs to be checked.

You can find more details about a specific Pod using the `kubectl describe pods <pod-name>` command.

```
$ $ kubectl describe pods cluster1-instance1-b5mr-0
```

**Expected output**

```
...
Name:         cluster1-instance1-b5mr-0
Namespace:    default
...
Controlled By:  StatefulSet/cluster1-instance1-b5mr
Init Containers:
 postgres-startup:
...
Containers:
 database:
...
 pgbackrest:
...
   Restart Count:  0
   Liveness:   http-get https://:8008/liveness delay=3s timeout=5s period=10s #success=1
#failure=3
   Readiness:  http-get https://:8008/readiness delay=3s timeout=5s period=10s #success=1
#failure=3
   Environment:
...
   Mounts:
...
Volumes:
...
Events:
...
```

This gives a lot of information about containers, resources, container status and also events. So, describe output should be checked to see any abnormalities.

Last update: 2023-06-15

## 16.2 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 `database` of the `cluster1-instance1-b5mr-0` Pod.

- Run `date` command:

  ```
  $ kubectl exec -ti cluster1-instance1-b5mr-0 -c database -- date
  ```

  **Expected output**

  ```
  Wed Jun 14 11:18:47 UTC 2023
  ```

  You will see an error if the command is not present in a container. For example, trying to run the `time` command, which is not present in the container, by executing `kubectl exec -ti cluster1-instance1-b5mr-0 -c database -- 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 log files to a terminal:

  ```
  $ kubectl exec -ti cluster1-instance1-b5mr-0 -c database -- cat /pgdata/pg15/log/
  postgresql-*.log
  ```

- Similarly, opening an Interactive terminal, executing a pair of commands in the container, and exiting it may look as follows:

  ```
  $ kubectl exec -ti cluster1-instance1-b5mr-0 -c database -- bash
  bash-4.4$ hostname
  cluster1-pxc-0
  bash-4.4$ ls /pgdata/pg15/log/
  postgresql-Wed.log
  bash-4.4$ exit
  exit
  $
  ```

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-06-15

## 16.3 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 |
|---|---|
| `-c`, `--container=<container-name>` | Print log of a specific container in case of multiple containers in a Pod |
| `-f`, `--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) |
| `-p`, `--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 `cluster1-instance1-b5mr-0` Pod.

- Check logs of the `database` container:

```
$ kubectl logs cluster1-instance1-b5mr-0 --container database
```

- Check logs of the `pgbackrest` container:

```
$ kubectl logs cluster1-instance1-b5mr-0 --container pgbackrest
```

- Filter logs of the `database` container which are not older than 600 seconds:

```
$ kubectl logs cluster1-instance1-b5mr-0 --container database --since=600s
```

- Check logs of a previous instantiation of the `database` container, if any:

```
$ kubectl logs cluster1-instance1-b5mr-0 --container database --previous
```

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-06-30

# 17. Reference

## 17.1 Custom Resource options

The Cluster is configured via the deploy/cr.yaml file.

The metadata part of this file contains the following keys:

- `name` ( `cluster1` by default) sets the name of your Percona Distribution for PostgreSQL Cluster; it should include only URL-compatible characters, not exceed 22 characters, start with an alphabetic character, and end with an alphanumeric character;
- `finalizers.percona.com/delete-ssl` if present, activates the Finalizer which deletes objects, created for SSL (Secret, certificate, and issuer) after the cluster deletion event (off by default).
- `finalizers.percona.com/delete-pvc` if present, activates the Finalizer which deletes Persistent Volume Claims for Percona XtraDB Cluster Pods after the cluster deletion event (off by default).

The spec part of the deploy/cr.yaml file contains the following:

| | |
|---|---|
| **Key** | crVersion |
| **Value** | string |
| **Example** | `2.2.0` |
| **Description** | Version of the Operator the Custom Resource belongs to |

| | |
|---|---|
| **Key** | standby.enabled |
| **Value** | boolean |
| **Example** | `false` |
| **Description** | Enables or disables running the cluster in a standby mode (read-only copy of an existing cluster, useful for disaster recovery, etc) |

| | |
|---|---|
| **Key** | standby.host |
| **Value** | string |
| **Example** | `"<primary-ip>"` |
| **Description** | Host address of the primary cluster this standby cluster connects to |

| | |
|---|---|
| **Key** | standby.port |
| **Value** | string |
| **Example** | `"<primary-port>"` |
| **Description** | Port number used by a standby copy to connect to the primary cluster |
| **Key** | openshift |
| **Value** | boolean |
| **Example** | `true` |
| **Description** | Set to `true` if the cluster is being deployed on OpenShift, set to `false` otherwise, or unset it for autodetection |

| | |
|---|---|
| **Key** | standby.repoName |
| **Value** | string |
| **Example** | `repo1` |
| **Description** | Name of the pgBackRest repository in the primary cluster this standby cluster connects to |

| | |
|---|---|
| **Key** | secrets.customTLSSecret.name |
| **Value** | string |
| **Example** | `cluster1-cert` |
| **Description** | A secret with TLS certificate generated for *external* communications, see Transport Layer Security (TLS) for details |

| | |
|---|---|
| **Key** | secrets.customReplicationTLSSecret.name |
| **Value** | string |
| **Example** | `replication1-cert` |

| | |
|---|---|
| **Description** | A secret with TLS certificate generated for *internal* communications, see Transport Layer Security (TLS) for details |

| | |
|---|---|
| **Key** | users.name |
| **Value** | string |
| **Example** | `rhino` |
| **Description** | The name of the PostgreSQL user |

| | |
|---|---|
| **Key** | users.databases |
| **Value** | string |
| **Example** | `zoo` |
| **Description** | Databases accessible by a specific PostgreSQL user with rights to create objects in them (the option is ignored for `postgres` user; also, modifying it can't be used to revoke the already given access) |

| | |
|---|---|
| **Key** | users.password.type |
| **Value** | string |
| **Example** | `ASCII` |
| **Description** | The set of characters used for password generation: can be either `ASCII` (default) or `AlphaNumeric` |

| | |
|---|---|
| **Key** | users.options |
| **Value** | string |
| **Example** | `"SUPERUSER"` |
| **Description** | The `ALTER ROLE` options other than password (the option is ignored for `postgres` user) |

| | |
|---|---|
| **Key** | users.secretName |
| **Value** | string |
| **Example** | `"rhino-credentials"` |
| **Description** | The custom name of the user's Secret; if not specified, the default `<clusterName>-pguser-<userName>` variant will be used |

| | |
|---|---|
| **Key** | databaseInitSQL.key |
| **Value** | string |
| **Example** | `init.sql` |
| **Description** | Data key for the Custom configuration options ConfigMap with the init SQL file, which will be executed at cluster creation time |

| | |
|---|---|
| **Key** | databaseInitSQL.name |
| **Value** | string |
| **Example** | `cluster1-init-sql` |
| **Description** | Name of the ConfigMap with the init SQL file, which will be executed at cluster creation time |

| | |
|---|---|
| **Key** | pause |
| **Value** | string |
| **Example** | `false` |
| **Description** | Setting it to `true` gracefully stops the cluster, scaling workloads to zero and suspending CronJobs; setting it to `false` after shut down starts the cluster back |

| | |
|---|---|
| **Key** | unmanaged |
| **Value** | string |
| **Example** | `false` |
| **Description** | Setting it to `true` stops the Operator’s activity including the rollout and reconciliation of changes made in the Custom Resource; setting it to `false` starts the Operator’s activity back |

| | |
|---|---|
| **Key** | dataSource.postgresCluster.clusterName |
| **Value** | string |
| **Example** | `cluster1` |
| **Description** | Name of an existing cluster to use as the data source when restoring backup to a new cluster |

| | |
|---|---|
| **Key** | dataSource.postgresCluster.repoName |
| **Value** | string |
| **Example** | `repo1` |
| **Description** | Name of the pgBackRest repository in the source cluster that contains the backup to be restored to a new cluster |

| | |
|---|---|
| **Key** | dataSource.postgresCluster.options |
| **Value** | string |
| **Example** | |
| **Description** | The pgBackRest command-line options for the pgBackRest restore command |

| | |
|---|---|
| **Key** | dataSource.pgbackrest.stanza |
| **Value** | string |
| **Example** | `db` |
| **Description** | Name of the pgBackRest stanza to use as the data source when restoring backup to a new cluster |

| | |
|---|---|
| **Key** | dataSource.pgbackrest.configuration.secret.name |
| **Value** | string |
| **Example** | `pgo-s3-creds` |
| **Description** | Name of the Kubernetes Secret object with custom pgBackRest configuration, which will be added to the pgBackRest configuration generated by the Operator |

| | |
|---|---|
| **Key** | dataSource.pgbackrest.global |
| **Value** | subdoc |
| **Example** | `/pgbackrest/postgres-operator/hippo/repo1` |
| **Description** | Settings, which are to be included in the `global` section of the pgBackRest configuration generated by the Operator |

| | |
|---|---|
| **Key** | dataSource.pgbackrest.repo.name |
| **Value** | string |
| **Example** | `repo1` |
| **Description** | Name of the pgBackRest repository |

| | |
|---|---|
| **Key** | dataSource.pgbackrest.repo.s3.bucket |
| **Value** | string |
| **Example** | `"my-bucket"` |
| **Description** | The Amazon S3 bucket or Google Cloud Storage bucket |
| name used for backups | |

| | |
|---|---|
| **Key** | dataSource.pgbackrest.repo.s3.endpoint |
| **Value** | string |
| **Example** | `"s3.ca-central-1.amazonaws.com"` |
| **Description** | The endpoint URL of the S3-compatible storage to be used for backups (not needed for the original Amazon S3 cloud) |

| | |
|---|---|
| **Key** | dataSource.pgbackrest.repo.s3.region |
| **Value** | boolean |
| **Example** | `"ca-central-1"` |
| **Description** | The AWS region to use for Amazon and all S3-compatible storages |

| | |
|---|---|
| **Key** | image |
| **Value** | string |
| **Example** | `perconalab/percona-postgresql-operator:main-ppg14-postgres` |
| **Description** | The PostgreSQL Docker image to use |

| | |
|---|---|
| **Key** | imagePullPolicy |
| **Value** | string |
| **Example** | `Always` |
| **Description** | This option is used to set the policy for updating PostgreSQL images |

| | |
|---|---|
| **Key** | postgresVersion |
| **Value** | int |

| | |
|---|---|
| **Example** | 14 |
| **Description** | The major version of PostgreSQL to use |

| | |
|---|---|
| **Key** | port |
| **Value** | int |
| **Example** | 5432 |
| **Description** | The port number for PostgreSQL |

| | |
|---|---|
| **Key** | expose.annotations |
| **Value** | label |
| **Example** | `my-annotation: value1` |
| **Description** | The Kubernetes annotations metadata for PostgreSQL |

| | |
|---|---|
| **Key** | expose.labels |
| **Value** | label |
| **Example** | `my-label: value2` |
| **Description** | Set labels for the PostgreSQL Service |

| | |
|---|---|
| **Key** | expose.type |
| **Value** | string |
| **Example** | `LoadBalancer` |
| **Description** | Specifies the type of Kubernetes Service for PostgreSQL |

### 17.1.1 Instances section

The `instances` section in the deploy/cr.yaml file contains configuration options for PostgreSQL instances.

| | |
|---|---|
| **Key** | instances.name |
| **Value** | string |
| **Example** | `rs 0` |
| **Description** | The name of the PostgreSQL instance |

| | |
|---|---|
| **Key** | instances.replicas |
| **Value** | int |
| **Example** | 3 |
| **Description** | The number of Replicas to create for the PostgreSQL instance |

| | |
|---|---|
| **Key** | instances.resources.limits.cpu |
| **Value** | string |
| **Example** | `2.0` |
| **Description** | Kubernetes CPU limits for a PostgreSQL instance |

| | |
|---|---|
| **Key** | instances.resources.limits.memory |
| **Value** | string |
| **Example** | `4Gi` |
| **Description** | The Kubernetes memory limits for a PostgreSQL instance |

| | |
|---|---|
| **Key** | instances.topologySpreadConstraints.maxSkew |
| **Value** | int |
| **Example** | 1 |
| **Description** | The degree to which Pods may be unevenly distributed under the Kubernetes Pod Topology Spread Constraints |

| | |
|---|---|
| **Key** | instances.topologySpreadConstraints.topologyKey |
| **Value** | string |
| **Example** | `my-node-label` |
| **Description** | The key of node labels for the Kubernetes Pod Topology Spread Constraints |

| | |
|---|---|
| **Key** | instances.topologySpreadConstraints.whenUnsatisfiable |
| **Value** | string |
| **Example** | `DoNotSchedule` |
| **Description** | What to do with a Pod if it doesn't satisfy the Kubernetes Pod Topology Spread Constraints |

| | |
|---|---|
| **Key** | instances.topologySpreadConstraints.labelSelector.matchLabels |
| **Value** | label |
| **Example** | `postgres-operator.crunchydata.com/instance-set: instance1` |
| **Description** | The Label selector for the Kubernetes Pod Topology Spread Constraints |

| | |
|---|---|
| **Key** | instances.tolerations.effect |
| **Value** | string |
| **Example** | `NoSchedule` |
| **Description** | The Kubernetes Pod tolerations effect for the PostgreSQL instance |

| | |
|---|---|
| **Key** | instances.tolerations.key |
| **Value** | string |
| **Example** | `role` |
| **Description** | The Kubernetes Pod tolerations key for the PostgreSQL instance |

| | |
|---|---|
| **Key** | instances.tolerations.operator |
| **Value** | string |
| **Example** | `Equal` |
| **Description** | The Kubernetes Pod tolerations operator for the PostgreSQL instance |

| | |
|---|---|
| **Key** | instances.tolerations.value |
| **Value** | string |
| **Example** | `connection-poolers` |
| **Description** | The Kubernetes Pod tolerations value for the PostgreSQL instance |

| | |
|---|---|
| **Key** | instances.priorityClassName |
| **Value** | string |
| **Example** | `high-priority` |
| **Description** | The Kuberentes Pod priority class for PostgreSQL instance Pods |

| | |
|---|---|
| **Key** | instances.walVolumeClaimSpec.accessModes |
| **Value** | string |
| **Example** | `ReadWriteOnce` |
| **Description** | The Kubernetes PersistentVolumeClaim access modes for the PostgreSQL Write-ahead Log storage |

| | |
|---|---|
| **Key** | instances.walVolumeClaimSpec.resources.requests.storage |
| **Value** | string |
| **Example** | `1Gi` |
| **Description** | The Kubernetes storage requests for the storage the PostgreSQL instance will use |

| | |
|---|---|
| **Key** | instances.dataVolumeClaimSpec.accessModes |
| **Value** | string |
| **Example** | `ReadWriteOnce` |

| | |
|---|---|
| **Description** | The Kubernetes PersistentVolumeClaim access modes for the PostgreSQL Write-ahead Log storage |

| | |
|---|---|
| **Key** | instances.dataVolumeClaimSpec.resources.requests.storage |
| **Value** | string |
| **Example** | `1Gi` |
| **Description** | The Kubernetes storage requests for the storage the PostgreSQL instance will use |

### instances.sidecars subsection

The `instances.sidecars` subsection in the deploy/cr.yaml file contains configuration options for custom sidecar containers which can be added to PostgreSQL Pods.

| | |
|---|---|
| **Key** | instances.sidecars.image |
| **Value** | string |
| **Example** | `mycontainer1:latest` |
| **Description** | Image for the custom sidecar container for PostgreSQL Pods |

| | |
|---|---|
| **Key** | instances.sidecars.name |
| **Value** | string |
| **Example** | `testcontainer` |
| **Description** | Name of the custom sidecar container for PostgreSQL Pods |

| | |
|---|---|
| **Key** | instances.sidecars.imagePullPolicy |
| **Value** | string |
| **Example** | `Always` |
| **Description** | This option is used to set the policy for the PostgreSQL Pod sidecar container |

| | |
|---|---|
| **Key** | instances.sidecars.env |
| **Value** | subdoc |
| **Example** | |
| **Description** | The environment variables set as key-value pairs for the custom sidecar container for PostgreSQL Pods |

| | |
|---|---|
| **Key** | instances.sidecars.envFrom |
| **Value** | subdoc |
| **Example** | |
| **Description** | The environment variables set as key-value pairs in ConfigMaps for the custom sidecar container for PostgreSQL Pods |

| | |
|---|---|
| **Key** | instances.sidecars.command |
| **Value** | array |
| **Example** | `["/bin/sh"]` |
| **Description** | Command for the custom sidecar container for PostgreSQL Pods |

| | |
|---|---|
| **Key** | instances.sidecars.args |
| **Value** | array |
| **Example** | `["-c", "while true; do trap 'exit 0' SIGINT SIGTERM SIGQUIT SIGKILL; done;"]` |
| **Description** | Command arguments for the custom sidecar container for PostgreSQL Pods |

### 17.1.2 Backup section

The `backup` section in the deploy/cr.yaml file contains the following configuration options for the regular Percona Distribution for PostgreSQL backups.

| | |
|---|---|
| **Key** | backups.pgbackrest.image |
| **Value** | string |
| **Example** | `perconalab/percona-postgresql-operator:main-ppg14-pgbackrest` |
| **Description** | The Docker image for pgBackRest |

| | |
|---|---|
| **Key** | backups.pgbackrest.configuration.secret.name |
| **Value** | string |
| **Example** | `cluster1-pgbackrest-secrets` |
| **Description** | Name of the Kubernetes Secret object with custom pgBackRest configuration, which will be added to the pgBackRest configuration generated by the Operator |

| | |
|---|---|
| **Key** | backups.pgbackrest.jobs.priorityClassName |
| **Value** | string |
| **Example** | `high-priority` |
| **Description** | The Kuberentes Pod priority class for pgBackRest jobs |

| | |
|---|---|
| **Key** | backups.pgbackrest.jobs.resources.limits.cpu |
| **Value** | int |
| **Example** | `200` |
| **Description** | Kubernetes CPU limits for a pgBackRest job |

| | |
|---|---|
| **Key** | backups.pgbackrest.jobs.resources.limits.memory |
| **Value** | int |
| **Example** | `128Mi` |
| **Description** | The Kubernetes memory limits for a pgBackRest job |

| | |
|---|---|
| **Key** | backups.pgbackrest.jobs.tolerations.effect |
| **Value** | string |
| **Example** | `NoSchedule` |
| **Description** | The Kubernetes Pod tolerations effect for a pgBackRest job |

| | |
|---|---|
| **Key** | backups.pgbackrest.jobs.tolerations.key |
| **Value** | string |
| **Example** | `role` |
| **Description** | The Kubernetes Pod tolerations key for a pgBackRest job |

| | |
|---|---|
| **Key** | backups.pgbackrest.jobs.tolerations.operator |
| **Value** | string |
| **Example** | `Equal` |
| **Description** | The Kubernetes Pod tolerations operator for a pgBackRest job |

| | |
|---|---|
| **Key** | backups.pgbackrest.jobs.tolerations.value |
| **Value** | string |
| **Example** | `connection-poolers` |
| **Description** | The Kubernetes Pod tolerations value for a pgBackRest job |

| | |
|---|---|
| **Key** | backups.pgbackrest.global |
| **Value** | subdoc |
| **Example** | `/pgbackrest/postgres-operator/hippo/repo1` |
| **Description** | Settings, which are to be included in the `global` section of the pgBackRest configuration generated by the Operator |

| | |
|---|---|
| **Key** | backups.pgbackrest.repoHost.priorityClassName |
| **Value** | string |
| **Example** | `high-priority` |
| **Description** | The Kuberentes Pod priority class for pgBackRest repo |

| | |
|---|---|
| **Key** | backups.pgbackrest.repoHost.topologySpreadConstraints.maxSkew |
| **Value** | int |
| **Example** | 1 |
| **Description** | The degree to which Pods may be unevenly distributed under the Kubernetes Pod Topology Spread Constraints |

| | |
|---|---|
| **Key** | backups.pgbackrest.repoHost.topologySpreadConstraints.topologyKey |
| **Value** | string |
| **Example** | `my-node-label` |
| **Description** | The key of node labels for the Kubernetes Pod Topology Spread Constraints |

| | |
|---|---|
| **Key** | backups.pgbackrest.repoHost.topologySpreadConstraints.whenUnsatisfiable |
| **Value** | string |
| **Example** | `ScheduleAnyway` |
| **Description** | What to do with a Pod if it doesn't satisfy the Kubernetes Pod Topology Spread Constraints |

| | |
|---|---|
| **Key** | backups.pgbackrest.repoHost.topologySpreadConstraints.labelSelector.matchLabels |
| **Value** | label |
| **Example** | `postgres-operator.crunchydata.com/pgbackrest: ""` |
| **Description** | The Label selector for the Kubernetes Pod Topology Spread Constraints |

| | |
|---|---|
| **Key** | backups.pgbackrest.repoHost.affinity.podAntiAffinity |
| **Value** | subdoc |
| **Example** | |

| | |
|---|---|
| **Description** | Pod anti-affinity, allows setting the standard Kubernetes affinity constraints of any complexity |

| | |
|---|---|
| **Key** | backups.pgbackrest.manual.repoName |
| **Value** | string |
| **Example** | `repo1` |
| **Description** | Name of the pgBackRest repository for on-demand backups |

| | |
|---|---|
| **Key** | backups.pgbackrest.manual.options |
| **Value** | string |
| **Example** | `--type=full` |
| **Description** | The on-demand backup command-line options which will be passed to pgBackRest for on-demand backups |

| | |
|---|---|
| **Key** | backups.pgbackrest.repos.name |
| **Value** | string |
| **Example** | `repo1` |
| **Description** | Name of the pgBackRest repository for backups |

| | |
|---|---|
| **Key** | backups.pgbackrest.repos.schedules.full |
| **Value** | string |
| **Example** | `0 0 \* \* 6` |
| **Description** | Scheduled time to make a full backup specified in the crontab format |

| | |
|---|---|
| **Key** | backups.pgbackrest.repos.schedules.differential |
| **Value** | string |
| **Example** | `0 0 \* \* 6` |
| **Description** | Scheduled time to make a differential backup specified in the crontab format |

| | |
|---|---|
| **Key** | backups.pgbackrest.repos.volume.volumeClaimSpec.accessModes |
| **Value** | string |
| **Example** | `ReadWriteOnce` |
| **Description** | The Kubernetes PersistentVolumeClaim access modes for the pgBackRest Storage |

| | |
|---|---|
| **Key** | backups.pgbackrest.repos.volume.volumeClaimSpec.resources.requests.storage |
| **Value** | string |
| **Example** | `1Gi` |
| **Description** | The Kubernetes storage requests for the pgBackRest storage |

| | |
|---|---|
| **Key** | backups.pgbackrest.repos.s3.bucket |
| **Value** | string |

| | |
|---|---|
| **Example** | `"my-bucket"` |
| **Description** | The Amazon S3 bucket |
| name used for backups | |

| | |
|---|---|
| **Key** | backups.pgbackrest.repos.s3.endpoint |
| **Value** | string |
| **Example** | `"s3.ca-central-1.amazonaws.com"` |
| **Description** | The endpoint URL of the S3-compatible storage to be used for backups (not needed for the original Amazon S3 cloud) |

| | |
|---|---|
| **Key** | backups.pgbackrest.repos.s3.region |
| **Value** | boolean |
| **Example** | `"ca-central-1"` |
| **Description** | The AWS region to use for Amazon and all S3-compatible storages |

| | |
|---|---|
| **Key** | backups.pgbackrest.repos.gcs.bucket |
| **Value** | string |
| **Example** | `"my-bucket"` |
| **Description** | The Google Cloud Storage bucket |
| name used for backups | |

| | |
|---|---|
| **Key** | backups.pgbackrest.repos.azure.container |
| **Value** | string |
| **Example** | `my-container` |
| **Description** | Name of the Azure Blob Storage container for backups |

| | |
|---|---|
| **Key** | backups.restore.enabled |
| **Value** | boolean |
| **Example** | `false` |
| **Description** | Enables or disables restoring a previously made backup |

| | |
|---|---|
| **Key** | backups.restore.repoName |
| **Value** | string |
| **Example** | `repo1` |
| **Description** | Name of the pgBackRest repository that contains the backup to be restored |

| | |
|---|---|
| **Key** | backups.restore.options |
| **Value** | string |
| **Example** | |

| | |
|---|---|
| **Description** | The pgBackRest command-line options for the pgBackRest restore command |

### 17.1.3 PMM section

The `pmm` section in the deploy/cr.yaml file contains configuration options for Percona Monitoring and Management.

| | |
|---|---|
| **Key** | pmm.enabled |
| **Value** | boolean |
| **Example** | `false` |
| **Description** | Enables or disables monitoring Percona Distribution for PostgreSQL cluster with PMM |

| | |
|---|---|
| **Key** | pmm.image |
| **Value** | string |
| **Example** | `percona/pmm-client:2.37.0` |
| **Description** | Percona Monitoring and Management (PMM) Client Docker image |

| | |
|---|---|
| **Key** | pmm.imagePullPolicy |
| **Value** | string |
| **Example** | `IfNotPresent` |
| **Description** | This option is used to set the policy for updating PMM Client images |

| | |
|---|---|
| **Key** | pmm.pmmSecret |
| **Value** | string |
| **Example** | `cluster1-pmm-secret` |
| **Description** | Name of the Kubernetes Secret object for the PMM Server password |

| | |
|---|---|
| **Key** | pmm.serverHost |
| **Value** | string |
| **Example** | `monitoring-service` |
| **Description** | Address of the PMM Server to collect data from the cluster |

### 17.1.4 Proxy section

The `proxy` section in the deploy/cr.yaml file contains configuration options for the pgBouncer connection pooler for PostgreSQL.

| | |
|---|---|
| **Key** | proxy.pgBouncer.replicas |
| **Value** | int |
| **Example** | `3` |
| **Description** | The number of the pgBouncer Pods to provide connection pooling |

| | |
|---|---|
| **Key** | proxy.pgBouncer.image |
| **Value** | string |
| **Example** | `perconalab/percona-postgresql-operator:main-ppg14-pgbouncer` |
| **Description** | Docker image for the pgBouncer connection pooler |

| | |
|---|---|
| **Key** | proxy.pgBouncer.exposeSuperusers |
| **Value** | boolean |
| **Example** | `false` |
| **Description** | Enables or disables exposing superuser user through pgBouncer |

| | |
|---|---|
| **Key** | proxy.pgBouncer.resources.limits.cpu |
| **Value** | int |
| **Example** | `200m` |
| **Description** | Kubernetes CPU limits for a pgBouncer container |

| | |
|---|---|
| **Key** | proxy.pgBouncer.resources.limits.memory |
| **Value** | int |
| **Example** | `128Mi` |
| **Description** | The Kubernetes memory limits for a pgBouncer container |

| | |
|---|---|
| **Key** | proxy.pgBouncer.expose.type |
| **Value** | string |
| **Example** | `ClusterIP` |
| **Description** | Specifies the type of Kubernetes Service for pgBouncer |

| | |
|---|---|
| **Key** | proxy.pgBouncer.expose.annotations |
| **Value** | label |
| **Example** | `pg-cluster-annot: cluster1` |
| **Description** | The Kubernetes annotations metadata for pgBouncer |

| | |
|---|---|
| **Key** | proxy.pgBouncer.expose.labels |
| **Value** | label |
| **Example** | `pg-cluster-label: cluster1` |
| **Description** | Set labels for the pgBouncer Service |

| | |
|---|---|
| **Value** | string |
| **Example** | `preferred` |
| **Description** | Pod anti-affinity type, can be either `preferred` or `required` |

| | |
|---|---|
| **Key** | proxy.pgBouncer.config |
| **Value** | subdoc |
| **Example** | `global:`<br>`pool_mode: transaction` |
| **Description** | Custom configuration options for pgBouncer. Please note that configuration changes are automatically applied to the running instances without validation, so having an invalid config can make the cluster unavailable |

### proxy.pgBouncer.sidecars subsection

The `proxy.pgBouncer.sidecars` subsection in the deploy/cr.yaml file contains configuration options for custom sidecar containers which can be added to pgBouncer Pods.

| | |
|---|---|
| **Key** | proxy.pgBouncer.sidecars.image |
| **Value** | string |
| **Example** | `mycontainer1:latest` |
| **Description** | Image for the custom sidecar container for pgBouncer Pods |

| | |
|---|---|
| **Key** | proxy.pgBouncer.sidecars.name |
| **Value** | string |
| **Example** | `testcontainer` |
| **Description** | Name of the custom sidecar container for pgBouncer Pods |

| | |
|---|---|
| **Key** | proxy.pgBouncer.sidecars.imagePullPolicy |
| **Value** | string |
| **Example** | `Always` |
| **Description** | This option is used to set the policy for the pgBouncer Pod sidecar container |

| | |
|---|---|
| **Key** | proxy.pgBouncer.sidecars.env |
| **Value** | subdoc |
| **Example** | |
| **Description** | The environment variables set as key-value pairs for the custom sidecar container for pgBouncer Pods |

| | |
|---|---|
| **Key** | proxy.pgBouncer.sidecars.envFrom |
| **Value** | subdoc |
| **Example** | |
| **Description** | The environment variables set as key-value pairs in ConfigMaps for the custom sidecar container for pgBouncer Pods |

| | |
|---|---|
| **Key** | proxy.pgBouncer.sidecars.command |
| **Value** | array |
| **Example** | `["/bin/sh"]` |
| **Description** | Command for the custom sidecar container for pgBouncer Pods |

| | |
|---|---|
| **Key** | proxy.pgBouncer.sidecars.args |
| **Value** | array |
| **Example** | `["-c", "while true; do trap 'exit 0' SIGINT SIGTERM SIGQUIT SIGKILL; done;"]` |
| **Description** | Command arguments for the custom sidecar container for pgBouncer Pods |

## 17.1.5 Patroni Section

The `patroni` section in the deploy/cr.yaml file contains configuration options to customize the PostgreSQL high-availability implementation based on Patroni.

| | |
|---|---|
| **Key** | patroni.dynamicConfiguration |
| **Value** | subdoc |
| **Example** | `postgresql:`<br>`  parameters:`<br>`    max_parallel_workers: 2`<br>`    max_worker_processes: 2`<br>`    shared_buffers: 1GB`<br>`    work_mem: 2MB` |
| **Description** | Custom PostgreSQL configuration options. Please note that configuration changes are automatically applied to the running instances without validation, so having an invalid config can make the cluster unavailable |

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only ”ask me anything” sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-06-30

## 17.2 Percona certified images

Following table presents Percona's certified docker images to be used with the Percona Operator for PostgreSQL:

| Image | Digest |
|---|---|
| percona/percona-postgresql-operator:2.2.0 | c4b300498b319e6458f98310a3804d08af6680c9ce76ea64baecc57917838be4 |
| percona/percona-postgresql-operator:2.2.0-ppg12-postgres | 20703450ccc4f1d020b54b22809ca0814b16c49d0894d73cdaafccf31beb6a26 |
| percona/percona-postgresql-operator:2.2.0-ppg13-postgres | c6dea3348111b24036e1e88b8efd5a9af94934793ee2d763c20f320100b22e45 |
| percona/percona-postgresql-operator:2.2.0-ppg14-postgres | 5f628a8a98c4ce2c25ae2b5db53c1515d94d86edef1e97b48c0059fdaff60f22 |
| percona/percona-postgresql-operator:2.2.0-ppg15-postgres | 762874bd5dab6984942c96bb31bcfda5348e7a511f7383f282620bff48669f5f |
| percona/percona-postgresql-operator:2.2.0-ppg12-pgbouncer | 67fb5808461da6fead4a4608abab4ee903630c0944a86ff2c513cdf1038d393d |
| percona/percona-postgresql-operator:2.2.0-ppg13-pgbouncer | 80f0ad8c0edd902f799f148333a854e4b9ecbbcf034e6eeff4c49eca25e08e94 |
| percona/percona-postgresql-operator:2.2.0-ppg14-pgbouncer | ad46d14445ff0980f613f6ae89d9c6400715b756f52f4d3268c93e99ed12514f |
| percona/percona-postgresql-operator:2.2.0-ppg15-pgbouncer | b704485b4b39205719edad6042527c45ccdcf7e2f2d4d56eda4c15819db053a4 |
| percona/percona-postgresql-operator:2.2.0-ppg12-pgbackrest | 292854a005d350abb22bbe9c1d5ecaaf5f807dc05de01a94e5c1cca675ea331d |
| percona/percona-postgresql-operator:2.2.0-ppg13-pgbackrest | 0c38e1356cf06b318bd73ae3d6947a79535816b5a6d311f2ca8fcc945edb92a1 |
| percona/percona-postgresql-operator:2.2.0-ppg14-pgbackrest | 943d46fb2162760d884e14888f70ec168b62e21b728dc9ed4a49004a89009484 |
| percona/percona-postgresql-operator:2.2.0-ppg15-pgbackrest | e444b4296be43d17a6d7d6b1be48fc95312700ba16efd8662f4f04f6618a1230 |
| percona/pmm-client:2.37.0 | e1e2f4cbfd4ce4be5d883330d810e9962a62531e2da07f1b115077a49ff97ed5 |

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only ”ask me anything” sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-06-30

## 17.3 Copyright and licensing information

### 17.3.1 Documentation licensing

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-06-27

## 17.4 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.

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only ”ask me anything” sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

---

Last update: 2023-06-27

# 18. Release Notes

## 18.1 Percona Operator for PostgreSQL Release Notes

Last update: 2023-06-30

## 18.2 Percona Operator for PostgreSQL 2.2.0

- **Date**

  June 30, 2023

- **Installation**

  Installing Percona Operator for PostgreSQL

**Percona announces the general availability of Percona Operator for PostgreSQL 2.2.0.**

Starting with this release, Percona Operator for PostgreSQL version 2 is out of technical preview and can be used in production with all the improvements it brings over the version 1 in terms of architecture, backup and recovery features, and overall flexibility.

We prepared a detailed migration guide which allows existing Operator 1.x users to move their PostgreSQL clusters to the Operator 2.x. Also, see this blog post to find out more about the Operator 2.x features and benefits.

### 18.2.1 Improvements

- K8SPG-378: A new `crVersion` Custom Resource option was added to indicate the API version this Custom Resource corresponds to
- K8SPG-359: The new `users.secretName` option allows to define a custom Secret name for the users defined in the Custom Resource (thanks to Vishal Anarase for contributing)
- K8SPG-301: Amazon Elastic Container Service for Kubernetes (EKS) was added to the list of officially supported platforms
- K8SPG-302: Minikube is now officially supported by the Operator to enable ease of testing and developing
- K8SPG-326: Both the Operator and database can be now installed with the Helm package manager
- K8SPG-342: There is now no need in manual restart of PostgreSQL Pods after the monitor user password changed in Secrets
- K8SPG-345: The new `proxy.pgBouncer.exposeSuperusers` Custom Resource option makes it possible for administrative users to connect to PostgreSQL through PgBouncer
- K8SPG-355: The Operator can now be deployed in multi-namespace ("cluster-wide") mode to track Custom Resources and manage database clusters in several namespaces

### 18.2.2 Bugs Fixed

- K8SPG-373: Fix the bug due to which the Operator did not not create Secrets for the `pguser` user if PMM was enabled in the Custom Resource
- K8SPG-362: It was impossible to install Custom Resource Definitions for both 1.x and 2.x Operators in one environment, preventing the migration of a cluster to the newer Operator version
- K8SPG-360: Fix a bug due to which manual password changing or resetting via Secret didn't work

Known limitations

- Query analytics (QAN) will not be available in Percona Monitoring and Management (PMM) due to bugs PMM-12024 and PMM-11938. The fixes are included in the upcoming PMM 2.38, so QAN can be used as soon as it is released and both PMM Client and PMM Server are upgraded.

### 18.2.3 Supported platforms

The Operator was developed and tested with PostgreSQL versions 12.14, 13.10, 14.7, and 15.2. Other options may also work but have not been tested. The Operator 2.2.0 provides connection pooling based on pgBouncer 1.18.0 and high-availability implementation based on Patroni 3.0.1.

The following platforms were tested and are officially supported by the Operator 2.2.0:

- Google Kubernetes Engine (GKE) 1.23 - 1.26
- Amazon Elastic Container Service for Kubernetes (EKS) 1.23 - 1.27
- Minikube 1.30.1 (based on Kubernetes 1.27)

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.

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-06-30

## 18.3 Percona Operator for PostgreSQL 2.1.0 (Tech preview)

- **Date**

  May 4, 2023
- **Installation**

  Installing Percona Operator for PostgreSQL

The Percona Operator built with best practices of configuration and setup of Percona Distribution for PostgreSQL on Kubernetes.

Percona Operator for PostgreSQL helps create and manage highly available, enterprise-ready PostgreSQL clusters on Kubernetes. It is 100% open source, free from vendor lock-in, usage restrictions and expensive contracts, and includes enterprise-ready features: backup/restore, high availability, replication, logging, and more.

The benefits of using Percona Operator for PostgreSQL include saving time on database operations via automation of Day-1 and Day-2 operations and deployment of consistent and vetted environment on Kubernetes.

**Note**

Version 2.1.0 of the Percona Operator for PostgreSQL is a **tech preview release** and it is **not recommended for production environments.** As of today, we recommend using Percona Operator for PostgreSQL 1.x, which is production-ready and contains everything you need to quickly and consistently deploy and scale PostgreSQL clusters in a Kubernetes-based environment, on-premises or in the cloud.

### 18.3.1 Release Highlights

- PostgreSQL 15 is now officially supported by the Operator with the new exciting features it brings to developers
- UX improvements related to Custom Resource have been added in this release, including the handy `pg`, `pg-backup`, and `pg-restore` short names useful to quickly query the cluster state with the `kubectl get` command and additional information in the status fields, which now show `name`, `endpoint`, `status`, and `age`

### 18.3.2 New Features

- K8SPG-328: The new `delete-pvc` finalizer allows to either delete or preserve Persistent Volumes at Custom Resource deletion
- K8SPG-330: The new `delete-ssl` finalizer can now be used to automatically delete objects created for SSL (Secret, certificate, and issuer) in case of cluster deletion
- K8SPG-331: Starting from now, the Operator adds short names to its Custom Resources: `pg`, `pg-backup`, and `pg-restore`
- K8SPG-282: PostgreSQL 15 is now officially supported by the Operator

### 18.3.3 Improvements

- K8SPG-262: The Operator now does not attempt to start Percona Monitoring and Management (PMM) client if the corresponding secret does not contain the `pmmserver` or `pmmserverkey` key
- K8SPG-285: To improve the Operator we capture anonymous telemetry and usage data. In this release we add more data points to it

- K8SPG-295: Additional information was added to the status of the Operator Custom Resource, which now shows `name`, `endpoint`, `status`, and `age` fields
- K8SPG-304: The Operator stops using trust authentication method in `pg_hba.conf` for better security
- K8SPG-325: Custom Resource options previously named `paused` and `shutdown` were renamed to `unmanaged` and `pause` for better alignment with other Percona Operators

### 18.3.4 Bugs Fixed

- K8SPG-272: Fix a bug due to which PMM agent related to the Pod wasn't deleted from the PMM Server inventory on Pod termination
- K8SPG-279: Fix a bug which made the Operator to crash after creating a backup if there was no `backups.pgbackrest.manual` section in the Custom Resource
- K8SPG-298: Fix a bug due to which the `shutdown` Custom Resource option didn't work making it impossible to pause the cluster
- K8SPG-334: Fix a bug which made it possible for the monitoring user to have special characters in the autogenerated password, making it incompatible with the PMM Client

### 18.3.5 Supported platforms

The following platforms were tested and are officially supported by the Operator 2.1.0:

- Google Kubernetes Engine (GKE) 1.23 - 1.25
- Amazon Elastic Container Service for Kubernetes (EKS) 1.23 - 1.25

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.

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only "ask me anything" sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2023-05-04

## 18.4 Percona Operator for PostgreSQL 2.0.0 (Tech preview)

- **Date**

  December 30, 2022

- **Installation**

  Installing Percona Operator for PostgreSQL

The Percona Operator is based on best practices for configuration and setup of a Percona Distribution for PostgreSQL on Kubernetes. The benefits of the Operator are many, but saving time and delivering a consistent and vetted environment is key.

**Note**

Version 2.0.0 of the Percona Operator for PostgreSQL is a **tech preview release** and it is **not recommended for production environments.** As of today, we recommend using Percona Operator for PostgreSQL 1.x, which is production-ready and contains everything you need to quickly and consistently deploy and scale PostgreSQL clusters in a Kubernetes-based environment, on-premises or in the cloud.

The *Percona Operator for PostgreSQL 2.x* is based on the 5.x branch of the Postgres Operator developed by Crunchy Data. Please see the main changes in this version below.

### 18.4.1 Architecture

Operator SDK is now used to build and package the Operator. It simplifies the development and brings more contribution friendliness to the code, resulting in better potential for growing the community. Users now have full control over Custom Resource Definitions that Operator relies on, which simplifies the deployment and management of the operator.

In version 1.x we relied on Deployment resources to run PostgreSQL clusters, whereas in 2.0 Statefulsets are used, which are the de-facto standard for running stateful workloads in Kubernetes. This change improves stability of the clusters and removes a lot of complexity from the Operator.

### 18.4.2 Backups

One of the biggest challenges in version 1.x is backups and restores. There are two main problems that our user faced:

- Not possible to change backup configuration for the existing cluster
- Restoration from backup to the newly deployed cluster required workarounds

In this version both these issues are fixed. In addition to that:

- Run up to 4 pgBackrest repositories
- Bootstrap the cluster from the existing backup through Custom Resource
- Azure Blob Storage support

### 18.4.3 Operations

Deploying complex topologies in Kubernetes is not possible without affinity and anti-affinity rules. In version 1.x there were various limitations and issues, whereas this version comes with substantial improvements that enables users to craft the topology of their choice.

Within the same cluster users can deploy multiple instances. These instances are going to have the same data, but can have different configuration and resources. This can be useful if you plan to migrate to new hardware or need to test the new topology.

Each postgreSQL node can have sidecar containers now to provide integration with your existing tools or expand the capabilities of the cluster.

### 18.4.4 Try it out now

Excited with what you read above?

- We encourage you to install the Operator following our documentation.
- Feel free to share feedback with us on the forum or raise a bug or feature request in JIRA.
- See the source code in our Github repository.

CONTACT US

For free technical help, visit the Percona Community Forum.

To get early access to new product features, invite-only ”ask me anything” sessions with Percona Kubernetes experts, and monthly swag raffles, join K8S Squad.

To report bugs or submit feature requests, open a JIRA ticket.

For paid support and managed or consulting services , contact Percona Sales.

Last update: 2022-12-30