PERCONA

Operator for PostgreSQL 2.5.1

(March 03, 2025)

Documentation

Table of Contents

Percona Operator for PostgreSQL documentation

The Percona Operator for PostgreSQL automates the creation, modification, or deletion of items in your Percona Distribution for PostgreSQL environment. The Operator contains the necessary Kubernetes settings to maintain a consistent PostgreSQL cluster.

Percona Kubernetes Operator 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.

This is the documentation for the latest release, **2.5.1** (Release Notes).

Starting with Percona Kubernetes Operator is easy. Follow our documentation guides, and you'll be set up in a minute.

Installation guides

Want to see it for yourself? Get started quickly with our step-by-step installation instructions.

Quickstart guides →

Security and encryption

Rest assured! Learn more about our security features designed to protect your valuable data.

Security measures →

Backup management

Learn what you can do to maintain regular backups of your PostgrgeSQL cluster.

Backup management →

Troubleshooting

Our comprehensive resources will help you overcome challenges, from everyday issues to specific doubts.

Diagnostics →

About

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

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 - 16	14 - 16	13 - 16	12 - 16	11 - 15
Kubernetes conformance	Various versions are tested	Various versions are tested	Various versions are tested	Various versions are tested	AWS EKS

Maintenance

Feature/ Product	Percona Operator for PostgreSQL	Stackgres	CrunchyData	CloudNativePG (EDB)	Zalando
Operator upgrade	✅	✅	✅	✅	✅

Feature/ Product	Percona Operator for PostgreSQL	Stackgres	CrunchyData	CloudNativePG (EDB)	Zalando
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

PostgreSQL topologies

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

Backups

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

Feature/ Product	Percona Operator for PostgreSQL	Stackgres	CrunchyData	CloudNativePG (EDB)	Zalando
GCS	✅	✅	✅	✅	✅
S3	✅	✅	✅	✅	✅
Azure	✅	✅	✅	✅	✅

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

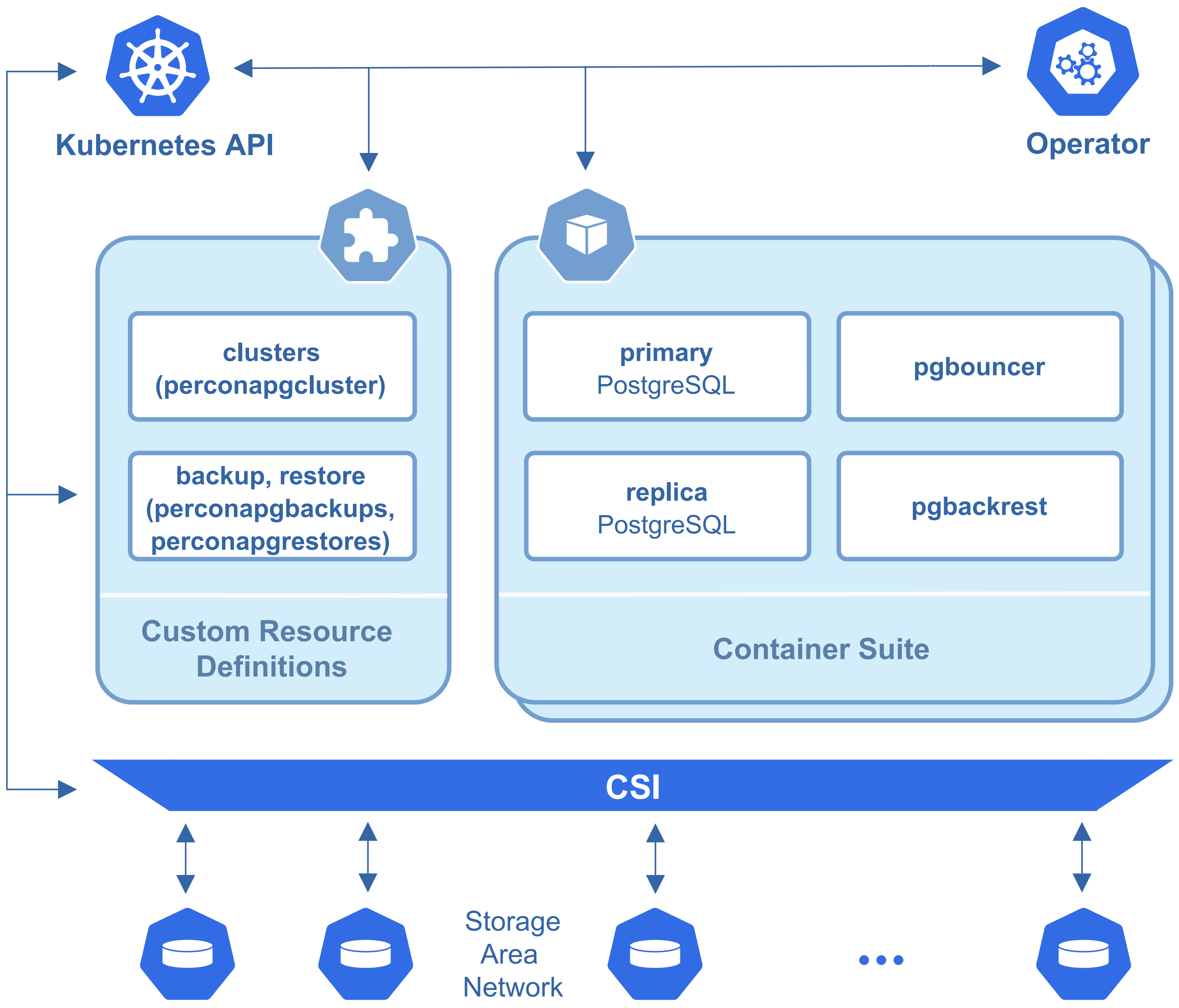
Miscellaneous

Feature/Product	Percona Operator for PostgreSQL	Stackgres	CrunchyData	CloudNativePG (EDB)	Zalando
Customize PostgreSQL configuration	✅	✅	✅	✅	✅
Sidecar containers for customization	✅	🚫	✅	🚫	✅
Helm	✅	✅	✅	✅	✅
Transport encryption	✅	✅	✅	✅	✅
Data-at-rest encryption	Through storage class	Through storage class	Through storage class	Through storage class	Through storage class

Feature/Product	Percona Operator for PostgreSQL	Stackgres	CrunchyData	CloudNativePG (EDB)	Zalando
Create users/roles	✅	✅	✅	✅	limited

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.



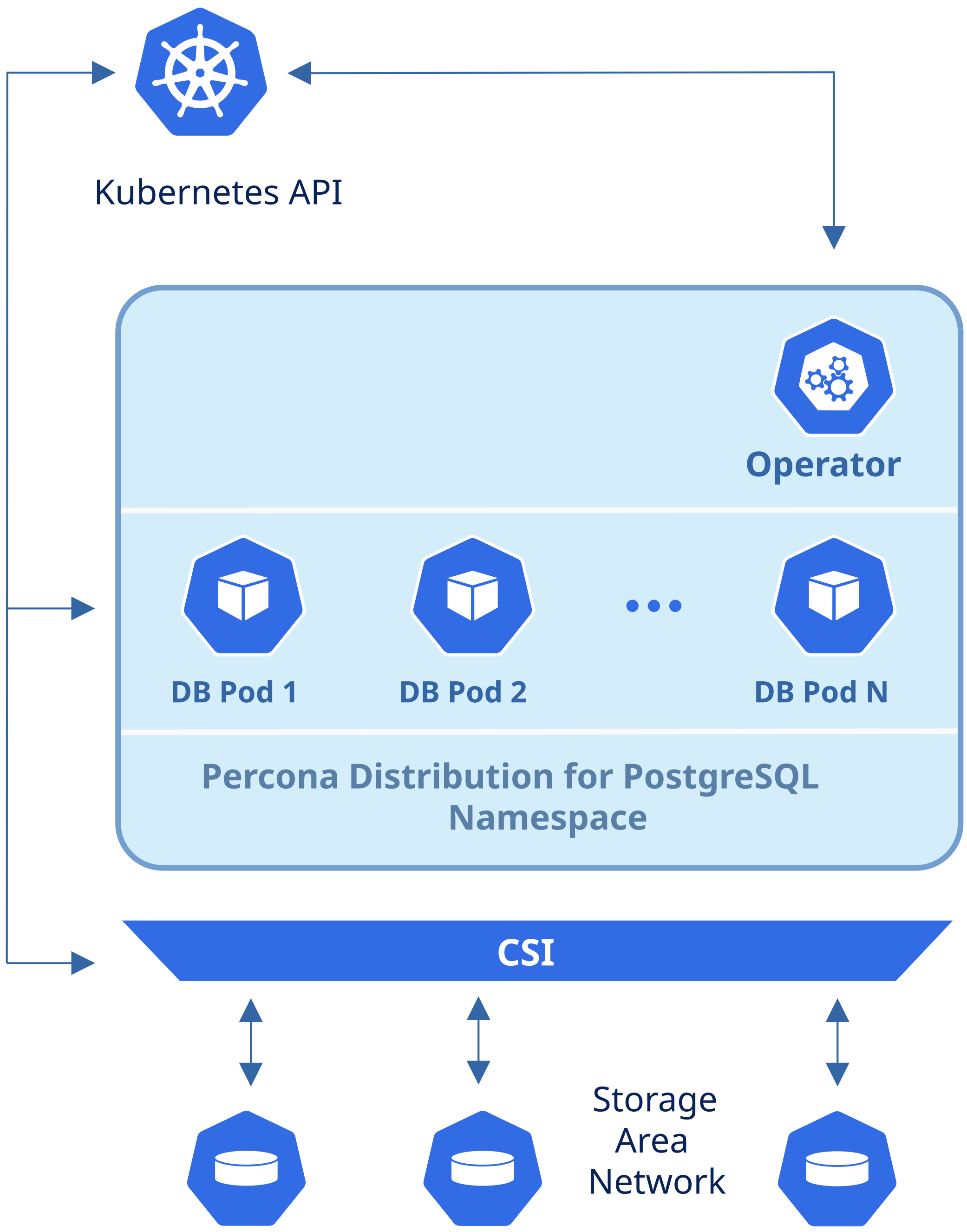
PostgreSQL containers deployed with the Operator include the following components:

- 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).

Each PostgreSQL cluster includes one member availiable for read/write transactions (PostgreSQL primary instance, or leader in terms of Patroni) and a number of replicas which can serve read requests only (standby members of the cluster).

To provide high availability from the Kubernetes side 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.

Overview

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

- Install and deploy an Operator
- Connect to PostgreSQL
- Insert sample data to the database
- Set up and make a manual backup
- Monitor the database health with PMM

Next steps

Install the Operator →

Quickstart guide

Install Percona Distribution for PostgreSQL using kubectl

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

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

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

Install with kubectl ↓ Install with Helm →

Prerequisites

To install Percona Distribution for PostgreSQL, you need the following:

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

See also

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

Procedure

Here's a sequence of steps to follow:

1. Create the Kubernetes namespace for your cluster. It is a good practice to isolate workloads in Kubernetes by installing the Operator in a custom namespace. For example, let's name it `postgres-operator`:

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

Expected output

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

We will use this namespace further on in this document. If you used another name, make sure to replace it in the following commands.

2. Deploy the Operator using the following command:

```
$ kubectl apply --server-side -f https://raw.githubusercontent.com/percona/
percona-postgresql-operator/v2.5.1/deploy/bundle.yaml -n postgres-operator
```

Expected output

```
customresourcedefinition.apiextensions.k8s.io/crunchybridgeclusters.postgres-
operator.crunchydata.com serverside-applied
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/perconapgupgrades.pgv2.percona.com
serverside-applied
customresourcedefinition.apiextensions.k8s.io/pgadmins.postgres-operator.crunchydata.com
serverside-applied
customresourcedefinition.apiextensions.k8s.io/pgupgrades.postgres-operator.crunchydata.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
```

At this point, the Operator Pod is up and running.

3. Deploy Percona Distribution for PostgreSQL cluster:

```
$ kubectl apply -f https://raw.githubusercontent.com/percona/percona-postgresql-
operator/v2.5.1/deploy/cr.yaml -n postgres-operator
```

Expected output

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

4 Check the Operator and replica set Pods status.

```
$ kubectl get pg -n postgres-operator
```

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

Expected output

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

You have successfully installed and deployed the Operator with default parameters. You can check them in the Custom Resource options reference.

Next steps

Connect to PostgreSQL →

1 Quick install

Install Percona Distribution for PostgreSQL using Helm

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

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

Prerequisites

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

1. Helm v3.
2. kubectl command line utility.
3. A Kubernetes environment. You can deploy it locally on Minikube for testing purposes or using any cloud provider of your choice. Check the list of our officially supported platforms.

See also

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

Installation

Here's a sequence of steps to follow:

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

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

2. It is a good practice to isolate workloads in Kubernetes via namespaces. Create a namespace:

```
$ kubectl create namespace <my-namespace>
```

3. Install the Percona Operator for PostgreSQL:

```
$ helm install my-operator percona/pg-operator --namespace <my-namespace>
```

The `my-namespace` is the name of your namespace. The `my-operator` parameter 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).

4. Install Percona Distribution for PostgreSQL:

```
$ helm install cluster1 percona/pg-db -n <my-namespace>
```

The `cluster1` parameter 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).

5. Check the Operator and replica set Pods status.

```
$ kubectl get pg -n <my-namespace>
```

The creation process is over when both the Operator and replica set Pods report the `ready` status:

Expected output

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

You have successfully installed and deployed the Operator with default parameters. You can check them in the Custom Resource options reference.

Next steps

Connect to PostgreSQL →

2 Connect to the PostgreSQL cluster

When the installation is done, we can connect to the cluster.

The `pgBouncer` component of Percona Distribution for PostgreSQL provides the point of entry to the PostgreSQL cluster. We will use the `pgBouncer` URI to connect.

The `pgBouncer` URI is stored in the Secret object, which the Operator generates during the installation.

To connect to PostgreSQL, do the following:

1. List the Secrets objects

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

The Secrets object we target is named as `<cluster_name>-pguser-<cluster_name>`. The `<cluster_name>` value is the name of your Percona Distribution for PostgreSQL Cluster. The default variant is:

via kubectl

`cluster1-pguser-cluster1`

via Helm

`cluster1-pg-db-pguser-cluster1-pg-db`

2. Retrieve the pgBouncer URI from your secret, decode and pass it as the `PGBOUNCER_URI` environment variable. Replace the `<secret>`, `<namespace>` placeholders with your Secret object and namespace accordingly:

```
$ PGBOUNCER_URI=$(kubectl get secret <secret> --namespace <namespace> -o
jsonpath='{.data.pgbouncer-uri}' | base64 --decode)
```

The following example shows how to pass the pgBouncer URI from the default Secret object `cluster1-pguser-cluster1`:

```
$ PGBOUNCER_URI=$(kubectl get secret cluster1-pguser-cluster1 --namespace
<namespace> -o jsonpath='{.data.pgbouncer-uri}' | base64 --decode)
```

3. Create a Pod where you start a container with Percona Distribution for PostgreSQL and connect to the database. The following command does it, naming the Pod `pg-client` and connects you to the `cluster1` database:

```
$ kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-
postgresql:16 --restart=Never -- psql $PGBOUNCER_URI
```

It may take some time to create the Pod and connect to the database. As the result, you should see the following sample output:

Expected output

```
psql (16.8)
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression:
off)
Type "help" for help.
cluster1=>
```

Congratulations! You have connected to your PostgreSQL cluster.

Next steps

Insert testing data →

3 Insert sample data

The next step after connecting to the cluster is to insert some sample data to PostgreSQL.

Create a schema

Every database in PostgreSQL has a default schema called `public`. A schema stores database objects like tables, views, indexes and allows organizing them into logical groups.

When you create a table, it ends up in the `public` schema by default. In recent PostgreSQL versions (starting from PostgreSQL 15), non-database owners cannot access the `public` schema. Therefore, you need to create a new schema to insert the data.

Use the following statement to create a schema

```
CREATE SCHEMA demo;
```

Create a table

After you created a schema, all tables you create end up in this schema if not specified otherwise.

At this step, we will create a sample table `Library` as follows:

```
CREATE TABLE LIBRARY(
   ID INTEGER NOT NULL,
   NAME TEXT,
   SHORT_DESCRIPTION TEXT,
   AUTHOR TEXT,
   DESCRIPTION TEXT,
   CONTENT TEXT,
   LAST_UPDATED DATE,
   CREATED DATE
);
```

Tip

If the schema has not been automatically set to the one you created, set it manually using the following SQL statement:

```
SET schema 'demo';
```

Replace the `demo` schema name with your value if you used another name.

Insert the data

PostgreSQL does not have the built-in support to generate random data. However, it provides the `random()` function which generates random numbers and `generate_series()` function which generates the series of rows and populates them with the numbers incremented by 1 (by default).

Combine these functions with a couple of others to populate the table with the data:

```
INSERT INTO LIBRARY(id, name, short_description, author,
                              description,content, last_updated, created)
SELECT id, 'name', md5(random()::text), 'name2'
      ,md5(random()::text),md5(random()::text)
      ,NOW() - '1 day'::INTERVAL * (RANDOM()::int * 100)
      ,NOW() - '1 day'::INTERVAL * (RANDOM()::int * 100 + 100)
FROM generate_series(1,100) id;
```

This command does the following:

- Fills in the columns `id`, `name`, `author` with the values `id`, `name` and `name2` respectively;
- generates the random md5 hash sum as the values for the columns `short_description`, `description` and `content`;
- generates the random number of dates from the current date and time within the last 100 days, and
- inserts 100 rows of this data

Now your cluster has some data in it.

Next steps

Make a backup →

4 Make a backup

Now your database contains some data, so it's a good time to learn how to manually make a full backup of your data with the Operator.

Note

If you are interested to learn more about backups, their types and retention policy, see the Backups section.

Considerations and prerequisites

- In this tutorial we use the AWS S3 as the backup storage. You need the following S3-related information:
  - The name of S3 bucket;
  - The endpoint - the URL to access the bucket
  - The region - the location of the bucket
  - S3 credentials such as S3 key and secret to access the storage.

  If you don't have access to AWS, you can use any S3-compatible storage like MinIO. Check the list of supported storages. Find the storage configuration instructions for each
- The Operator uses the `pgBackRest` tool to make backups. `pgBackRest` stores the backups and archives WAL segments in repositories. The Operator has up to four `pgBackRest` repositories named `repo1`, `repo2`, `repo3` and `repo4`. In this tutorial we use `repo2` for backups.
- Also, we will use some files from the Operator repository for setting up backups. So, clone the percona-postgresql-operator repository:

  ```
  $ git clone -b v2.5.1 https://github.com/percona/percona-postgresql-operator
  $ cd percona-postgresql-operator
  ```

  Note

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

Configure backup storage

1. Encode the S3 credentials and the pgBackRest repository name (`repo2` in our setup).

Linux

```
$ cat <<EOF | base64 --wrap=0
[global]
repo2-s3-key=<YOUR_AWS_S3_KEY>
repo2-s3-key-secret=<YOUR_AWS_S3_KEY_SECRET>
EOF
```

macOS

```
$ cat <<EOF | base64
[global]
repo2-s3-key=<YOUR_AWS_S3_KEY>
repo2-s3-key-secret=<YOUR_AWS_S3_KEY_SECRET>
EOF
```

2. Create the Secret configuration file and specify the base64-encoded string from the previous step. The following is the example of the `cluster1-pgbackrest-secrets.yaml` Secret file:

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

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

```
$ kubectl apply -f cluster1-pgbackrest-secrets.yaml -n <namespace>
```

4. Update your `deploy/cr.yaml` configuration. Specify the Secret file you created in the `backups.pgbackrest.configuration` subsection, and put all other S3 related information in the `backups.pgbackrest.repos` subsection under the repository name that you intend to use for backups. This name must match the name you used when you encoded S3 credentials on step 1.

   For example, the S3 storage for the `repo2` repository looks as follows:

```
...
backups:
  pgbackrest:
    ...
    configuration:
      - secret:
          name: cluster1-pgbackrest-secrets
    ...
    repos:
    - name: repo2
      s3:
        bucket: "<YOUR_AWS_S3_BUCKET_NAME>"
        endpoint: "<YOUR_AWS_S3_ENDPOINT>"
        region: "<YOUR_AWS_S3_REGION>"
```

5. Create or update the cluster. Specify your namespace instead of the `<namespace>` placeholder:

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

Make a backup

For manual backups, you need a backup configuration file.

1. Edit the example backup configuration file deploy/backup.yaml. Specify your cluster name and the `repo` name.

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGBackup
metadata:
  name: backup1
spec:
  pgCluster: cluster2
  repoName: repo1
#  options:
#  - --type=full
```

2. Apply the configuration. This instructs the Operator to start a backup.

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

3. List the backup

```
$ kubectl get pg-backup -n <namespace>
```

Congratulations! You have made the first backup manually. Want to learn more about backups? See the Backup and restore section for details like types, retention and how to automatically make backups according to the schedule.

Next steps

Monitor the database →

5 Monitor the database

Finally, when we are done with backup, it's time for one more step. In this section you will learn how to monitor the health of Percona Distribution for PostgreSQL with Percona Monitoring and Management (PMM).

Note

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

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

A PMM Client collects needed metrics and sends gathered data to the PMM Server. As a user, you connect to the PMM Server to see database metrics on a number of dashboards. PMM Server and PMM Client are installed separately.

Install PMM Server

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

Install PMM Client

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

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

From PMM UI

Generate the PMM API key

From command line

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

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

Note

The API key is not rotated.

2. Specify the API key as the `PMM_SERVER_KEY` value in the deploy/secrets.yaml secrets file.

```
apiVersion: v1
kind: Secret
metadata:
  name: cluster1-pmm-secret
type: Opaque
stringData:
  PMM_SERVER_KEY: ""
```

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

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

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

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

```
  pmm:
    enabled: true
    image: percona/pmm-client:2.44.0
#    imagePullPolicy: IfNotPresent
    secret: cluster1-pmm-secret
    serverHost: monitoring-service
```

5. Update the cluster

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

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

```
$ kubectl get pods -n postgres-operator
$ kubectl logs <pod_name> -c pmm-client
```

Update the secrets file

The `deploy/secrets.yaml` file contains all values for each key/value pair in a convenient plain text format. But the resulting Secrets Objects contains passwords stored as base64-encoded strings. If you want to *update* the password field, you need to encode the new password into the base64 format and pass it to the Secrets Object.

To encode a password or any other parameter, run the following command:

Linux

```
$ echo -n "password" | base64 --wrap=0
```

macOS

```
$ echo -n "password" | base64
```

For example, to set the new PMM API key in the `my-cluster-name-secrets` object, do the following:

Linux

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

macOS

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

Check the metrics

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

1. Log in to PMM server.
2. Click **PostgreSQL** from the left-hand navigation menu. You land on the **Instances Overview** page.
3. Click **PostgreSQL** → **Other dashboards** to see the list of available dashboards that allow you to drill down to the metrics you are interested in.

Next steps

What’s next →

What’s next?

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

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

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

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.

Supported versions

The Operator 2.5.1 is developed, tested and based on:

- PostgreSQL 16.8 as the database. Other versions may also work but have not been tested.
- pgBouncer 1.24.0 for connection pooling
- Patroni 3.3.2 for high-availability.

Supported platforms

The following platforms were tested and are officially supported by the Operator 2.5.1:

- Google Kubernetes Engine (GKE) 1.28 - 1.30
- Amazon Elastic Container Service for Kubernetes (EKS) 1.28 - 1.30
- OpenShift 4.13.46 - 4.16.7
- Azure Kubernetes Service (AKS) 1.28 - 1.30
- Minikube 1.34.0 with Kubernetes 1.31.0

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

Installation guidelines

Choose how you wish to install Percona Operator for PostgreSQL:

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

Installation

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.

This document describes how to deploy the Operator and Percona Distribution for PostgreSQL on Minikube.

Set up Minikube

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

    1. kubectl tool,
    2. a hypervisor, if it is not already installed,
    3. actual minikube package

2. After the installation, initialize and start the Kubernetes cluster. The parameters we pass for the following command increase the virtual machine limits for the CPU cores, memory, and disk, to ensure stable work of the Operator:

    ```
    $ minikube start --memory=5120 --cpus=4 --disk-size=30g
    ```

    This command downloads needed virtualized images, then initializes and runs the cluster.

3. After Minikube is successfully started, you can optionally run the Kubernetes dashboard, which visually represents the state of your cluster. Executing `minikube dashboard` starts the dashboard and opens it in your default web browser.

Deploy the Percona Operator for PostgreSQL

1.

Create the Kubernetes namespace for your cluster. It is a good practice to isolate workloads in Kubernetes by installing the Operator in a custom namespace. For example, let's name it `postgres-operator`:

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

Expected output

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

We will use this namespace further on in this document. If you used another name, make sure to replace it in the following commands.

2. Deploy the Operator using the following command:

```
$ kubectl apply --server-side -f https://raw.githubusercontent.com/percona/
percona-postgresql-operator/v2.5.1/deploy/bundle.yaml -n postgres-operator
```

Expected output

```
customresourcedefinition.apiextensions.k8s.io/crunchybridgeclusters.postgres-
operator.crunchydata.com serverside-applied
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/perconapgupgrades.pgv2.percona.com
serverside-applied
customresourcedefinition.apiextensions.k8s.io/pgadmins.postgres-operator.crunchydata.com
serverside-applied
customresourcedefinition.apiextensions.k8s.io/pgupgrades.postgres-operator.crunchydata.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 have the Operator Pod up and running.

3. Deploy Percona Distribution for PostgreSQL:

```
$ kubectl apply -f https://raw.githubusercontent.com/percona/percona-postgresql-
operator/v2.5.1/deploy/cr.yaml -n postgres-operator
```

Expected output

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

Note

This deploys the 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 v2.5.1 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
```

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

```
$ kubectl get pg -n postgres-operator
```

Expected output

```
NAME       ENDPOINT                         STATUS   POSTGRES   PGBOUNCER   AGE
cluster1   cluster1-pgbouncer.default.svc   ready    3          3           30m
```

Verify the Percona Distribution for PostgreSQL cluster operation

When creation process is over, the output of the `kubectl get pg` command shows the cluster status as `ready`. You can try to connect to the cluster.

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

1.

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`.

2. Use the following command to get the password of this user. Replace the `<cluster_name>` and `<namespace>` placeholders with your values:

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

3. Create a pod and start Percona Distribution for PostgreSQL inside. The following command will do this, naming the new Pod `pg-client`:

```
$ kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-
postgresql:16.8 --restart=Never -- bash -il
```

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

4. Run a container with `psql` tool and connect its console output to your terminal. The following command will connect you as a `cluster1` user to a `cluster1` database via the PostgreSQL interactive terminal.

```
[postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-
pgbouncer.postgres-operator.svc -p 5432 -U cluster1 cluster1
```

Sample output

```
psql (16.8)
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression:
off)
Type "help" for help.
pgdb=>
```

Delete the cluster

If you need to delete the Operator and PostgreSQL cluster (for example, to clean up the testing deployment before adopting it for production use), check this HowTo.

If you no longer need the Kubernetes cluster in Minikube, the following are the steps to remove it.

1. Stop the Minikube cluster:

   ```
   $ minikube stop
   ```

2. Delete the cluster

   ```
   $ minikube delete
   ```

   This command deletes the virtual machines, and removes all associated files.

Install Percona Distribution for PostgreSQL cluster using Everest

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

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

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

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

Following steps help you 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 GKE, see the Kubernetes Engine Quickstart.

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. This is the Kubernetes command-line tool you will use to manage and deploy applications. To install the tool, run the following command:

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

Create and configure the GKE cluster

You can configure the settings using the `gcloud` tool. You can run it either in the Cloud Shell or in your local shell (if you have installed Google Cloud SDK locally on the previous step). The following command creates a cluster named `cluster-1`:

```
$ gcloud container clusters create cluster-1 --project <project ID> --zone us-central1-a --cluster-version 1.30 --machine-type n1-standard-4 --num-nodes=3
```

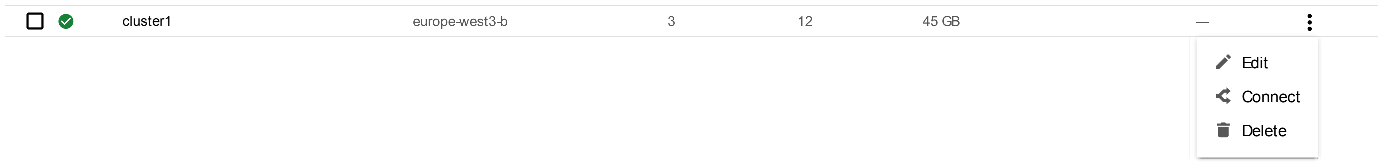
Note

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

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

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

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


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
```

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 v2.5.1 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 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/crunchybridgeclusters.postgres-
operator.crunchydata.com serverside-applied
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/perconapgupgrades.pgv2.percona.com
serverside-applied
customresourcedefinition.apiextensions.k8s.io/pgadmins.postgres-operator.crunchydata.com
serverside-applied
customresourcedefinition.apiextensions.k8s.io/pgupgrades.postgres-operator.crunchydata.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
```

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

```
$ kubectl get pg -n postgres-operator
```

Expected output

```
NAME        ENDPOINT                          STATUS    POSTGRES    PGBOUNCER    AGE
cluster1    cluster1-pgbouncer.default.svc    ready     3           3            30m
```

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

Verifying the cluster operation

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

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

1. 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`.

2. Use the following command to get the password of this user. Replace the `<cluster_name>` and `<namespace>` placeholders with your values:

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

3. Create a pod and start Percona Distribution for PostgreSQL inside. The following command will do this, naming the new Pod `pg-client`:

```
$ kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-
postgresql:16.8 --restart=Never -- bash -il
```

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

4. Run a container with `psql` tool and connect its console output to your terminal. The following command will connect you as a `cluster1` user to a `cluster1` database via the PostgreSQL interactive terminal.

```
[postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-
pgbouncer.postgres-operator.svc -p 5432 -U cluster1 cluster1
```

Sample output

```
psql (16.8)
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression:
off)
Type "help" for help.
pgdb=>
```

Removing the cluster

If you need to delete the Operator and PostgreSQL cluster (for example, to clean up the testing deployment before adopting it for production use), check this HowTo.

Also, there are several ways that you can delete your Kubernetes cluster in GKE.

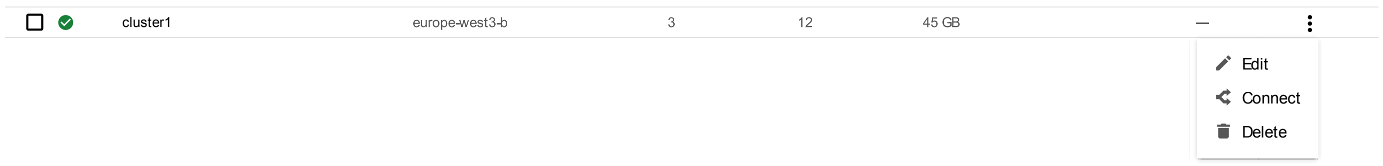
You can clean up the cluster with the `gcloud` command as follows:

```
$ gcloud container clusters delete <cluster name> --zone us-central1-a --project
<project ID>
```

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

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

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



☐ ✓	cluster1	europe-west3-b	3	12	45 GB	—

The cluster deletion may take time.

Warning

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

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.

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 servers 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.

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/v2.5.1/deploy/bundle.yaml -n postgres-operator
   ```

Expected output

```
customresourcedefinition.apiextensions.k8s.io/crunchybridgeclusters.postgres-
operator.crunchydata.com serverside-applied
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/perconapgupgrades.pgv2.percona.com
serverside-applied
customresourcedefinition.apiextensions.k8s.io/pgadmins.postgres-operator.crunchydata.com
serverside-applied
customresourcedefinition.apiextensions.k8s.io/pgupgrades.postgres-operator.crunchydata.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/v2.5.1/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 v2.5.1 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. When the process is over your cluster will obtain the `ready` status. You can check it with the following command:

```
$ kubectl get pg
```

Expected output

```
NAME       ENDPOINT                        STATUS   POSTGRES   PGBOUNCER   AGE
cluster1   cluster1-pgbouncer.default.svc  ready    3          3           30m
```

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.

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

1. 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`.

2. Use the following command to get the password of this user. Replace the `<cluster_name>` and `<namespace>` placeholders with your values:

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

3. Create a pod and start Percona Distribution for PostgreSQL inside. The following command will do this, naming the new Pod `pg-client`:

```
$ kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-
postgresql:16.8 --restart=Never -- bash -il
```

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

4. Run a container with `psql` tool and connect its console output to your terminal. The following command will connect you as a `cluster1` user to a `cluster1` database via the PostgreSQL interactive terminal.

```
[postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-
pgbouncer.postgres-operator.svc -p 5432 -U cluster1 cluster1
```

Sample output

```
psql (16.8)
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression:
off)
Type "help" for help.
pgdb=>
```

Removing the cluster

If you need to delete the Operator and PostgreSQL cluster (for example, to clean up the testing deployment before adopting it for production use), check this HowTo.

To delete your Kubernetes cluster in EKS, 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!

Install Install Percona Distribution for PostgreSQL on Azure Kubernetes Service (AKS)

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

Prerequisites

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

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

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

Create and configure the AKS cluster

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

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

You can create your cluster via command line using `az aks create` command. The following command will create a 3-node cluster named `cluster1` within some already existing resource group named `my-resource-group`:

```
$ az aks create --resource-group my-resource-group --name  cluster1 --enable-
managed-identity --node-count 3 --node-vm-size Standard_B4ms --node-osdisk-size 30
--network-plugin kubenet  --generate-ssh-keys --outbound-type loadbalancer
```

Other parameters in the above example specify that we are creating a cluster with machine type of Standard_B4ms and OS disk size reduced to 30 GiB. You can see detailed information about cluster creation options in the AKS official documentation.

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

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

```
az aks get-credentials --resource-group my-resource-group --name  cluster1
```

Install the Operator and deploy your PostgreSQL cluster

1. Create the Kubernetes namespace for your cluster. It is a good practice to isolate workloads in Kubernetes by installing the Operator in a custom namespace. For example, let's name it `postgres-operator`:

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

   Expected output

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

   We will use this namespace further on in this document. If you used another name, make sure to replace it in the following commands.

2. Deploy the Operator[using](#) the following command:

   ```
   $ kubectl apply --server-side -f https://raw.githubusercontent.com/percona/
   percona-postgresql-operator/v2.5.1/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
   ```

   At this point, the Operator Pod is up and running.

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

```
$ kubectl apply -f https://raw.githubusercontent.com/percona/percona-postgresql-
operator/v2.5.1/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 v2.5.1 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. When the process is over your cluster will obtain the `ready` status. You can check it with the following command:

```
$ kubectl get pg
```

Expected output

```
NAME       ENDPOINT                         STATUS   POSTGRES   PGBOUNCER   AGE
cluster1   cluster1-pgbouncer.default.svc   ready    3          3           30m
```

Verifying the cluster operation

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

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

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`.

2. Use the following command to get the password of this user. Replace the `<cluster_name>` and `<namespace>` placeholders with your values:

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

3. Create a pod and start Percona Distribution for PostgreSQL inside. The following command will do this, naming the new Pod `pg-client`:

```
$ kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-
postgresql:16.8 --restart=Never -- bash -il
```

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

4. Run a container with `psql` tool and connect its console output to your terminal. The following command will connect you as a `cluster1` user to a `cluster1` database via the PostgreSQL interactive terminal.

```
[postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-
pgbouncer.postgres-operator.svc -p 5432 -U cluster1 cluster1
```

Sample output

```
psql (16.8)
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression:
off)
Type "help" for help.
pgdb=>
```

Removing the AKS cluster

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

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

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

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

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

Warning

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

Install Percona Distribution for PostgreSQL on OpenShift

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

Installing Percona Distribution for PostgreSQL on OpenShift includes two steps:

- Installing the Percona Operator for PostgreSQL,
- Install Percona Distribution for PostgreSQL using the Operator.

Install the Operator

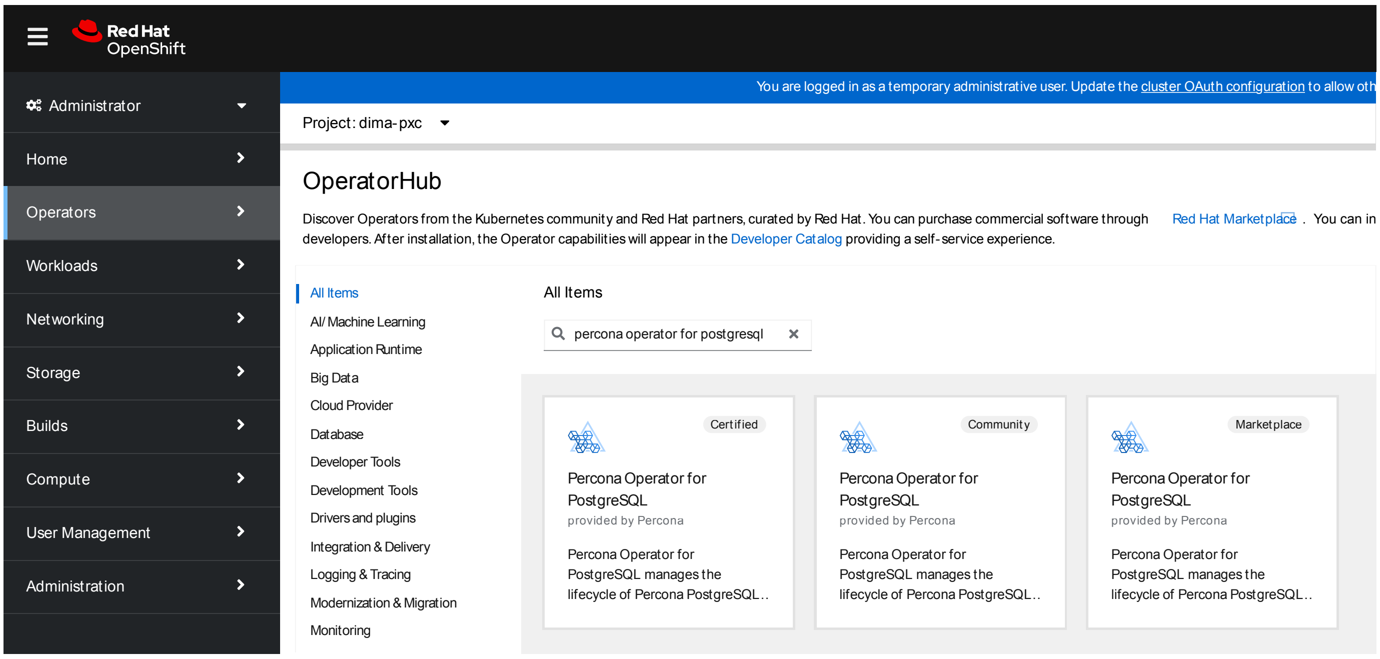
You can install Percona Operator for MySQL on OpenShift using the web interface (the [Operator Lifecycle Manager]() or [Red Hat Marketplace](), or using the command line interface.

Install the Operator via the Operator Lifecycle Manager (OLM)

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

Following steps will allow you to deploy the Operator and PostgreSQL cluster on your OLM installation:

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



You are logged in as a temporary administrative user. Update the cluster OAuth configuration to allow oth

Project: dima-pxc

OperatorHub

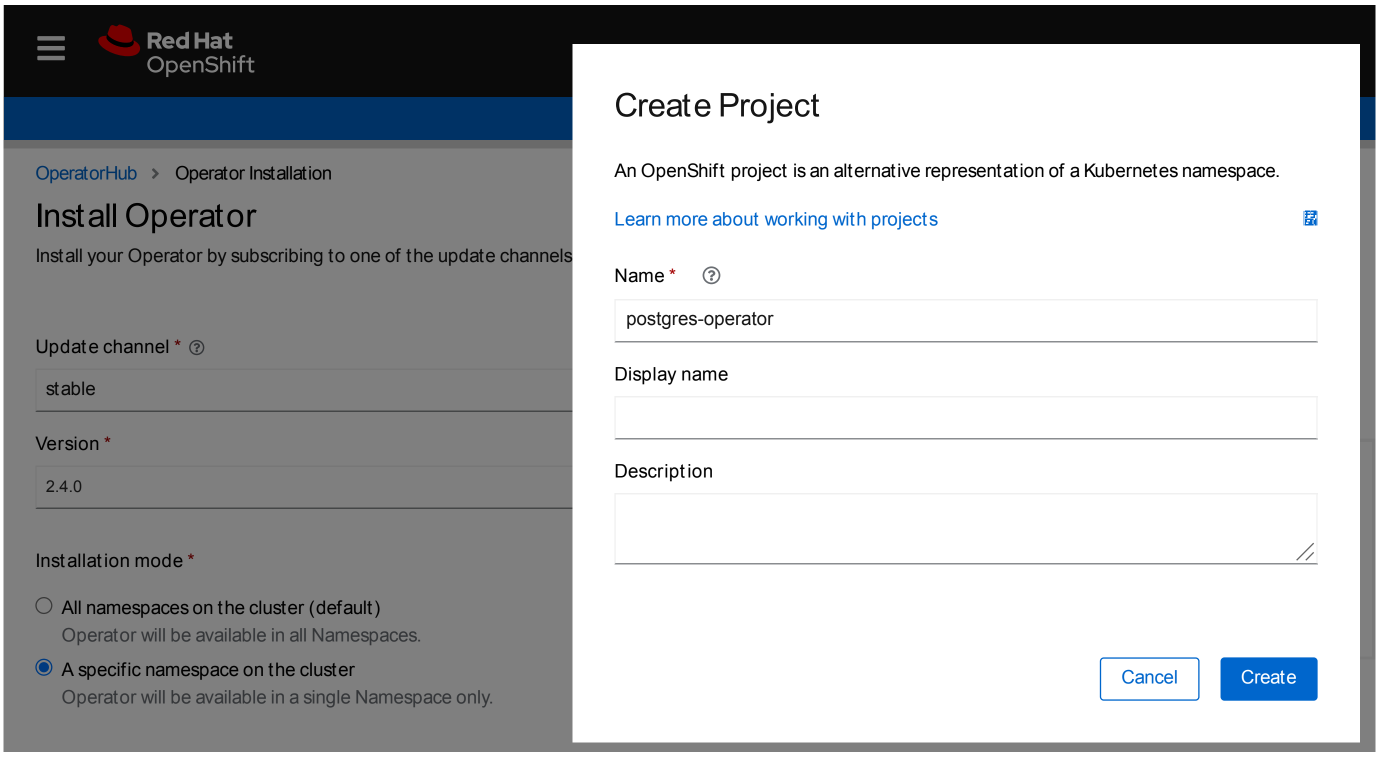
Discover Operators from the Kubernetes community and Red Hat partners, curated by Red Hat. You can purchase commercial software through Red Hat Marketplace. You can in developers. After installation, the Operator capabilities will appear in the Developer Catalog providing a self-service experience.

percona operator for postgresql

Then click “Contiune”, and “Install”.

2.

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


OperatorHub > Operator Installation

Install Operator

Install your Operator by subscribing to one of the update channels

Update channel * : stable

Version * : 2.4.0

Installation mode *

( ) All namespaces on the cluster (default)
Operator will be available in all Namespaces.

(●) A specific namespace on the cluster
Operator will be available in a single Namespace only.

Create Project

An OpenShift project is an alternative representation of a Kubernetes namespace.

Learn more about working with projects

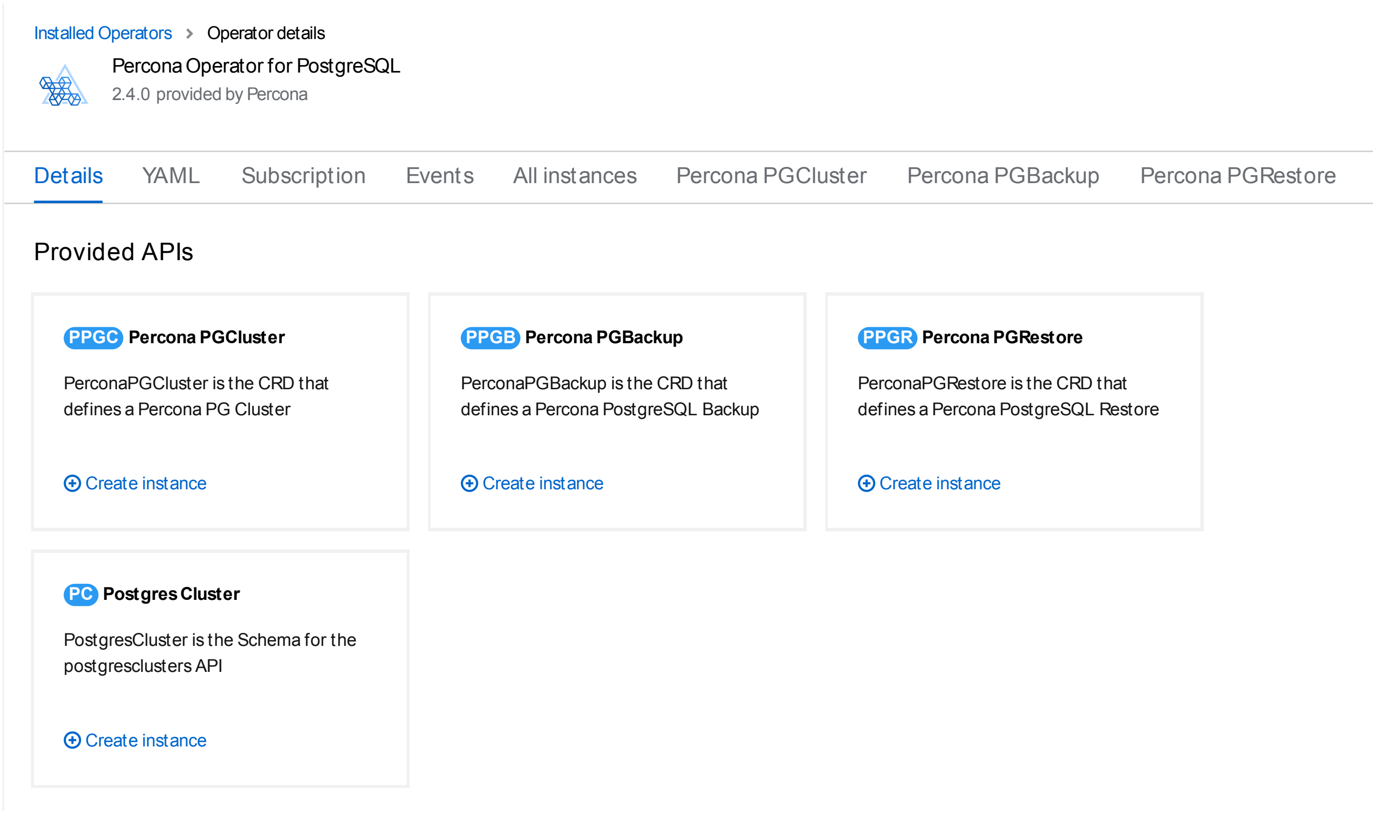
Name * : postgres-operator

Display name :

Description :

Click “Install” button to actually install the Operator.

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


Installed Operators > Operator details

Percona Operator for PostgreSQL
2.4.0 provided by Percona

Provided APIs

PPGC Percona PGCluster
PerconaPGCluster is the CRD that defines a Percona PG Cluster

PPGB Percona PGBackup
PerconaPGBackup is the CRD that defines a Percona PostgreSQL Backup

PPGR Percona PGRestore
PerconaPGRestore is the CRD that defines a Percona PostgreSQL Restore

PC Postgres Cluster
PostgresCluster is the Schema for the postgresclusters API

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

Install the Operator via the command-line interface

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

   ```
   $ git clone -b v2.5.1 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. The Custom Resource Definition for Percona Distribution for PostgreSQL should be created from the `deploy/crd.yaml` file. Custom Resource Definition extends the standard set of resources which OpenShift “knows” about with the new items (in our case ones which are the core of the Operator). Apply it as follows:

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

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

3. Create the OpenShift namespace for your cluster if needed (for example, let’s name it `postgres-operator`):

   ```
   $ oc create namespace postgres-operator
   ```

   > **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.

4. The role-based access control (RBAC) for Percona Distribution for PostgreSQL is configured with the `deploy/rbac.yaml` file. Role-based access is based on defined roles and the available actions which correspond to each role. The role and actions are defined for Kubernetes resources in the yaml file. Further details about users and roles can be found in specific OpenShift documentation)

   ```
   $ oc apply -f deploy/rbac.yaml -n postgres-operator
   ```

Note

Setting RBAC requires your user to have cluster-admin role privileges. For example, those using Google OpenShift Engine can grant user needed privileges with the following command:

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

5. If you are going to use the operator with anyuid security context constraint please execute the following command:

```
$ sed -i '/disable_auto_failover: "false"/a \ \ \ \ disable_fsgroup: "false"'
deploy/operator.yaml
```

6. Start the Operator within OpenShift:

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

Optionally, you can add PostgreSQL Users secrets and TLS certificates to OpenShift. If you don't, the Operator will create the needed users and certificates automatically, when you create the database cluster. You can see documentation on Users and TLS certificates if still want to create them yourself.

Note

You can simplify the Operator installation by applying a single `deploy/bundle.yaml` file instead of running commands from the steps 2 and 4:

```
$ oc apply -f deploy/bundle.yaml
```

This will automatically create Custom Resource Definition, set up role-based access control and install the Operator as one single action.

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

```
$ oc 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:

```
$ oc get pg -n postgres-operator
```

Expected output

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

Verifying the cluster operation

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

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

1. Use `oc 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`.

2. Use the following command to get the password of this user. Replace the `<cluster_name>` and `<namespace>` placeholders with your values:

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

3. Create a pod and start Percona Distribution for PostgreSQL inside. The following command will do this, naming the new Pod `pg-client`:

   ```
   $ oc run -i --rm --tty pg-client --image=perconalab/percona-distribution-
   postgresql:16.8 --restart=Never -- bash -il
   ```

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

4. Run a container with `psql` tool and connect its console output to your terminal. The following command will connect you as a `cluster1` user to a `cluster1` database via the PostgreSQL interactive terminal.

   ```
   [postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-
   pgbouncer.postgres-operator.svc -p 5432 -U cluster1 cluster1
   ```

Sample output

```
psql (16.8)
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression:
off)
Type "help" for help.
pgdb=>
```

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 v2.5.1 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. The Custom Resource Definition for Percona Distribution for PostgreSQL should be created from the `deploy/crd.yaml` file. Custom Resource Definition extends the standard set of resources which Kubernetes "knows" about with the new items (in our case ones which are the core of the Operator). Apply it as follows:

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

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

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

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

   > **Note**
   >
   > To use a different namespace, specify another 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.

4. The role-based access control (RBAC) for Percona Distribution for PostgreSQL is configured with the `deploy/rbac.yaml` file. Role-based access is based on defined roles and the available actions which correspond to each role. The role and actions are defined for Kubernetes resources in the yaml file. Further details about users and roles can be found in Kubernetes documentation.

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

Note

Setting RBAC requires your user to have cluster-admin role privileges. For example, those using Google Kubernetes Engine can grant user needed privileges with the following command:

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

5. Start the Operator within Kubernetes:

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

Optionally, you can add PostgreSQL Users secrets and TLS certificates to Kubernetes. If you don't, the Operator will create the needed users and certificates automatically, when you create the database cluster. You can see documentation on Users and TLS certificates if still want to create them yourself.

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

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

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

```
$ kubectl get pg -n postgres-operator
```

Expected output

```
NAME       ENDPOINT                        STATUS    POSTGRES    PGBOUNCER    AGE
cluster1   cluster1-pgbouncer.default.svc  ready     3           3            30m
```

Verifying the cluster operation

When creation process is over, the output of the `kubectl get pg` command shows the cluster status as `ready`. You can now try to connect to the cluster.

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

1. 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`.

2. Use the following command to get the password of this user. Replace the `<cluster_name>` and `<namespace>` placeholders with your values:

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

3. Create a pod and start Percona Distribution for PostgreSQL inside. The following command will do this, naming the new Pod `pg-client`:

   ```
   $ kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-
   postgresql:16.8 --restart=Never -- bash -il
   ```

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

4. Run a container with `psql` tool and connect its console output to your terminal. The following command will connect you as a `cluster1` user to a `cluster1` database via the PostgreSQL interactive terminal.

   ```
   [postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-
   pgbouncer.postgres-operator.svc -p 5432 -U cluster1 cluster1
   ```

   Sample output

   ```
   psql (16.8)
   SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression:
   off)
   Type "help" for help.
   pgdb=>
   ```

Deleting the cluster

If you need to delete the cluster (for example, to clean up the testing deployment before adopting it for production use), check this HowTo.

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.

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.

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):

```
...
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:

```
...
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`:

```
...
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:

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

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

```
...
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:

```
...
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.

Configuration

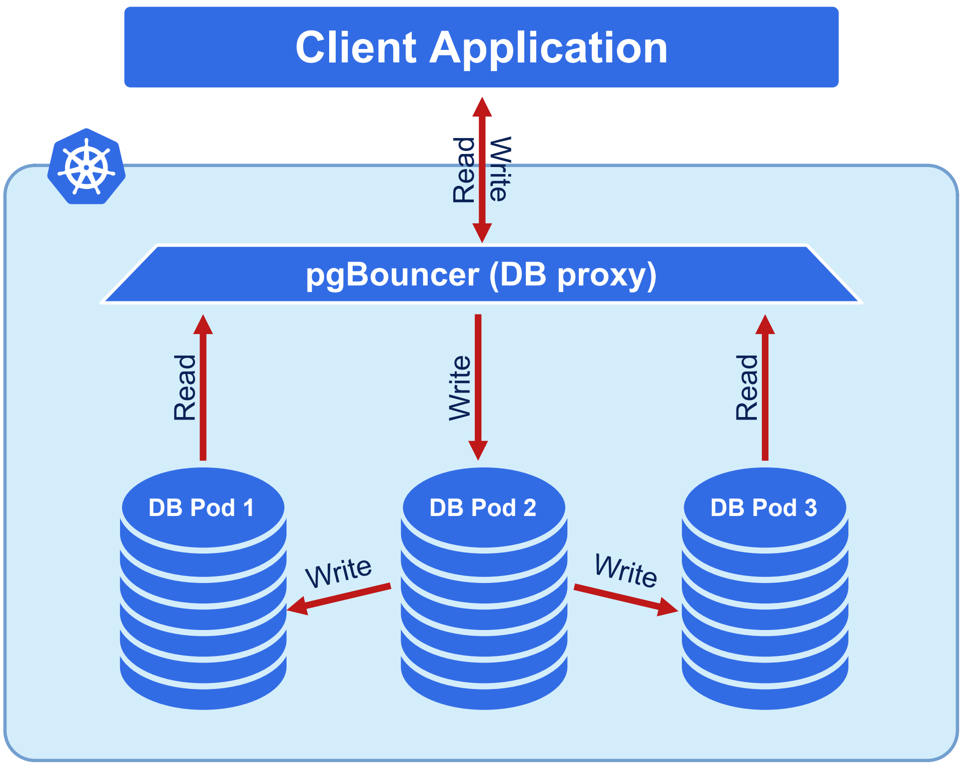
Exposing cluster

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.

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:2.5.1-ppg14-pgbouncer
    expose:
      type: LoadBalancer
```

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

```
$ kubectl get service
```

Expected output

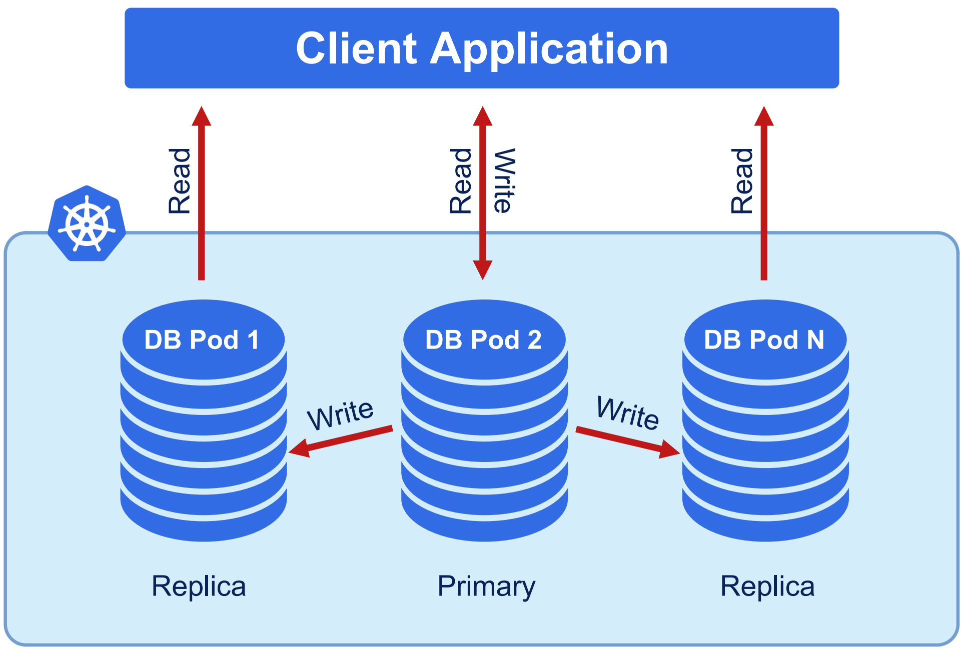
```
NAME                 TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)          AGE
...
cluster1-pgbouncer   LoadBalancer   10.88.8.48     34.133.38.186   5432:30601/TCP   20m
...
```

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`.

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
...
cluster1-replicas    ClusterIP   10.88.8.115   <none>        5432/TCP   2m16s
```

The `cluster1-ha` service points to the active primary. In case of failover to the replica node, will change the endpoint automatically. Also, you can use `cluster1-replicas` service to make read requests to PostgreSQL replica instances.

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:

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

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:

```
...
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.

Using host-based authentication (pg_hba)

PostgreSQL Host-Based Authentication (pg_hba) allows controlling access to the PostgreSQL database based on the IP address or the host name of the connecting host. You can configure `pg_hba` through the Custom Resource `patroni.dynamicConfiguration.postgresql.pg_hba` subsection as follows:

```
...
patroni:
  dynamicConfiguration:
    postgresql:
      pg_hba:
      - host    all all 0.0.0.0/0 md5
```

As you may guess, this example allows all hosts to connect to any database with MD5 password-based authentication.

Obviously, you can connect both `dynamicConfiguration.postgresql.parameters` and `dynamicConfiguration.postgresql.pg_hba` subsections:

```
...
patroni:
  dynamicConfiguration:
    postgresql:
      parameters:
        max_parallel_workers: 2
        max_worker_processes: 2
        shared_buffers: 1GB
        work_mem: 2MB
      pg_hba:
      - local   all all trust
      - host    all all 0.0.0.0/0 md5
      - host    all all ::1/128   md5
      - host    all mytest 123.123.123.123/32 reject
```

The changes will be applied after you update Custom Resource in a usual way:

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

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.

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.

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]().

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:

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

The [Kubernetes Taints and Toleratins]() contains more examples on this topic.

Labels and annotations

Labels and annotations are used to attach additional metadata information to Kubernetes resources.

Labels and annotations are rather similar. The difference between them is that labels are used by Kubernetes to identify and select objects, while annotations are assigning additional *non-identifying* information to resources. Therefore, typical role of Annotations is facilitating integration with some external tools.

Setting labels and annotations in the Custom Resource

You can set labels and/or annotations as key/value string pairs in the Custom Resource metadata section of the `deploy/cr.yaml`. For PostgreSQL, pgBouncer and pgBackRest Pods, use `instances.metadata.annotations` / `instances.metadata.labels`, `proxy.pgbouncer.metadata.annotations` / `proxy.pgbouncer.metadata.labels`, or `backups.pgbackrest.metadata.annotations` / `backups.pgbackrest.metadata.labels` keys as follows:

```yaml
apiVersion: pgv2.percona.com/v2
kind: PerconaPGCluster
...
spec:
...
  instances:
   - name: instance1
     replicas: 3
     metadata:
       annotations:
         my-annotation: value1
       labels:
         my-label: value2
    ...
```

For PostgreSQL and pgBouncer Services, use `expose.annotations` / `expose.labels` or `proxy.pgbouncer.expose.annotations` / `proxy.pgbouncer.expose.labels` keys as follows:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGCluster
...
spec:
  ...
  expose:
    annotations:
      my-annotation: value1
    labels:
      my-label: value2
    ...
```

The easiest way to check which labels are attached to a specific object with is using the additional `--show-labels` option of the `kubectl get` command. Checking the annotations is not much more difficult: it can be done as in the following example:

```
$ kubectl get service cluster1-pgbouncer -o jsonpath='{.metadata.annotations}'
```

Settings labels and annotations to the Operator Pod

You can assign labels and/or annotations to the Pod of the Operator itself by editing the deploy/operator.yaml configuration file before applying it during the installation.

```
apiVersion: apps/v1
kind: Deployment
...
spec:
...
  template:
    metadata:
      labels:
        app.kubernetes.io/component: operator
        app.kubernetes.io/instance: percona-postgresql-operator
        app.kubernetes.io/name: percona-postgresql-operator
        app.kubernetes.io/part-of: percona-postgresql-operator
        pgv2.percona.com/control-plane: postgres-operator
        ...
```

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 following ways:

- the Operator can generate long-term certificates automatically at cluster creation time,
- you can 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.

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.

Note

With the Operator versions before 2.5.0, autogenerated certificates for all database clusters were based on the same generated root CA. Starting from 2.5.0, the Operator creates root CA on per-cluster basis.

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:16.8
          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 (16.8)
Type "help" for help.
pgdb=>
```

Generate certificates manually

Provide pre-existing certificates to the Operator

To allow the Operator to use custom certificates, simply create the appropriate Secrets in your cluster namespace *before* deploying the cluster with the `kubectl apply -f deploy/cr.yaml` command. The Secret should contain the TLS key (`tls.key`), TLS certificate (`tls.crt`) and the CA certificate (`ca.crt`) to use:

```
apiVersion: v1
kind: Secret
metadata:
  name: cluster1-cert
type: Opaque
data:
  ca.crt: <value>
  tls.crt: <value>
  tls.key: <value>
```

For example, if you have files named `ca.crt`, `my_tls.key`, and `my_tls.crt` stored on your local machine, you could run the following command to create a Secret named `cluster1.tls` in the `postgres-operator` namespace:

```
$ kubectl create secret generic -n postgres-operator cluster1.tls \
  --from-file=ca.crt=ca.crt \
  --from-file=tls.key=my_tls.key \
  --from-file=tls.crt=my_tls.crt
```

You should use two sets of certificates: 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 in your Custom Resource. You can do it in the `deplou/cr.yaml` configuration file as follows:

```
spec:
  ...
  secrets:
     customTLSSecret:
       name: cluster1-cert
     customReplicationTLSSecret:
       name: replication1-cert
  ...
```

Don’t forget to apply changes as usual:

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

Provide custom root CA certificate to the Operator

You can also provide a custom root CA certificate to the Operator. In this case the Operator will not generate one itself, but will use the user-provided CA. Particularly, this can be useful if you would like to have several database clusters with certificates generated by the Operator based on the same root CA.

To make the Operator using custom root certificate, create a separate secret with this certificate and specify this secret in Custom Resource options.

For example, if you have files named `my_tls.key` and `my_tls.crt` stored on your local machine, you could run the following command to create a Secret named `cluster1-ca-cert` in the `postgres-operator` namespace:

```
$ kubectl create secret generic -n postgres-operator cluster1-ca-cert \
  --from-file=tls.crt=my_tls.crt \
  --from-file=tls.key=my_tls.key
```

You also need to specify details about this secret in your `deploy/cr.yaml` manifest:

```
...
secrets:
  customRootCATLSSecret:
    name: cluster1-ca-cert
    items:
      - key: "tls.crt"
        path: "root.crt"
      - key: "tls.key"
        path: "root.key"
```

Now, when you create the cluster with the `kubectl apply -f deploy/cr.yaml` command, the Operator should use the root CA certificate you had provided.

Warning

This approach allows using root CA certificate auto-generated by the Operator for some other clusters, but it needs caution. If the cluster with auto-generated certificate has `delete-ssl` finalizer enabled, the certificate will be deleted at the cluster deletion event even if it was manually provided to some other cluster.

Generate custom certificates for the Operator yourself

The good option to find out the certificates specifics needed for the Operator would be to look at certificates, generated by the Operator automatically. Supposing that your cluster name is `cluster1`, you can examine the auto-generated CA certificate (`ca.crt`) after deploying the cluster as follows:

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

Expected output

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            ec:f3:d6:f5:35:5c:97:0c:66:cc:90:ed:e6:4b:0a:07
        Signature Algorithm: ecdsa-with-SHA384
        Issuer: CN = postgres-operator-ca
        Validity
            Not Before: Dec 24 13:58:21 2023 GMT
            Not After : Dec 21 14:58:21 2033 GMT
        Subject: CN = postgres-operator-ca
        Subject Public Key Info:
        ...
    ...
```

You can check the auto-generated TLS certificate (`tls.crt`) in a similar way:

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

Expected output

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            59:f3:44:09:f1:73:b3:8e:ba:d4:a0:52:cc:fb:9c:1f
        Signature Algorithm: ecdsa-with-SHA384
        Issuer: CN = postgres-operator-ca
        Validity
            Not Before: Dec 24 13:58:21 2023 GMT
            Not After : Dec 23 14:58:21 2024 GMT
        Subject: CN = cluster1-primary.default.svc.cluster.local.
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:b1:2f:37:1b:ca:ab:5f:19:38:24:69:11:54:82:
                    10:49:fd:00:3c:26:ef:83:32:82:b1:73:96:e8:9d:
                    eb:2f:60:89:ea:3a:cb:95:a7:0a:2e:46:63:ce:29:
                    87:17:1a:d4:3e:c5:5a:90:8c:71:3b:23:75:21:42:
                    09:60:81:da:c1
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Authority Key Identifier:
                3C:25:65:88:F2:CD:29:37:05:06:7C:E8:F3:C4:2B:CD:9B:DC:5E:74
            X509v3 Subject Alternative Name:
                DNS:cluster1-primary.default.svc.cluster.local., DNS:cluster1-
primary.default.svc, DNS:cluster1-primary.default, DNS:cluster1-primary, DNS:cluster1-
replicas.default.svc.cluster.local., DNS:cluster1-replicas.default.svc, DNS:cluster1-
replicas.default, DNS:cluster1-replicas
    Signature Algorithm: ecdsa-with-SHA384
    ...
```

While sharing the same `ca.crt`, certificates for external communications (referenced in the `secrets.customTLSSecret.name` Custom Resource option) and certificates for internal ones (referenced in the `secrets.customReplicationTLSSecret.name` Custom Resource option) can't share the same `tls.crt`. The `tls.crt` for external communications should have a Common Name (CN) setting that matches the primary Service name (`CN = cluster1-primary.default.svc.cluster.local.` in the above example). Similarly, the `tls.crt` for internal communications should have a Common Name (CN) setting that matches the preset replication user: `CN=_crunchyrepl`.

One of the options to create certificates yourself is to use [CloudFlare PKI and TLS toolkit](). Supposing that your cluster name is `cluster1` and the desired namespace is `postgres-operator`, certificates generation may look as follows:

~~~
``` {.bash data-prompt="$" }
$ export CLUSTER_NAME=cluster1
$ export NAMESPACE=postgres-operator
$ cat <<EOF | cfssl gencert -initca - | cfssljson -bare ca
{
  "CN": "*",
  "key": {
    "algo": "ecdsa",
    "size": 384
  }
}
EOF

$ cat <<EOF > ca-config.json
{
   "signing": {
     "default": {
        "expiry": "87600h",
        "usages": ["digital signature", "key encipherment", "content commitment"]
      }
   }
}
EOF

$ cat <<EOF | cfssl gencert -ca=ca.pem  -ca-key=ca-key.pem -config=./ca-config.json
- | cfssljson -bare server
{
   "hosts": [
     "localhost",
     "${CLUSTER_NAME}-primary",
     "${CLUSTER_NAME}-primary.${NAMESPACE}",
     "${CLUSTER_NAME}-primary.${NAMESPACE}.svc.cluster.local",
     "${CLUSTER_NAME}-primary.${NAMESPACE}.svc"
   ],
   "CN": "${CLUSTER_NAME}-primary.${NAMESPACE}.svc.cluster.local",
   "key": {
     "algo": "ecdsa",
     "size": 384
   }
}
EOF
```
~~~

You can find more on genrating certificates this way in official Kubernetes documentation.

Don’t forget that you should generate certificates twice: one set is for external communications, and another set is for internal ones!

Check your certificates for expiration

```
$ kubectl get secrets
```

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

   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
   ```

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.

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 2.5.1 --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` environment variable and set it to `"true"`

```
...
- name: DISABLE_TELEMETRY
  value: "true"
...
```

Upgrade Database and Operator

Starting from the version 2.2.0 Percona Operator for PostgreSQL allows upgrades to newer 2.x versions.

> **Note**
>
> Upgrading from the 1.x branch of the Operator to 2.x versions ca be done in several ways and is completely different from the normal upgrade scenario due to substantial changes in the architecture.

Upgrading to a newer version typically involves two steps:

1. Upgrading the Operator and Custom Resource Definition (CRD) ,
2. Upgrading the Database Management System (Percona Distribution for PostgreSQL).

Alternatively, it is also possible to carry on minor version upgrades of Percona Distribution for PostgreSQL *without* the Operator upgrade.

Upgrading the Operator and CRD

The Operator supports only the incremental update to its nearest version (such as updating the Operator from 2.4.0 to 2.4.1). To update to a newer version, which differs from the current version by more than one, make several incremental updates sequentially. For example, to upgrade the Operator and CRD from the version 2.3.0 to 2.4.1, you need to do the following sequence of upgrades:

1. upgrading the Operator and CRD from 2.3.0 to 2.3.1,
2. upgrading from 2.3.1 to 2.4.0,
3. upgrading from 2.4.0 to 2.4.1.

You can find Operator versions listed here.

> **Note**
>
> The Operator supports **last 3 versions of the CRD**, so it is technically possible to skip upgrading the CRD and just upgrade the Operator. If the CRD is older than the new Operator version *by no more than three releases*, you should be able to continue using the old CRD and even carry on Percona Distribution for PostgreSQL minor version upgrades with it. But updating the Operator *and* CRD is the **recommended path**.

Manual upgrade

You can upgrade the Operator and CRD as follows, considering the Operator uses `postgres-operator` namespace, and you are upgrading to the version 2.5.1.

1. First update the CRD for the Operator, taking it from the official repository on Github (it is important to use `--server-side` flag when applying `deploy/crd.yaml`), and do the same for the Role-based access control. Applying the new CRD manifest must be done with server-side flag (otherwise you can encounter a number of errors caused by applying the CRD client-side: the command may fail, the built-in PosgreSQL extensions can be lost during such upgrade, etc.).

```
$ kubectl apply --server-side -f https://raw.githubusercontent.com/percona/
percona-postgresql-operator/v2.5.1/deploy/crd.yaml
$ kubectl apply -f https://raw.githubusercontent.com/percona/percona-postgresql-
operator/v2.5.1/deploy/rbac.yaml -n postgres-operator
```

Note

In case of cluster-wide installation, use `deploy/cw-rbac.yaml` instead of `deploy/rbac.yaml`.

2. Now you should Apply a patch to your deployment, supplying necessary image name with a newer version tag. You can find the proper image names and version tags for the current Operator version in the list of certified images. For older versions, please refer to the old releases documentation archive).

   Updating to the `2.5.1` version should look as follows:

```
$ kubectl -n postgres-operator patch deployment percona-postgresql-operator \
   -p'{"spec":{"template":{"spec":{"containers":
[{"name":"operator","image":"percona/percona-postgresql-operator:2.5.1"}]}}}}'
```

3. The deployment rollout will be automatically triggered by the applied patch. You can track the rollout process in real time with the `kubectl rollout status` command with the name of your cluster:

```
$ kubectl rollout status deployments percona-postgresql-operator -n postgres-
operator
```

Expected output

```
deployment "percona-postgresql-operator" successfully rolled out
```

Upgrade via Helm

If you have installed the Operator using Helm, you can upgrade the Operator with the `helm upgrade` command.

Note

You can use `helm upgrade` to upgrade the Operator. But the database management system (Percona Distribution for PostgreSQL) should be upgraded in the same way whether you used Helm to install it or not.

1. Update the [Custom Resource Definition]() for the Operator, taking it from the official repository on Github, and do the same for the Role-based access control:

   ```
   $ kubectl apply --server-side --force-conflicts -f https://
   raw.githubusercontent.com/percona/percona-postgresql-operator/v2.5.1/deploy/
   crd.yaml
   $ kubectl apply -f https://raw.githubusercontent.com/percona/percona-postgresql-
   operator/v2.5.1/deploy/rbac.yaml -n postgres-operator
   ```

2. If you installed the Operator with no [customized parameters](), the upgrade can be done as follows:

   ```
   $ helm upgrade my-operator percona/pg-operator --version 2.5.1
   ```

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

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

   Note

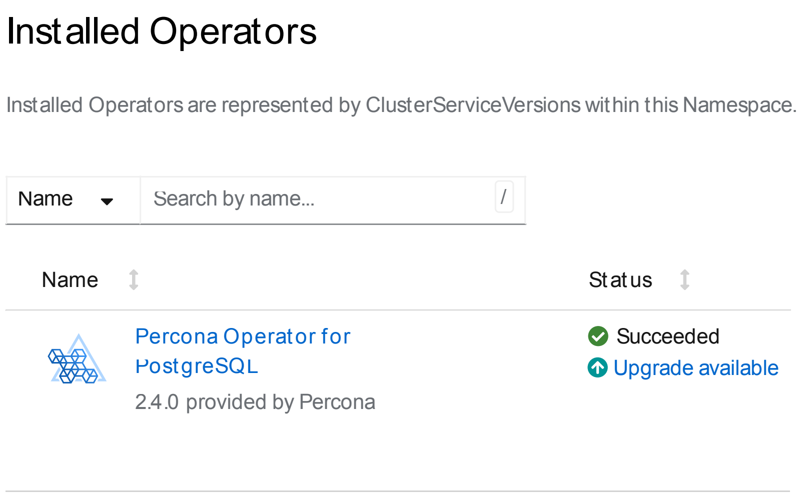
   You can get list of used options in YAML format with the `helm get values my-operator -a > my-values.yaml` command, and this file can be directly passed to the upgrade command as follows:

   ```
   $ helm upgrade my-operator percona/pg-operator --version 2.5.1 -f my-values.yaml
   ```

Upgrade via Operator Lifecycle Manager (OLM)

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

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


Installed Operators

Installed Operators are represented by ClusterServiceVersions within this Namespace.

Name	Status
Percona Operator for PostgreSQL 2.4.0 provided by Percona	Succeeded Upgrade available

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

Upgrading Percona Distribution for PostgreSQL

Before the Operator version 2.4, you could upgrade Percona Distribution for PostgreSQL from one minor version to another (such as upgrading from 15.5 to 15.7, or from 16.1 to 16.3). Starting from the Operator 2.4 you can also upgrade from one PostgreSQL major version to another (for example, upgrade from PostgreSQL 15.5 to PostgreSQL 16.3). Minor version upgrade and major version upgrade are technically different tasks with different scenarios.

Note

Upgrading a PostgreSQL cluster upgrade may result in downtime, as well as failover caused by updating the primary instance.

Minor version upgrade

Upgrading Percona Distribution for PostgreSQL minor version (for example, 16.1 to 16.3) can be done as follows:

1. Apply a patch to your Custom Resource, setting necessary Custom Resource version and image names with a newer version tag.

   Note

   Check the version of the Operator you have in your Kubernetes environment. Please refer to the Operator upgrade guide to upgrade the Operator and CRD first, if needed.

Patching Custom Resource is done with the `kubectl patch pg` command. Actual image names can be found in the list of certified images. For example, updating `cluster1` cluster to the `2.5.1` version should look as follows:

```
$ kubectl -n postgres-operator patch pg cluster1 --type=merge --patch '{
   "spec": {
      "crVersion":"2.5.1",
      "image": "percona/percona-postgresql-operator:2.5.1-ppg16.8-postgres",
      "proxy": { "pgBouncer": { "image": "percona/percona-postgresql-
operator:2.5.1-ppg16.8-pgbouncer1.24.0" } },
      "backups": { "pgbackrest":  { "image": "percona/percona-postgresql-
operator:2.5.1-ppg16.8-pgbackrest2.54.2" } },
      "pmm": { "image": "percona/pmm-client:2.44.0" }
   }}'
```

The following image names in the above example were taken from the list of certified images:

- `percona/percona-postgresql-operator:2.5.1-ppg16.8-postgres`,
- `percona/percona-postgresql-operator:2.5.1-ppg16.8-pgbouncer1.24.0`,
- `percona/percona-postgresql-operator:2.5.1-ppg16.8-pgbackrest2.54.2`,
- `percona/pmm-client:2.44.0`.

Warning

The above command upgrades various components of the cluster including PMM Client. It is highly recommended to upgrade PMM Server **before** upgrading PMM Client. If it wasn't done and you would like to avoid PMM Client upgrade, remove it from the list of images, reducing the last of two patch commands as follows:

```
$ kubectl -n postgres-operator patch pg cluster1 --type=merge --patch '{
   "spec": {
      "crVersion":"2.5.1",
      "image": "percona/percona-postgresql-operator:2.5.1-ppg16.8-postgres",
      "proxy": { "pgBouncer": { "image": "percona/percona-postgresql-operator:2.5.1-
ppg16.8-pgbouncer1.24.0" } },
      "backups": { "pgbackrest":  { "image": "percona/percona-postgresql-operator:2.5.1-
ppg16.8-pgbackrest2.54.2" } }
   }}'
```

The deployment rollout will be automatically triggered by the applied patch. The update process is successfully finished when all Pods have been restarted.

```
$ kubectl get pods -n postgres-operator
```

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
```

Major version upgrade

Major version upgrade allows you to jump from one database major version to another (for example, upgrade from PostgreSQL 15.5 to PostgreSQL 16.3).

Note

Major version upgrades feature is currently a **tech preview**, and it is **not recommended for production environments.**

Also, currently the major version upgrade only works if the images in Custom Resource (`deploy/cr.yaml` manifest) are specified without minor version numbers:

```
...
image: percona/percona-postgresql-operator:2.4.0-ppg15-postgres
postgresVersion: 15
...
```

It will not work for images specified like `percona/percona-postgresql-operator:2.4.0-ppg15.7-postgres`.

The upgrade is triggered by applying the YAML file which refers to the special *Operator upgrade image* and contains the information about the existing and desired major versions. An example of this file is present in `deploy/upgrade.yaml`:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGUpgrade
metadata:
  name: cluster1-15-to-16
spec:
  postgresClusterName: cluster1
  image: percona/percona-postgresql-operator:2.5.1-upgrade
  fromPostgresVersion: 15
  toPostgresVersion: 16
  toPostgresImage: percona/percona-postgresql-operator:2.5.1-ppg16.8-postgres
  toPgBouncerImage: percona/percona-postgresql-operator:2.5.1-ppg16.8-
pgbouncer1.24.0
  toPgBackRestImage: percona/percona-postgresql-operator:2.5.1-ppg16.8-
pgbackrest2.54.2
```

As you can see, the manifest includes image names for the database cluster components (PostgreSQL, pgBouncer, and pgBackRest). You can find them in the list of certified images for the current Operator release. For older versions, please refer to the old releases documentation archive).

After you apply the YAML manifest as usual (by running `kubectl apply -f deploy/upgrade.yaml` command), the actual upgrade takes place:

1. The Operator pauses the cluster, so the cluster will be unavailable for the duration of the upgrade,
2. The cluster is specially annotated with `pgv2.percona.com/allow-upgrade`: `<PerconaPGUpgrade.Name>` annotation,
3. Jobs are created to migrate the data,
4. The cluster starts up after the upgrade finishes.

Note

If the upgrade fails for some reason, the cluster will stay in paused mode. Resume the cluster manually to check what went wrong with upgrade (it will start with the old version). You can check the PerconaPGUpgrade resource with `kubectl get perconapgupgrade -o yaml` command, and check the logs of the upgraded Pods to debug the issue.

During the upgrade data are duplicated in the same PVC for each major upgrade, and old version data are not deleted automatically. Make sure your PVC has enough free space to store data. You can remove data at your discretion by executing into containers and running the following commands (example for PostgreSQL 15):

```
$ rm -rf /pgdata/pg15
$ rm -rf /pgdata/pg15_wal
```

You can also delete the `PerconaPGUpgrade` resource (this will clean up the jobs and Pods created during the upgrade):

```
$ kubectl delete perconapgupgrade cluster1-15-to-16
```

Upgrading PostgreSQL extensions

If there are custom PostgreSQL extensions installed in the cluster, they need to be taken into account: you need to build and package each custom extension for the new PostgreSQL major version. During the upgrade the Operator will install extensions into the upgrade container.

The only built-in extension which demands special treatment after the database upgrade is `pg_stat_monitor` one. It is used to provide query analytics for Percona Monitoring and Management (PMM), if enabled in the Custom Resource (`deploy/cr.yaml` manifest):

```
extensions:
  ...
  builtin:
    pg_stat_monitor: true
    ...
```

If you need it, do the following after the database uprgade (this manual step will be not required for the Operator versions 2.6.0 and newer):

1. Find the primary instance of your PostgreSQL cluster. You can do this using Kubernetes Labels as follows (replace the `<namespace>` placeholder with your value):

   ```
   $ kubectl get pods -n <namespace> -l postgres-operator.crunchydata.com/
   cluster=cluster1 \
       -L postgres-operator.crunchydata.com/instance \
       -L postgres-operator.crunchydata.com/role | grep instance1
   ```

   Sample output

   ```
   cluster1-instance1-bmdp-0              4/4     Running    0          2m23s    cluster1-
   instance1-bmdp    replica
   cluster1-instance1-fm7w-0              4/4     Running    0          2m22s    cluster1-
   instance1-fm7w    replica
   cluster1-instance1-ttm9-0              4/4     Running    0          2m22s    cluster1-
   instance1-ttm9    master
   ```

   PostgreSQL primary is labeled as `master`, while other PostgreSQL instances are labeled as `replica`.

2. Login to a primary instance (`cluster1-instance1-ttm9-0` in the above example) as an administrative user:

```
$ kubectl exec  -n <namespace> -ti cluster1-instance1-ttm9-0 -c database -- psql
postgres
```

3. Execute the following SQL statement:

```
postgres=# alter extension pg_stat_monitor update;
```

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 1.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 - 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 - Minimal downtime during upgrade	Requires significant computing resources to run two clusters in parallel

Management

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:

chown-pod.yaml

```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:

```
$ 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:2.5.1-ppg14-postgres
  postgresVersion: 14
  proxy:
    pgBouncer:
      image: percona/percona-postgresql-operator:2.5.1-ppg14-pgbouncer
  backups:
    pgbackrest:
      image: percona/percona-postgresql-operator:2.5.1-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.

Upgrade from version 1 to version 2

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 brings 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
```

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.

About backups

In this section you will learn how to set up and manage backups of your data using the Operator.

You can make backups in two ways:

- *On-demand*. You can do them manually at any moment.
- *Schedule backups*. Configure backups and their schedule in the deploy/cr.yaml file. The Operator makes them automatically according to the schedule.

What you need to know

Backup repositories

To make backups, the Operator uses the open source pgBackRest backup and restore utility.

When the Operator creates a new PostgreSQL cluster, it also creates a special *pgBackRest repository* to facilitate the usage of the pgBackRest features. You can notice an additional `repo-host` Pod after the cluster creation.

A pgBackRest repository consists of the following Kubernetes objects:

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

In the `/deploy/cr.yml` file, pgBackRest repositories are listed in the `backups.pgbackrest.repos` subsection. You can have up to 4 repositories as `repo1`, `repo2`, `repo3`, and `repo4`.

Backup types

You can make the following types of 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`: Default. A backup of only the files that have changed since the last full or differential backup.

Backup storage

You have the following options to store PostgreSQL backups:

- Cloud storage:
  - Amazon S3, or any S3-compatible storage,
  - Google Cloud Storage,
  - Azure Blob Storage
- A Persistent Volume attached to the pgBackRest Pod.

Next steps

Ready to move forward? Configure backup storage

Back up and restore

Configure backup storage

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

Follow the instructions relevant to the cloud storage or Persistent Volume you are using for backups.

S3-compatible backup storage

To use Amazon S3 or any S3-compatible storage for backups, you need to have the following S3-related information:

- The name of S3 bucket;
- The region - the location of the bucket
- S3 credentials such as S3 key and secret to access the storage. These are stored in an encoded form in Kubernetes Secrets along with other sensitive information.
- For S3-compatible storage other than native Amazon S3, you will also need to specify the endpoint - the actual URI to access the bucket - and the URI style (see below).

Note

The pgBackRest tool does backups based on write-ahead logs (WAL) archiving. If you are using an S3 storage in a region located far away from the region of your PostgreSQL cluster deployment, it could lead to the delay and impossibility to create a new replica/join delayed replica if the primary restarts. A new WAL file is archived in 60 seconds at the backup start by default, causing both full and incremental backups fail in case of long delay.

To prevent issues with PostgreSQL archiving and have faster restores, it's recommended to use the same S3 region for both the Operator and backup options. Additionally, you can replicate the S3 bucket to another region with tools like Amazon S3 Cross Region Replication.

Configuration steps

1. Encode the S3 credentials and the pgBackRest repository name that you will use for backups. In this example, we use AWS S3 key and S3 key secret and `repo2`.

   Linux

   ```
   $ cat <<EOF | base64 --wrap=0
   [global]
   repo2-s3-key=<YOUR_AWS_S3_KEY>
   repo2-s3-key-secret=<YOUR_AWS_S3_KEY_SECRET>
   EOF
   ```

   macOS

   ```
   $ cat <<EOF | base64
   [global]
   repo2-s3-key=<YOUR_AWS_S3_KEY>
   repo2-s3-key-secret=<YOUR_AWS_S3_KEY_SECRET>
   EOF
   ```

Create the Secret configuration file and specify the base64-encoded string from the previous step. The following is the example of the `cluster1-pgbackrest-secrets.yaml` Secret file:

```
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.

3. Create the Secrets object from this YAML file. Replace the `<namespace>` placeholder with your value:

```
$ kubectl apply -f cluster1-pgbackrest-secrets.yaml -n <namespace>
```

4. Update your `deploy/cr.yaml` configuration. Specify the Secret file you created in the `backups.pgbackrest.configuration` subsection, and put all other S3 related information in the `backups.pgbackrest.repos` subsection under the repository name that you intend to use for backups. This name must match the name you used when you encoded S3 credentials on step 1.

   Provide pgBackRest the directory path for backup on the storage. You can pass it in the backups.pgbackrest.global subsection via the pgBackRest `path` option (prefix it's name with the repository name, for example `repo1-path`). Also, if your S3-compatible storage requires additional repository options for the pgBackRest tool, you can specify these parameters in the same `backups.pgbackrest.global` subsection with standard pgBackRest option names, also prefixed with the repository name.

Amazon S3 storage

For example, the S3 storage for the `repo2` repository looks as follows:

```
...
backups:
  pgbackrest:
    ...
    configuration:
      - secret:
          name: cluster1-pgbackrest-secrets
    ...
    global:
      repo2-path: /pgbackrest/postgres-operator/cluster1/repo2
    ...
    repos:
    - name: repo2
      s3:
        bucket: "<YOUR_AWS_S3_BUCKET_NAME>"
        region: "<YOUR_AWS_S3_REGION>"
```

Using AWS EC2 instances for backups makes it possible to automate access to AWS S3 buckets based on IAM roles for Service Accounts with no need to specify the S3 credentials explicitly.

To use this feature, add annotation to the spec part of the Custom Resource and also add pgBackRest custom configuration option to the backups subsection as follows:

```
spec:
  crVersion: 2.5.1
  metadata:
    annotations:
      eks.amazonaws.com/role-arn: arn:aws:iam::1191:role/role-pgbackrest-access-s3-bucket
  ...
  backups:
    pgbackrest:
      image: percona/percona-postgresql-operator:2.5.1-ppg16-pgbackrest
      global:
        repo2-s3-key-type: web-id
```

S3-compatible storage

For example, the S3-compatible storage for the `repo2` repository looks as follows:

```
...
backups:
  pgbackrest:
    ...
    configuration:
      - secret:
          name: cluster1-pgbackrest-secrets
    ...
    global:
      repo2-path: /pgbackrest/postgres-operator/cluster1/repo2
      repo2-storage-verify-tls=y
      repo2-s3-uri-style: path
    ...
    repos:
    - name: repo2
      s3:
        bucket: "<YOUR_AWS_S3_BUCKET_NAME>"
        endpoint: "<YOUR_AWS_S3_ENDPOINT>"
        region: "<YOUR_AWS_S3_REGION>"
```

The `repo2-storage-verify-tls` option in the above example enables TLS verification for pgBackRest (when set to `y` or simply omitted) or disables it, when set to `n`.

The `repo2-s3-uri-style` option [should be set to `path`](#) if you use S3-compatible storage (otherwise you might see "host not found error" in your backup job logs), and is not needed for Amazon S3.

5. Create or update the cluster. Replace the `<namespace>` placeholder with your value:

To u

- a

  b

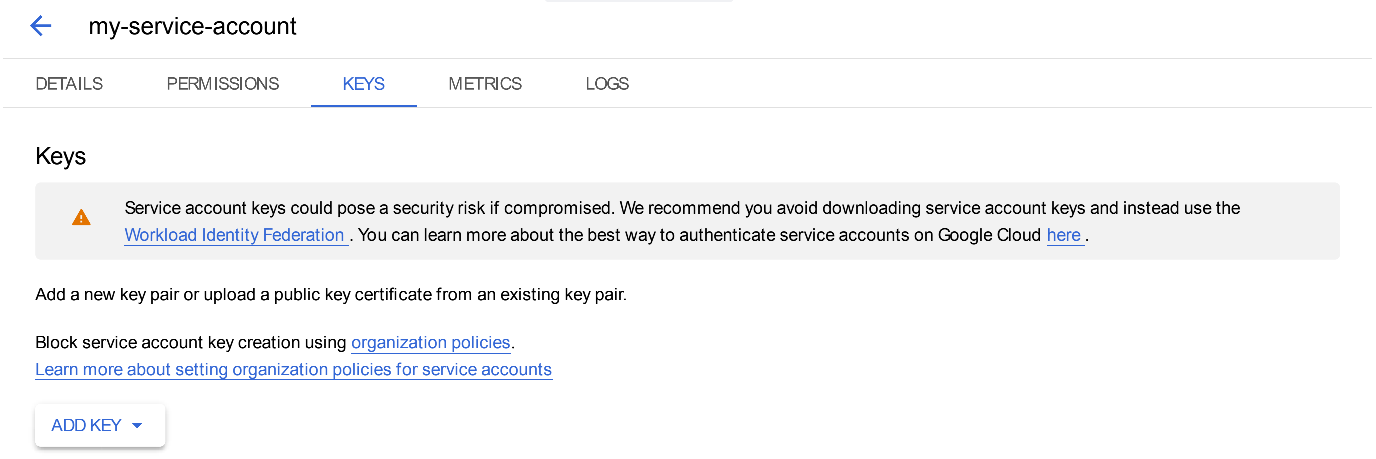
- yo

Configur

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, choose *Create new key* and choose *JSON* as a key type. These actions will result in downloading a file in JSON format with your new private key and related information (for example, `gcs-key.json`).

3. Create the [Kubernetes Secret](#). The Secret consists of base64-encoded versions of two files: the `gcs-key.json` file with the Google service account key you have just downloaded, and the special `gcs.conf` configuration file.

Create the `gcs.conf` configuration file. The file contents depends on the repository name for backups in the `deploy/cr.yaml` file. In case of the `repo3` repository, it looks as follows:

```
[global]
repo3-gcs-key=/etc/pgbackrest/conf.d/gcs-key.json
```

Encode both `gcs-key.json` and `gcs.conf` files.

Linux

```
base64 --wrap=0 <filename>
```

MacOS

```
base64 -i <filename>
```

Create the Kubernetes Secret configuration file and specify your cluster name and the base64-encoded contents of the files from previous steps. The following is the example of the `cluster1-pgbackrest-secrets.yaml` Secret file:

```
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>
```

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

4. Create the Secrets object from the Secret configuration file. Replace the `<namespace>` placeholder with your value:

```
$ kubectl apply -f cluster1-pgbackrest-secrets.yaml -n <namespace>
```

5. Update your `deploy/cr.yaml` configuration. Specify your GCS credentials Secret in the `backups.pgbackrest.configuration` subsection, and put GCS bucket name into the `bucket` option in the `backups.pgbackrest.repos` subsection. The repository name must be the same as the name you specified when you created the `gcs.conf` file.

   Also, provide pgBackRest the directory path for backup on the storage. You can pass it in the [backups.pgbackrest.global](#) subsection via the pgBackRest `path` option (prefix it's name with the repository name, for example `repo3-path`).

   For example, GCS storage configuration for the `repo3` repository would look as follows:

```
...
backups:
  pgbackrest:
    ...
    configuration:
      - secret:
          name: cluster1-pgbackrest-secrets
    ...
    global:
      repo3-path: /pgbackrest/postgres-operator/cluster1/repo3
    ...
    repos:
    - name: repo3
      gcs:
        bucket: "<YOUR_GCS_BUCKET_NAME>"
```

Azure Blob Storage (tech preview)

6. Create or update the cluster. Replace the `<namespace>` placeholder with your value:

To use [Microsoft Azure Blob Storage](#) for storing backups, you need the following:

- a
```
$ kubectl apply -f deploy/cr.yaml -n <namespace>
```
- Azure Storage credentials. These are stored in an encoded form in the Kubernetes Secret.

Configuration steps

1. Encode the Azure Storage credentials and the pgBackRest repo name that you will use for backups with base64. In this example, we are using `repo4`.

   Linux

   ```
   $ cat <<EOF | base64 --wrap=0
   [global]
   repo4-azure-account=<AZURE_STORAGE_ACCOUNT_NAME>
   repo4-azure-key=<AZURE_STORAGE_ACCOUNT_KEY>
   EOF
   ```

   macOS

   ```
   $ cat <<EOF | base64
   [global]
   repo4-azure-account=<AZURE_STORAGE_ACCOUNT_NAME>
   repo4-azure-key=<AZURE_STORAGE_ACCOUNT_KEY>
   EOF
   ```

2. Create the Secret configuration file and specify the base64-encoded string from the previous step. The following is the example of the `cluster1-pgbackrest-secrets.yaml` Secret file:

   ```
   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.

3. Create the Secrets object from this yaml file. Replace the `<namespace>` placeholder with your value:

   ```
   $ kubectl apply -f cluster1-pgbackrest-secrets.yaml -n <namespace>
   ```

4.

Update your deploy/cr.yaml configuration. Specify the Secret file you have created in the previous step in the `backups.pgbackrest.configuration` subsection. Put Azure container name in the `backups.pgbackrest.repos` subsection under the repository name that you intend to use for backups. This name must match the name you used when you encoded Azure credentials on step 1.

Also, provide pgBackRest the directory path for backup on the storage. You can pass it in the backups.pgbackrest.global subsection via the pgBackRest `path` option (prefix it's name with the repository name, for example `repo4-path`).

For example, the Azure storage for the `repo4` repository looks as follows.

```
...
backups:
  pgbackrest:
    ...
    configuration:
      - secret:
          name: cluster1-pgbackrest-secrets
    ...
    global:
      repo4-path: /pgbackrest/postgres-operator/cluster1/repo4
    ...
    repos:
    - name: repo4
      azure:
        container: "<YOUR_AZURE_CONTAINER>"
```

5 Create or update the cluster. Replace the `<namespace>` placeholder with your value:

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

Persistent Volume

Percona Operator for PostgreSQL uses Kubernetes Persistent Volumes to store Postgres data. You can also use them to store backups. A Persistent volume is created at the same time when the Operator creates PostgreSQL cluster for you. You can find the Persistent Volume configuration in the `backups.pgbackrest.repos` section of the `cr.yaml` file under the `repo1` name:

```
        ...
backups:
  pgbackrest:
    ...
    global:
      repo1-path: /pgbackrest/postgres-operator/cluster1/repo1
    ...
    repos:
    - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - ReadWriteOnce
            resources:
              requests:
                storage: 1Gi
```

This configuration is sufficient to make a backup.

Next steps

- Make an on-demand backup
- Make a scheduled backup

Make scheduled backups

Backups schedule is defined on the 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 that you specify in crontab format.

1. Before you start, make sure you have configured a backup storage.

2. Configure backup schedule in the `deploy/cr.yaml` file. The schedule is specified in crontab format as explained in Custom Resource options. The repository name must be the same as the one you defined in the backup storage configuration. The following example shows the schedule for `repo1` repository:

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

1. Update the cluster:

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

Next steps

Restore from a backup

Useful links

Backup retention

Making on-demand backups

To make an on-demand backup manually, you need a backup configuration file. You can use the example of the backup configuration file deploy/backup.yaml:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGBackup
metadata:
  name: backup1
spec:
  pgCluster: cluster1
  repoName: repo1
#  options:
#  - --type=full
```

Here's a sequence of steps to follow:

1. Before you start, make sure you have configured a backup storage.

2. In the `deploy/backup.yaml` configuration file, specify the cluster name and the repository name to be used for backups. The repository name must be the same as the one you defined in the backup storage configuration. It must also match the repository name specified in the `backups.pgbackrest.manual` subsection of the `deploy/cr.yaml` file.

3. If needed, you can add any pgBackRest command line options.

4. Make a backup with the following command (modify the `-n postgres-operator` parameter if your database cluster resides in a different namespace):

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

   Expected output

   ```
   perconapgbackup.pgv2.percona.com/backup1 created
   ```

5. Making a backup takes time. You can track the process with `kubectl get pg-backup` command. When finished, backup should obtain the `Succeeded` status:

   ```
   $ kubectl get pg-backup backup1 -n postgres-operator
   ```

Expected output

```
NAME       CLUSTER    REPO     DESTINATION   STATUS      TYPE   COMPLETED   AGE
backup1    cluster1   repo1                  Succeeded   incr   3m38s       3m53s
```

Tip

To list available backups, run:

```
$ kubectl get pg-backup -n postgres-operator
```

Next steps

Restore from a backup

Useful links

Backup retention

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 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).

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 an active one, or a former cluster whose pgBackRest repository still exists, edit the dataSource.postgresCluster subsection options in the Custom Resource manifest of the *new cluster* (the one you are going to create). 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 source cluster name,
- `dataSource.postgresCluster.clusterNamespace` should contain the namespace of the source cluster (it is needed if the new cluster will be created in a different namespace, and **you will need the Operator deployed in multi-namespace/cluster-wide mode to make such cross-namespace restore**),
- `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.

The following example bootstraps a new cluster from a backup, which was made on the `cluster1` cluster deployed in `percona-db-1` namespace. For simplicity, this backup uses `repo1` repository from the Persistent Volume backup storage example, which needs no cloud credentials. The resulting `deploy/cr.yaml` manifest for the *new* cluster should contain the following lines:

```
...
dataSource:
  postgresCluster:
    clusterName: cluster1
    repoName: repo1
    clusterNamespace: percona-db-1
...
```

Creating the new cluster in its namespace (for example, `percona-db-2`) with such a manifest will initiate the restoration process:

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

Restore to an existing PostgreSQL cluster

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

```
apiVersion: pgv2.percona.com/v2
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.

To start the restoration process, run the following command (modify the `-n postgres-operator` parameter if your database cluster resides in a different namespace):

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

Specifying which backup to restore

When there are multiple backups, the Operator will restore the latest full backup by default.

if you want to restore to some previous backup, not the last one, follow these steps:

1. Find the label of the backup you want to restore. For this, you can list available backups with `kubectl get pg-backup` command, and then get detailed information about the backup of your interest with `kubectl describe pg-backup <BACKUP NAME>`. The output should look as follows:

   ```
   Name:         cluster1-backup-c55w-f858g
   Namespace:    default
   Labels:       <none>
   Annotations:  pgv2.percona.com/pgbackrest-backup-job-name: cluster1-backup-c55w
                 pgv2.percona.com/pgbackrest-backup-job-type: replica-create
   API Version:  pgv2.percona.com/v2
   Kind:         PerconaPGBackup
   Metadata:
     Creation Timestamp:  2024-06-28T07:44:08Z
     Generate Name:       cluster1-backup-c55w-
     Generation:          1
     Resource Version:    1199
     UID:                 92a8193c-6cbd-4cdf-82e5-a4623bf7f2d9
   Spec:
     Pg Cluster:  cluster1
     Repo Name:   repo1
   Status:
     Backup Name:  20240628-074416F
     Backup Type:  full
   ...
   ```

   The "Backup Name" status field will contain needed backup label.

2. Now use a *backup restore* configuration file with additional `--set=<backup_label>` pgBackRest option. For example, the following yaml file will result in restoring to a backup labeled `20240628-074416F`:

   ```
   apiVersion: pgv2.percona.com/v2
   kind: PerconaPGRestore
   metadata:
     name: restore1
   spec:
     pgCluster: cluster1
     repoName: repo1
     options:
     - --type=immediate
     - --set=20240628-074416F
   ```

3. Start the restoration process, as usual:

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

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 UTC),
- optional `--set` argument followed with a pgBackRest backup ID allows you to choose the backup which will be the starting point for point-in-time recovery. This option must be specified if the target is one or more backups away from the current moment. You can look through the available backups with the pgBackRest info command to find out the proper backup ID.

pgBackRest backup ID example

After obtaining the Pod name with `kubectl get pods` command, you can run `pgbackrest --stanza=db info` command on the appropriate Pod as follows:

```
$ kubectl -n postgres-operator exec -it cluster1-instance1-hcgr-0 -c database -- pgbackrest
--stanza=db info
```

Then find ID of the needed backup in the output:

```
stanza: db
    status: ok
    cipher: none

    db (prior)
        wal archive min/max (16): 0000000F000000000000001C/000000200000000360000000C5

        full backup: 20240401-173403F
            timestamp start/stop: 2024-04-01 17:34:03+00 / 2024-04-01 17:36:57+00
            wal start/stop: 000000120000000000000022 / 000000120000000000000024
            database size: 31MB, database backup size: 31MB
            repo1: backup set size: 4.1MB, backup size: 4.1MB

        incr backup: 20240401-173403F_20240415-201250I
            timestamp start/stop: 2024-04-15 20:12:50+00 / 2024-04-15 20:14:19+00
            wal start/stop: 00000019000000000000005C / 00000019000000000000005D
            database size: 46.0MB, database backup size: 25.7MB
            repo1: backup set size: 6.1MB, backup size: 3.8MB
            backup reference list: 20240401-173403F

        incr backup: 20240401-173403F_20240415-201430I
...
```

Now you can put this backup ID to the *backup restore* configuration file as follows:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGRestore
metadata:
  name: restore1
spec:
  pgCluster: cluster1
  repoName: repo1
  options:
  - --set="20240401-173403F"
```

The example may look as follows:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGRestore
metadata:
  name: restore1
spec:
  pgCluster: cluster1
  repoName: repo1
  options:
  - --type=time
  - --target="2022-11-30 15:12:11+03"
```

Note

Latest succeeded backup available with the `kubectl get pg-backup` command has a "Latest restorable time" information field handy when selecting a backup to restore. Tracking latest restorable time is turned on by default, and you can easily query the backup for this information as follows:

```
$ kubectl get pg-backup <backup_name> -n postgres-operator -o
jsonpath='{.status.latestRestorableTime}'
```

After setting these options in the *backup restore* configuration file, start the restoration process:

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

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.

Fix the cluster if the restore fails

The restore process changes database files, and therefore restoring wrong information or causing restore fail by misconfiguring can put the database cluster in non-operational state.

For example, adding wrong pgBackRest arguments to `PerconaGPRestore` custom resource breaks existing database installation while the restore hangs.

In this case it's possible to remove the *restore annotation* from the Custom Resource correspondent to your cluster. Supposing that your cluster `cluster1` was deployed in `postgres-operator` namespace, you can do it with the following command:

```
$ kubectl annotate -n postgres-operator pg cluster1 postgres-
operator.crunchydata.com/pgbackrest-restore-
```

Alternatively, you can temporarily delete the database cluster by removing the Custom Resource (check the `finalizers.percona.com/delete-pvc` finalizer is not turned on, otherwise you will not retain your data!), and recreate the cluster back by running `kubectl apply -f deploy/cr.yaml -n postgres-operator` command you have used to deploy the it previously.

One more reason of failed restore to consider is the possibility of a corrupted backup repository or missing files. In this case, you may need to delete the database cluster by removing the Custom Resource, find the startup PVC to delete it and recreate again.

Configure backup encryption

Backup encryption is a security best practice that helps protect your organization's confidential information and prevents unauthorized access.

The pgBackRest tool used by the Operator allows encrypting backups using AES-256 encryption. The approach is **repository-based**: pgBackRest encrypts the whole repository where it stores backups. Encryption is enabled if a user-supplied encryption key was passed to pgBackRest with the `-repo-cypher-pass` option *when configuring the backup storage*.

⚠ **Limitation:** You cannot change encryption settings after the backups are established. You must create a new repository to enable encryption or change the encryption key.

This document describes how to configure backup encryption.

Generate the encryption key

You should use a long, random encryption key. You can generate it using OpenSSL as follows:

```
$ openssl rand -base64 48
```

Configure backup storage

Follow the general backup storage configuration instruction relevant to the backup storage you are using. The only difference is in encoding your cloud credentials and the pgBackRest repository name to be used for backups: you also add the encryption key to the configuration file as the `repo-cipher-pass` option. The repo name within the option must match the pgBackRest repo name.

The following example shows the configuration for S3-compatible storage and the pgBackRest repo name `repo2` (other cloud storages are configured similarly).

1. Encode the storage configuration file.

Linux

```
$ cat <<EOF | base64 --wrap=0
[global]
repo2-s3-key=<YOUR_AWS_S3_KEY>
repo2-s3-key-secret=<YOUR_AWS_S3_KEY_SECRET>
repo2-cipher-pass=<YOUR_ENCRYPTION_KEY>
EOF
```

macOS

```
$ cat <<EOF | base64
[global]
repo2-s3-key=<YOUR_AWS_S3_KEY>
repo2-s3-key-secret=<YOUR_AWS_S3_KEY_SECRET>
repo2-cipher-pass=<YOUR_ENCRYPTION_KEY>
EOF
```

2. Create the Secrets configuration file and the Secrets object as described in steps 2-3 of the S3-compatible backup storage configuration. Follow the instructions relevant to the backup storage you are using.

3. Update the `deploy/cr.yaml` configuration. Specify the following information:

   - The Secret name you created in the `backups.pgbackrest.configuration` subsection
   - All storage-related information in the `backups.pgbackrest.repos` subsection under the repository name that you intend to use for backups. This name must match the name you used when you encoded S3 credentials on step 1.
   - The cipher type in the `pgbackrest.global` subsection

   The following example shows the configuration for the S3-compatible storage and the pgBackRest repo name `repo2`:

```
backups:
  pgbackrest:
    ...
    configuration:
      - secret:
          name: cluster1-pgbackrest-secrets
    ...
    repos:
    - name: repo2
      s3:
        bucket: "<YOUR_AWS_S3_BUCKET_NAME>"
        endpoint: "<YOUR_AWS_S3_ENDPOINT>"
        region: "<YOUR_AWS_S3_REGION>"
    global:
      cipher-type: aes-256-cbc
```

4. Apply the changes. Replace the `<namespace>` placeholder with your value.

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

Make a backup

Make an on-demand backup

Make a scheduled backup

Speed-up backups with pgBackRest asynchronous archiving

Backing up a database with high write-ahead logs (WAL) generation can be rather slow, because PostgreSQL archiving process is sequential, without any parallelism or batching. In extreme cases backup can be even considered unsuccessful by the Operator because of the timeout.

The pgBackRest tool used by the Operator can, if necessary, solve this problem by using the WAL asynchronous archiving feature.

You can set up asynchronous archiving in your storage configuration file for pgBackRest. Turn on the additional `archive-async` flag, and set the `process-max` value for `archive-push` and `archive-get` commands. Your storage configuration file may look as follows:

s3.conf

```
[global]
repo2-s3-key=REPLACE-WITH-AWS-ACCESS-KEY
repo2-s3-key-secret=REPLACE-WITH-AWS-SECRET-KEY
repo2-storage-verify-tls=n
repo2-s3-uri-style=path
archive-async=y
spool-path=/pgdata

[global:archive-get]
process-max=2

[global:archive-push]
process-max=4
```

No modifications are needed aside of setting these additional parameters. You can find more information about WAL asynchronous archiving in gpBackRest official documentation and in this blog post.

Backup retention

The Operator 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 the write-ahead log. Thus, the expiration of a full backup with some incremental backups based on it results in expiring of all these incremental backups.

You can control backup retention 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 to keep a backup for).

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

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

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.

Vertical scaling

Scale compute

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.

Scale storage

Kubernetes manages storage with a PersistentVolume (PV), a segment of storage supplied by the administrator, and a PersistentVolumeClaim (PVC), a request for storage from a user. In Kubernetes v1.11 the feature was added to allow a user to increase the size of an existing PVC object (considered stable since Kubernetes v1.24). The user cannot shrink the size of an existing PVC object.

Scaling with Volume Expansion capability

Certain volume types support PVCs expansion (exact details about PVCs and the supported volume types can be found in [Kubernetes documentation](#)).

You can run the following command to check if your storage supports the expansion capability:

```
$ kubectl describe sc <storage class name> | grep AllowVolumeExpansion
```

Expected output

```
AllowVolumeExpansion: true
```

The Operator versions 2.5.0 and higher will automatically expand such storage for you when you change the appropriate options in the Custom Resource.

For example, you can do it by editing and applying the `deploy/cr.yaml` file:

```
spec:
  ...
  instances:
    ...
    dataVolumeClaimSpec:
      resources:
        requests:
          storage: <NEW STORAGE SIZE>
```

Apply changes as usual:

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

Automated scaling with auto-growable disk

The Operator 2.5.0 and newer is able to detect if the storage usage on the PVC reaches a certain threshold, and trigger the PVC resize. Such autoscaling needs the upstream "auto-growable disk" feature turned on when deploying the Operator. This is done via the `PGO_FEATURE_GATES` environment variable set in the `deploy/operator.yaml` manifest (or in the appropriate part of `deploy/bundle.yaml`):

```
...
subjects:
- kind: ServiceAccount
  name: percona-postgresql-operator
  namespace: pg-operator
...
spec:
  containers:
  - env:
    - name: PGO_FEATURE_GATES
      value: "AutoGrowVolumes=true"
...
```

When the support for auto-growable disks is turned on, the auto grow will be working automatically if the maximum value available for the Operator to scale up is set in the `spec.instances[].dataVolumeClaimSpec.resources.limits.storage` Custom Resource option:

```
spec:
  ...
  instances:
    ...
    dataVolumeClaimSpec:
      resources:
        requests:
          storage: 1Gi
        limits:
          storage: 5Gi
```

High availability

Percona Operator allows you to deploy highly-available PostgreSQL clusters. High-availability implementation is based on the Patroni template, which uses PostgreSQL streaming replication. The cluster includes a number of replicas, one of which is a primary PostgreSQL instance: it is available for writes, and streams changes to other replicas (*standby servers* in terms of PostgreSQL). Streaming replication used in this configuration is *asynchronous* by default, which means transferring data to a different instance without waiting for a confirmation of its receiving. Alternatively, a *synchronous* replication can be used, where the data transfer waits for a confirmation of its successful processing on the standby. If the primary server crashes then some transactions that were committed may not have been replicated to the standby server, causing data loss (the amount of data loss is proportional to the replication delay at the time of failover). Synchronous replication is slower but minimizes the data loss possibility in case if the primary server crash.

There are two ways how to control the number replicas in your HA cluster:

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

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.

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
```

Using Synchronous replication

Synchronous replication offers the ability to confirm that all changes made by a transaction have been transferred to one or more synchronous standby servers. When requesting synchronous replication, each commit of a write transaction will wait until confirmation is received that the commit has been written to the write-ahead log on disk of both the primary and standby server. The drawbacks of synchronous replication are increased latency and reduced throughput on writes.

You can turn on synchronous replication by customizing the `patroni.dynamicConfiguration` Custom Resource option.

- Enable synchronous replication by setting `synchronous_mode` option to `on`.
- Use `synchronous_node_count` option to set the number of replicas (PostgreSQL standby servers) which should operate in syncrhonous mode (the default value is `1`).

The result in your `deploy/cr.yaml` manifest may look as follows:

```
...
  patroni:
    dynamicConfiguration:
      synchronous_mode: "on"
      synchronous_node_count: 2
      ...
```

You will have the desired amount of replicas switched to synchronous replication after applying changes as usual, with `kubectl apply -f deploy/cr.yaml` command.

Find more options useful to tune how your database cluster should operate in synchronous mode in the official Patroni documentation.

Using sidecar containers

The Operator allows you to deploy additional (so-called *sidecar*) containers to the Pod. You can use this feature to run debugging tools, some specific monitoring solutions, etc.

Note

Custom sidecar containers can easily access other components of your cluster.

Therefore they should be used carefully and by experienced users only.

Adding a sidecar container

You can add sidecar containers to 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
```

Obviously, you cannot name your sidecar container by duplicating an already existing container name in the Pod. Use `kubectl describe pod` command to check which names are already in use. For example, PostgreSQL instance Pods cannot have custom sidecar containers named as `database`, `pgbackrest`, `pgbackrest-config`, and `replication-cert-copy`.

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)
....
```

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
/ #
```

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
```

Monitor with Percona Monitoring and Management (PMM)

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

Note

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

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

A PMM Client collects needed metrics and sends gathered data to the PMM Server. As a user, you connect to the PMM Server to see database metrics on a number of dashboards. PMM Server and PMM Client are installed separately.

Install PMM Server

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

Install PMM Client

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

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

From PMM UI

Generate the PMM API key

From command line

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

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

> **Note**
>
> The API key is not rotated.

2. Specify the API key as the `PMM_SERVER_KEY` value in the deploy/secrets.yaml secrets file.

```
apiVersion: v1
kind: Secret
metadata:
  name: cluster1-pmm-secret
type: Opaque
stringData:
  PMM_SERVER_KEY: ""
```

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

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

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

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

```
  pmm:
    enabled: true
    image: percona/pmm-client:2.44.0
#    imagePullPolicy: IfNotPresent
    secret: cluster1-pmm-secret
    serverHost: monitoring-service
```

5. Update the cluster

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

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

```
$ kubectl get pods -n postgres-operator
$ kubectl logs <pod_name> -c pmm-client
```

Update the secrets file

The `deploy/secrets.yaml` file contains all values for each key/value pair in a convenient plain text format. But the resulting Secrets Objects contains passwords stored as base64-encoded strings. If you want to *update* the password field, you need to encode the new password into the base64 format and pass it to the Secrets Object.

To encode a password or any other parameter, run the following command:

Linux

```
$ echo -n "password" | base64 --wrap=0
```

macOS

```
$ echo -n "password" | base64
```

For example, to set the new PMM API key in the `my-cluster-name-secrets` object, do the following:

Linux

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

macOS

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

Check the metrics

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

1. Log in to PMM server.
2. Click **PostgreSQL** from the left-hand navigation menu. You land on the **Instances Overview** page.
3. Click **PostgreSQL** → **Other dashboards** to see the list of available dashboards that allow you to drill down to the metrics you are interested in.

Install Percona Distribution for PostgreSQL with customized parameters

You can customize the configuration of Percona Distribution for PostgreSQL and install it with customized parameters.

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

kubectl

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

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

   ```
   $ git clone -b v2.5.1 https://github.com/percona/percona-postgresql-operator
   ```

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

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

Helm

To install Percona Distribution for PostgreSQL with custom parameters using Helm, use the following command:

```
$ helm install --set key=value
```

You can pass any of the Operator's Custom Resource options as a `--set key=value[,key=value]` argument.

The following example deploys a PostgreSQL 16.8 based cluster in the `my-namespace` namespace, with enabled Percona Monitoring and Management (PMM):

```
$ helm install my-db percona/pg-db --version 2.5.1 --namespace my-namespace \
  --set postgresVersion=16.8 \
  --set pmm.enabled=true
```

How-to

How to run initialization SQL commands at cluster creation time

The Operator can execute a custom sequence of PostgreSQL commands when creating the databse cluster. This sequence can include both SQL commands and meta-commands of the PostgreSQL interactive shell (psql). This feature may be useful to push any customizations to the cluster: modify user roles, change error handling, set and use variables, etc.

psql interactive terminal will execute these initialization statements when the cluster is created, after creating custom users and databases specifed in the Custom Resource.

To set SQL initialization sequence you need creating a special ConfigMap with it, and reference this ConfigMap in the `databaseInitSQL` subsection of your Custom Resource options.

The following example uses initialization SQL command to add a new role to a PostgreSQL database cluster:

1. Create YAML manifest for the ConfigMap as follows:

   my_init.yaml

   ```
   apiVersion: v1
   kind: ConfigMap
   metadata:
     name: cluster1-init-sql
     namespace: postgres-operator
   data:
     init.sql: CREATE ROLE someonenew WITH createdb superuser login password
   'someonenew';
   ```

   The `namespace` field should point to the namespace of your database cluster, and the `init.sql` key contains the sequence of commands, which will be passed to the psql.

   Create the ConfigMap by applying your manifest:

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

2. Update the `databaseInitSQL` part of the `deploy/cr.yaml` Custom Resource manifest as follows:

   ```
   ...
   databaseInitSQL:
     key: init.sql
     name: cluster1-init-sql
   ...
   ```

   Now, SQL commands will be executed when you create the cluster by apply the manifest:

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

The psql command is executed the standard input and the file flag (`psql -f -`). If the command returns `0` exit code, SQL will not be run again. When psql returns with an error exit code, the Operator will continue attempting to execute it as part of its reconcile loop until success. You can fix errors in the SQL sequence, for example by interactive `kubectl edit configmap cluster1-init-sql -n postgres-namespace` command.

Note

You can use following psql meta-command to make sure that any SQL errors would make psql to return the error code:

```
\set ON_ERROR_STOP
\echo Any error will lead to exit code 3
```

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.

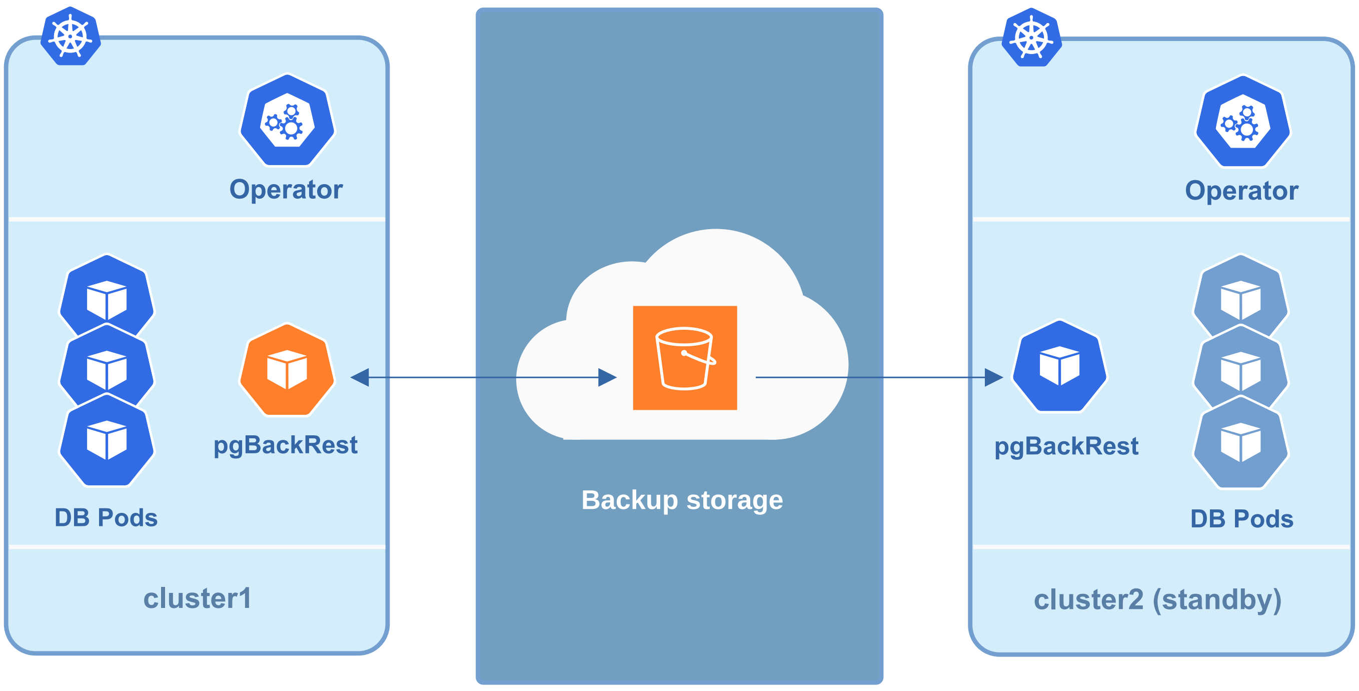
Operators automate routine tasks and remove toil. For standby, the Percona Operator for PostgreSQL version 2 provides the following options:

1. pgBackrest repo based standby. The standby cluster will be connected to a pgBackRest cloud repo, so it will receive WAL files from the repo and apply them to the database.
2. Streaming replication. The standby cluster will use an authenticated network connection to the primary cluster to receive WAL records directly.
3. Combination of (1) and (2). The standby cluster is configured for both repo-based standby and streaming replicaton. It bootstraps from the pgBackRest repo and continues to receive WAL files as they are pushed to the repo, and can also directly receive them from primary. Using this approach ensures the cluster will still be up to date with the pgBackRest repo if streaming falls behind.

Deploy a standby cluster for Disaster Recovery

Standby cluster deployment based on pgBackRest

The pgBackRest repo-based standby is the simplest one. The following is the architecture diagram:



pgBackrest repo based standby

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

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):

   ```
   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):

```yaml
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:

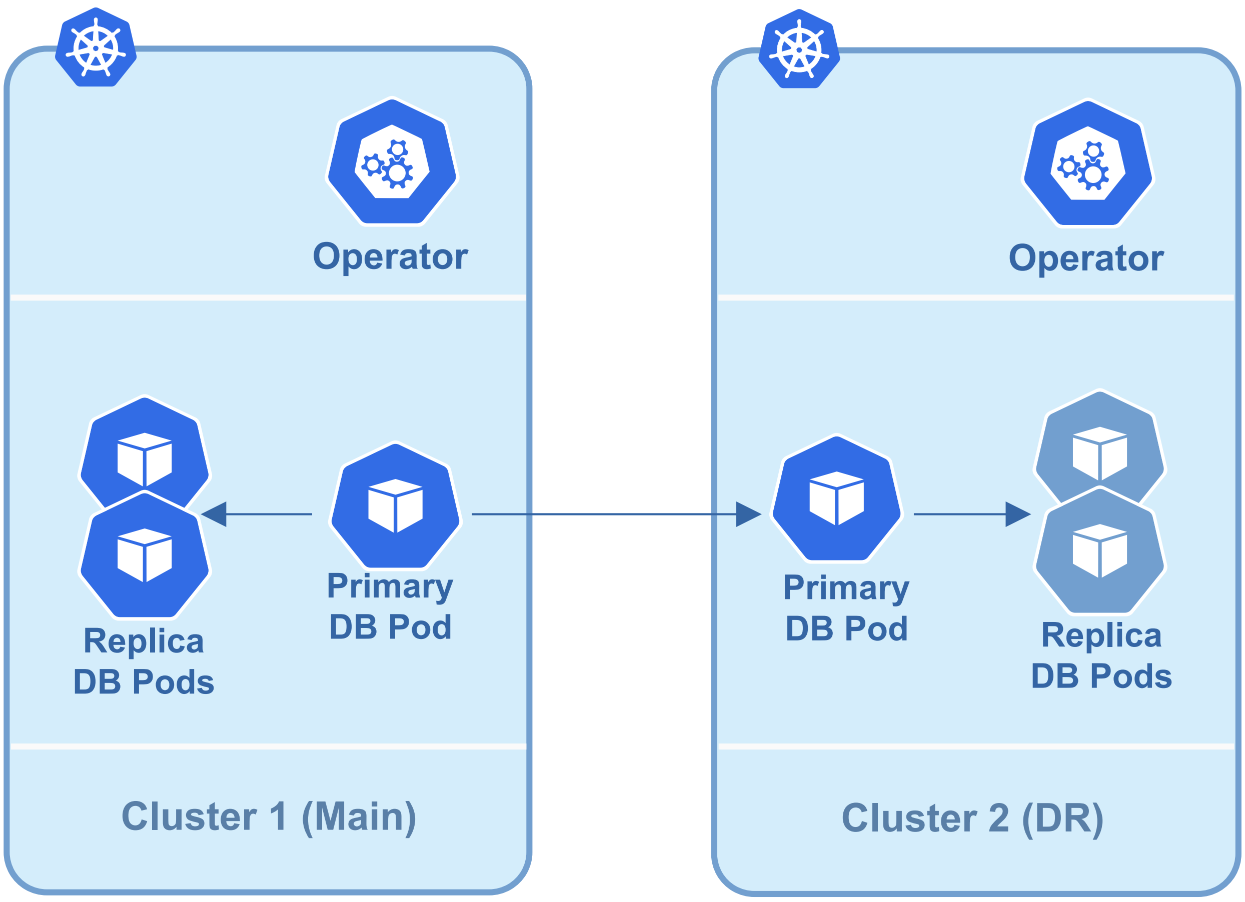
```
$ kubectl apply -f deploy/cr.yaml
```

Expected output

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

Standby cluster deployment based on streaming replication

The following diagram explains how the standby based on streaming replication works:



1. This solution describes two Kubernetes clusters in different regions, clouds, data centers or even two namespaces, or running in hybrid mode (on-premises and cloud). One cluster is Main site, and the other is Disaster Recovery site (DR)
2. Each site supposedly includes Percona Operator and for sure includes PostgreSQL cluster.
3. In the DR site the cluster is in Standby mode
4. We set up streaming replication between these two clusters

Deploy disaster recovery for PostgreSQL on Kubernetes

Configure Main site

1. Deploy the Operator using your favorite method.
2. The Main cluster needs to expose it, so that standby can connect to the primary PostgreSQL instance. To expose the primary PostgreSQL instance, use the `spec.expose` section:

   ```
   spec:
     ...
     expose:
       type: ClusterIP
   ```

   Use here a Service type of your choice. For example, `ClusterIP` is sufficient for two clusters in different Kubernetes namespaces.
3. Once configured, apply the custom resource:

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

   Expected output

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

   The service that you should use for connecting to standby is called -ha (main-ha in my case):

   ```
   main-ha          ClusterIP   10.118.227.214   <none>        5432/TCP   163m
   ```

Configure DR site

To get the replication working, the Standby cluster would need to authenticate with the Main one. To get there, both clusters must have certificates signed by the same certificate authority (CA). Default replication user `_crunchyrepl` will be used.

In the simplest case you can copy the certificates from the Main cluster. You need to look out for two files:

- main-cluster-cert
- main-replication-cert

Copy them to the namespace where DR cluster is going to be running and reference under `spec.secrets` (in the following example they were renamed, replacing “main” with “dr”):

```
spec:
  secrets:
    customTLSSecret:
      name: dr-cluster-cert
    customReplicationTLSSecret:
      name: dr-replication-cert
```

If you are generating your own certificates, just remember the following rules:

1. Certificates for both Main and Standby clusters must be signed by the same CA
2. `customReplicationTLSSecret` must have a Common Name (CN) setting that matches `_crunchyrepl`, which is a default replication user.

You can find more about certificates in the TLS doc.

Apart from setting certificates correctly, you should also set standby configuration.

```
  standby:
    enabled: true
    host: main-ha.main-pg.svc
```

- `standby.enabled` controls if it is a standby cluster or not
- `standby.host` must point to the primary node of a Main cluster. In this example it is a `main-ha` Service in another namespace.

Deploy the standby cluster by applying the manifest:

```
$ kubectl apply -f dr-cr.yaml -n dr-pg
```

Expected output

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

Once both clusters are up, you can verify that replication is working.

1. Insert some data into Main cluster
2. Connect to the DR cluster

To connect to the DR cluster, use the credentials that you used to connect to Main. This also verifies that the connection is working. You should see whatever data you have in the Main cluster in the Disaster Recovery.

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

Change the PostgreSQL primary instance

The Operator uses PostgreSQL high-availability implementation based on the Patroni template. This means that each PostgreSQL cluster includes one member availiable for read/write transactions (PostgreSQL primary instance, or leader in terms of Patroni) and a number of replicas which can serve read requests only (standby members of the cluster).

You may wish to manually change the primary instance in your PostgreSQL cluster to achieve more control and meet specific requirements in various scenarios like planned maintenance, testing failover procedures, load balancing and performance optimization activities. Primary instance is re-elected during the automatic failover (Patroni's "leader race" mechanism), but still there are use cases to controll this process manually.

In Percona Operator, the primary instance election can be controlled by the `patroni.switchover` section of the Custom Resource manifest. It allows you to enable switchover targeting a specific PostgreSQL instance as the new primary, or just running a failover if PostgreSQL cluster has entered a bad state.

This document provides instructions how to change the primary instance manually.

For the following steps, we assume that you have the PostgreSQL cluster up and running. The cluster name is `cluster1`.

1. Check the information about the cluster instances. Cluster instances are defined in the `spec.instances` Custom Resource section. By default you have one cluster instance named `instance1` with 3 PostgreSQL instances in it. You can check which cluster instances you have. Do this using Kubernetes Labels as follows (replace the `<namespace>` placeholder with your value):

   ```
   $ kubectl get pods -n <namespace> -l postgres-operator.crunchydata.com/
   cluster=cluster1 \
       -L postgres-operator.crunchydata.com/instance \
       -L postgres-operator.crunchydata.com/role | grep instance1
   ```

   Sample output

   ```
   cluster1-instance1-bmdp-0            4/4     Running   0          2m23s   cluster1-
   instance1-bmdp   replica
   cluster1-instance1-fm7w-0            4/4     Running   0          2m22s   cluster1-
   instance1-fm7w   replica
   cluster1-instance1-ttm9-0            4/4     Running   0          2m22s   cluster1-
   instance1-ttm9   master
   ```

   PostgreSQL primary is labeled as `master`, while other PostgreSQL instances are labeled as `replica`.

2. Now update the following options in the `patroni.switchover` subsection of the Custom Resource:

```
patroni:
  switchover:
    enabled: true
    targetInstance: <instance-name>
```

You can do it with `kubectl patch` command, specifying the name of the instance that you want to be the new primary. For example, let's set the `cluster1-instance1-bmdp` as a new PostgreSQL primary:

```
$ kubectl -n <namespace> patch pg cluster1 --type=merge --patch '{
  "spec": {
    "patroni": {
      "switchover": {
        "enabled": true,
        "targetInstance": "cluster1-instance1-bmdp"
      }
    }
  }
}'
```

3. Trigger the switchover by adding the annotation to your Custom Resource. The recommended way is to set the annotation with the timestamp, so you know when switchover took place. Replace the `<namespace>` placeholder with your value:

```
$ kubectl annotate --overwrite -n <namespace> pg cluster1 postgres-
operator.crunchydata.com/trigger-switchover="$(date)"
```

The `--overwrite` flag in the above command allows you to overwrite the annotation if it already exists (useful if that's not the first switchover you do).

4. Verify that the cluster was annotated (replace the `<namespace>` placeholder with your value, as usual):

```
$ kubectl get pg cluster1 -o yaml -n <namespace>
```

Sample output

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGCluster
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {....
      "patroni":{"switchover":{"enabled":true,"targetInstance":"cluster1-instance1-
bmdp"}},}
```

5. Now, check instances of your cluster once again to make sure the switchover took place:

```
$ kubectl get pods -n <namespace> -l postgres-operator.crunchydata.com/
cluster=cluster1 \
    -L postgres-operator.crunchydata.com/instance \
    -L postgres-operator.crunchydata.com/role | grep instance1
```

Sample output

```
cluster1-instance1-bmdp-0              4/4     Running     0          24m   cluster1-
instance1-bmdp   master
cluster1-instance1-fm7w-0              4/4     Running     0          24m   cluster1-
instance1-fm7w   replica
cluster1-instance1-ttm9-0              4/4     Running     0          23m   cluster1-
instance1-ttm9   replica
```

6. Set `patroni.switchover.enabled` Custom Resource option to `false` once the switchover is done:

```
$ kubectl -n <namespace> patch pg cluster1 --type=merge --patch '{
  "spec": {
    "patroni": {
      "switchover": {
        "enabled": false
      }
    }
  }
}'
```

Use Docker images from a private registry

Using images from a private Docker registry may be required for privacy, security or other reasons. In these cases, Percona Operator for PostgreSQL allows the use of a custom registry. The following example illustrates how this can be done by the example of the Operator deployed in the OpenShift environment.

Prerequisites

1. First of all login to the OpenShift and create project.

    ```
    $ oc login
    Authentication required for https://192.168.1.100:8443 (openshift)
    Username: admin
    Password:
    Login successful.
    $ oc new-project pg
    Now using project "pg" on server "https://192.168.1.100:8443".
    ```

2. There are two things you will need to configure your custom registry access:

    - the token for your user,
    - your registry IP address.

    The token can be found with the following command:

    ```
    $ oc whoami -t
    ADO8CqCDappWR4hxjfDqwijEHei31yXAvWg61Jg210s
    ```

    And the following one tells you the registry IP address:

    ```
    $ kubectl get services/docker-registry -n default
    NAME              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
    docker-registry   ClusterIP   172.30.162.173   <none>        5000/TCP   1d
    ```

3. Use the user token and the registry IP address to login to the registry:

    ```
    $ docker login -u admin -p ADO8CqCDappWR4hxjfDqwijEHei31yXAvWg61Jg210s
    172.30.162.173:5000
    ```

Expected output

```
Login Succeeded
```

4. Use the Docker commands to pull the needed image by its SHA digest:

```
$ docker pull docker.io/perconalab/percona-postgresql-
operator@sha256:991d6049059e5eb1a74981290d829a5fb4ab0554993748fde1e67b2f46f26bf0
```

Expected output

```
Trying to pull repository docker.io/perconalab/percona-postgresql-operator ...
sha256:991d6049059e5eb1a74981290d829a5fb4ab0554993748fde1e67b2f46f26bf0: Pulling from
docker.io/perconalab/percona-server-mongodb
Digest: sha256:991d6049059e5eb1a74981290d829a5fb4ab0554993748fde1e67b2f46f26bf0
Status: Image is up to date for docker.io/perconalab/percona-postgresql-
operator@sha256:991d6049059e5eb1a74981290d829a5fb4ab0554993748fde1e67b2f46f26bf0
```

You can find correct names and SHA digests in the current list of the Operator-related images officially certified by Percona.

5. The following method can push an image to the custom registry for the example OpenShift `pg` project:

```
$ docker tag \
    docker.io/perconalab/percona-postgresql-
operator@sha256:991d6049059e5eb1a74981290d829a5fb4ab0554993748fde1e67b2f46f26bf0
\
    172.30.162.173:5000/psmdb/percona-postgresql-operator:16.8
$ docker push 172.30.162.173:5000/pg/percona-postgresql-operator:16.8
```

6. Verify the image is available in the OpenShift registry with the following command:

```
$ oc get is
```

Expected output

```
NAME                             DOCKER REPO
TAGS              UPDATED
percona-postgresql-operator      docker-registry.default.svc:5000/pg/percona-postgresql-
operator   16.8   2 hours ago
```

7.

When the custom registry image is available, edit the the `image:` option in `deploy/operator.yaml` configuration file with a Docker Repo + Tag string (it should look like `docker-registry.default.svc:5000/pg/percona-postgresql-operator:16.8`)

> **Note**
>
> If the registry requires authentication, you can specify the `imagePullSecrets` option for all images.

8. Repeat steps 3-5 for other images, and update corresponding options in the `deploy/cr.yaml` file.
9. Now follow the standard Percona Operator for PostgreSQL [installation instruction](#).

Add custom PostgreSQL extensions

One of the specific PostgreSQL features is the ability to provide it with additional functionality via Extensions. Percona Distribution for PostgreSQL supports a number of extensions , making this list available for the database cluster managed by the Operator as well.

Still there are cases when the needed extension is not in this list, or when it's a custom extension developed by the end-user. Adding more extensions is not an easy task in case of a containerized database in Kubernetes-based environment, as normally it would make the user build a custom PostgreSQL image.

Still, starting from the Operator version 2.3 there is an alternative way to extend Percona Distribution for PostgreSQL by downloading prepackaged extensions from an external storage on the fly, as defined in the `extensions` section of the Operator Custom Resource.

Enabling or disabling built-in extensions

Built-in extensions are enabled by default, but can be easily disabled in the `extensions.builtin` subsection of the `deploy/cr.yaml` configuration file. To disable a built-in extension, you need to explicitly set the apporpriate option to `false`. Enabling means setting the option to `true` or simply omitting it:

```
extensions:
  ...
  builtin:
    pg_stat_monitor: false
    pg_audit: false
```

Apply changes after editing with `kubectl apply -f deploy/cr.yaml` command.

Note

Editing this section and applying it will cause the Pods to restart.

Adding custom extensions

Custom extensions are downloaded by the Operator from the cloud storage. User is in charge for properly packaging extension and uploading it to the storage.

Packaging custom extensions

Custom extension needs specific packaging to make the Operator able using it. The package must be a `.tar.gz` archive with all required files in a the correct directory structure.

1. Control file must be in `SHAREDIR/extension` directory
2. All required SQL script files must be in `SHAREDIR/extension` directory (there must be at least one SQL script)
3. Any shared library must be in `LIBDIR`

Note

In case of Percona Distribution for PostgreSQL images, `SHAREDIR` corresponds to `/usr/pgsql-${PG_MAJOR}/share` and `LIBDIR` to `/usr/pgsql-${PG_MAJOR}/lib`.

For example, the directory for `pg_cron` extension should look as follows:

```
$ tree ~/pg_cron-1.6.1/
/home/user/pg_cron-1.6.1/
└── usr
    └── pgsql-15
        ├── lib
        │   └── pg_cron.so
        └── share
            └── extension
                ├── pg_cron--1.0--1.1.sql
                ├── pg_cron--1.0.sql
                ├── pg_cron--1.1--1.2.sql
                ├── pg_cron--1.2--1.3.sql
                ├── pg_cron--1.3--1.4.sql
                ├── pg_cron--1.4--1.4-1.sql
                ├── pg_cron--1.4-1--1.5.sql
                ├── pg_cron--1.5--1.6.sql
                └── pg_cron.control
```

The archive must be created with `usr` at the root and the name must conform `${EXTENSION}-pg${PG_MAJOR}-${EXTENSION_VERSION}`:

```
$ cd pg_cron-1.6.1/
$ tar -czf pg_cron-pg15-1.6.1.tar.gz usr/
```

Note

To understand which files are required for given extension could be not an easy task. One of the option to figure this out would be building and installing the extension from source on a virtual machine with Percona Distribution for PostgreSQL and copy all the installed files to the archive.

Configuring custom extension loading

When the extension is packaged, it should be uploaded to the cloud storage (for now, Amazon S3 is the only supported storage type). When the upload is done, the needed access credentials for the cloud storage should be placed in a Secret, and both the storage and extension details should be specified in the Custom Resource to make the Operator download and install it.

1. Create the Secrets file with the credentials, which the Operator will need to access extensions stored on the Amazon S3:

   - the `metadata.name` key is the name which you will further use to refer your Kubernetes Secret,
   - the `data.AWS_ACCESS_KEY_ID` and `data.AWS_SECRET_ACCESS_KEY` keys are base64-encoded credentials used to access the storage (obviously these keys should contain proper values to make the access possible).

   Create the Secrets file with these base64-encoded keys as follows:

   extensions-secret.yaml

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: cluster1-extensions-secret
   type: Opaque
   data:
     AWS_ACCESS_KEY_ID: <base64 encoded secret>
     AWS_SECRET_ACCESS_KEY: <base64 encoded secret>
   ```

   Note

   You can use the following command to get a base64-encoded string from a plain text one:

   in Linux

   For GNU/Linux:

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

   in macOS

   For Apple macOS:

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

   Once the editing is over, create the Kubernetes Secret object as follows:

   ```
   $ kubectl apply -f extensions-secret.yaml
   ```

2. Storage credentials are specified in the Custom Resource `extensions.storage` subsection. The appropriate fragment of the `deploy/cr.yaml` configuration file should look as follows:

   ```
   extensions:
     ...
     storage:
       type: s3
       bucket: pg-extensions
       region: eu-central-1
       endpoint: s3.eu-central-1.amazonaws.com
       secret:
         name: cluster1-extensions-secret
   ```

3. When the storage is configured, and the archive with the extension is already present in the appropriate bucket, the extension itself can be specified to the Operator in the Custom Resource via the `deploy/cr.yaml` configuration file as in the following example:

   ```
   extensions:
     ...
     custom:
     - name: pg_cron
       version: 1.6.1
   ```

The installed extension will not be enabled by default. Enabling it in can be done for desired databases using the `CREATE EXTENSION` statement:

```
CREATE EXTENSION pg_cron;
```

Also, some extensions (such as `pg_cron`) can be used only if added to `shared_preload_libraries`. Users can do it via the `deploy/cr.yaml` configuration file as follows:

```
...
patroni:
  dynamicConfiguration:
    postgresql:
      parameters:
        shared_preload_libraries: pg_cron
        ...
```

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

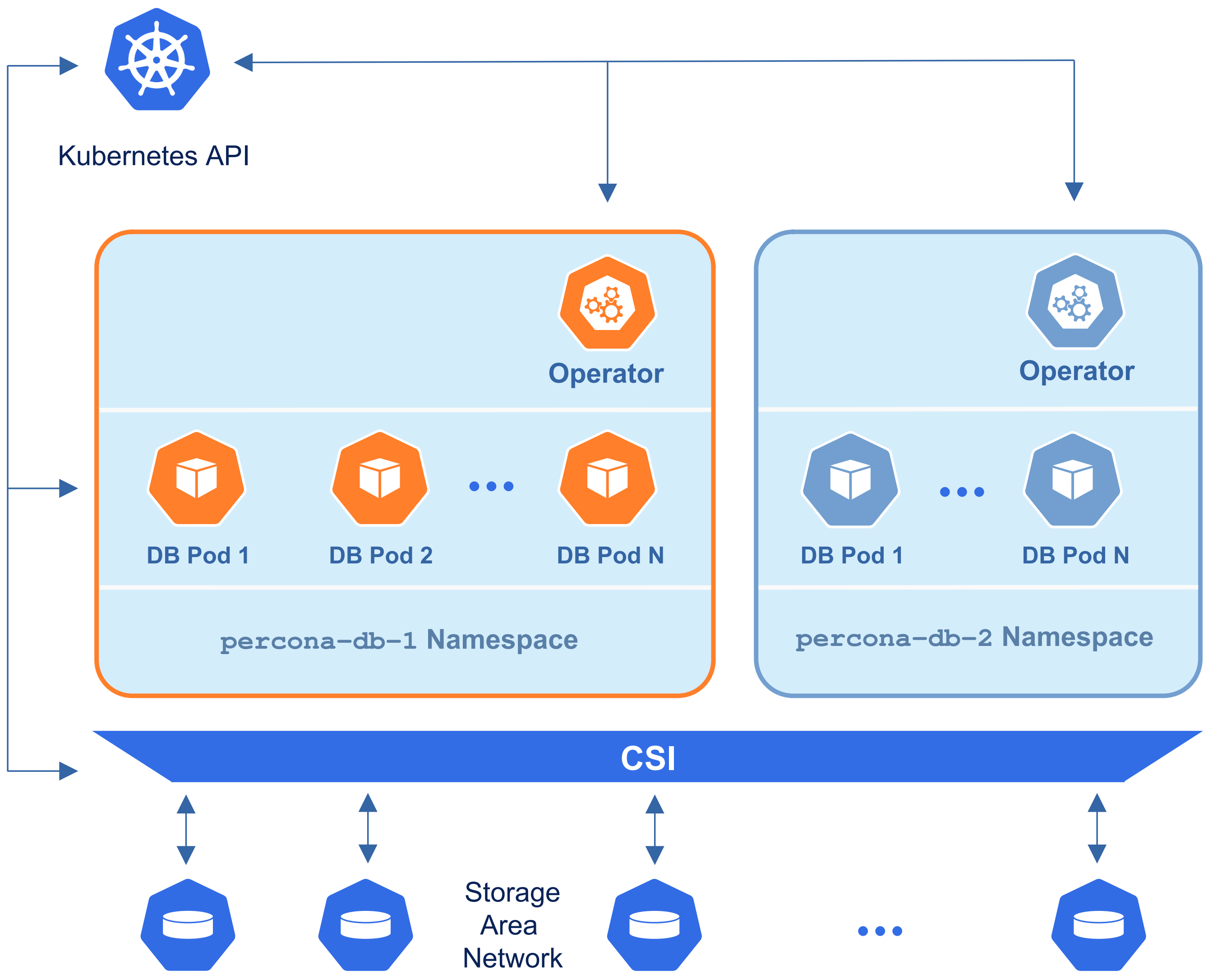
There are two design patterns that you can choose from when deploying Percona Operator for PostgreSQL and PostgreSQL clusters in Kubernetes:

- 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.

Namespace-scope

By default, Percona Operator for PostgreSQL functions in a specific Kubernetes namespace. You can create one during the installation (like it is shown in the installation instructions) or just use the default namespace. This approach allows several Operators to co-exist in one Kubernetes-based environment, being separated in different namespaces:



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

Let's say you will use a Kubernetes Namespace called `percona-db-1`.

1. Clone `percona-postgresql-operator` repository:

   ```
   $ git clone -b v2.5.1 https://github.com/percona/percona-postgresql-operator
   $ cd percona-postgresql-operator
   ```

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

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

3. Deploy the Operator using the following command:

   ```
   $ kubectl apply --server-side -f deploy/bundle.yaml -n percona-db-1
   ```

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

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

You can deploy multiple clusters in this namespace.

Add more namespaces

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

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

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

2. Deploy the Operator:

   ```
   $ kubectl apply --server-side -f deploy/bundle.yaml -n percona-db-2
   ```

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

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

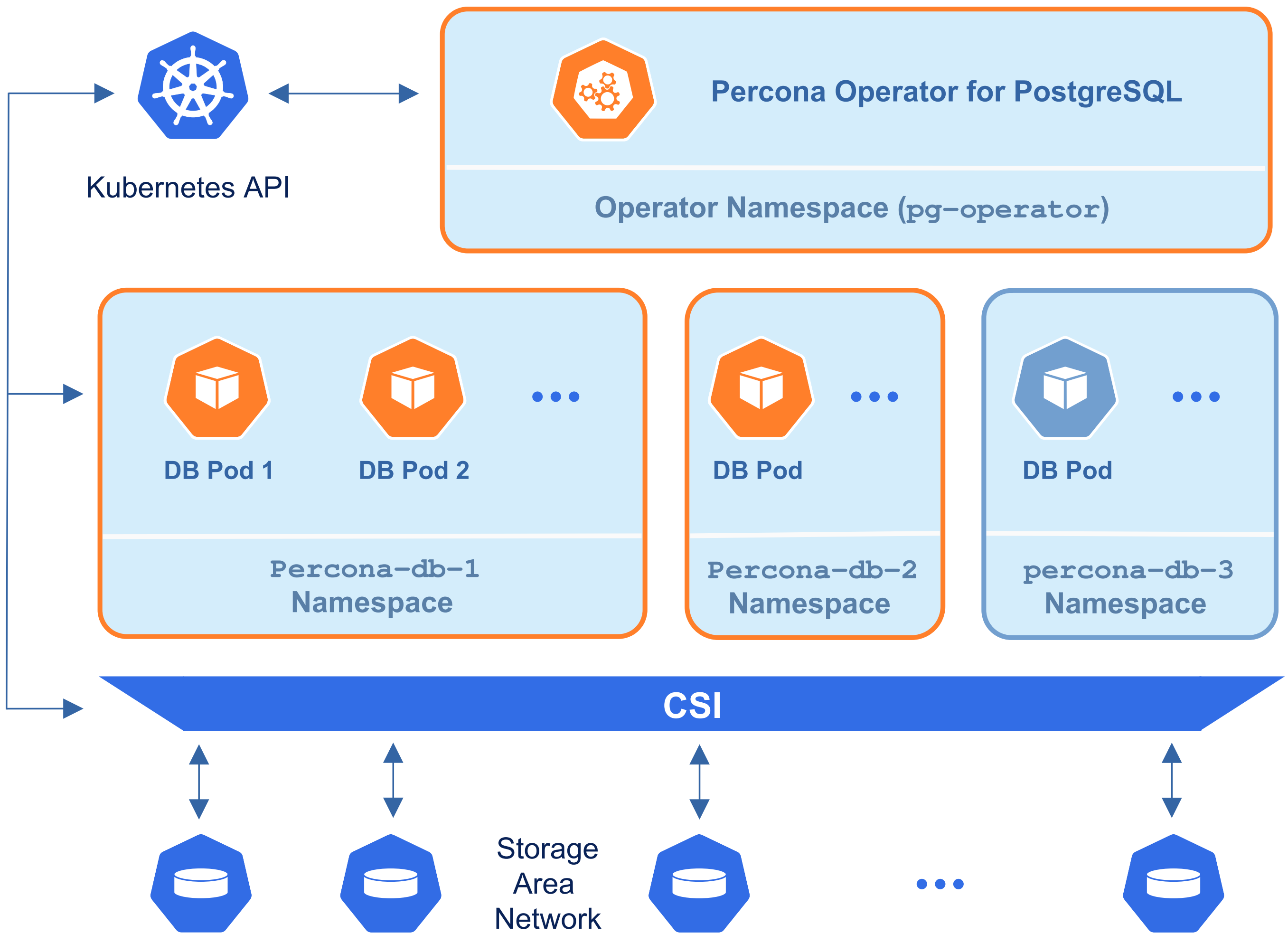
   Note

   Cluster names may be the same in different namespaces.

Install the Operator cluster-wide

Sometimes it is more convenient to have one Operator watching for Percona 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. If this key is set to a blank string, the Operator will watch **only the namespace it runs in**, which would be the same as single-namespace deployment.

The following simple example shows how to install Operator cluster-wide on Kubernetes.

1. Clone `percona-postgresql-operator` repository:

```
$ git clone -b v2.5.1 https://github.com/percona/percona-postgresql-operator
$ cd percona-postgresql-operator
```

2. Let's say you will use `pg-operator` namespace for the Operator, and `percona-db-1` namespace for the cluster. Create these namespaces, if needed:

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

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

   ```
   ...
   subjects:
   - kind: ServiceAccount
     name: percona-postgresql-operator
     namespace: pg-operator
   ...
   spec:
     containers:
     - env:
       - name: WATCH_NAMESPACE
         value: "pg-operator,percona-db-1"
   ...
   ```

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

   ```
   $ kubectl apply --server-side -f deploy/cw-bundle.yaml -n 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. Deploy the cluster in the namespace of your choice:

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

Verifying the cluster operation

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

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

❶

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`.

2. Use the following command to get the password of this user. Replace the `<cluster_name>` and `<namespace>` placeholders with your values:

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

3. Create a pod and start Percona Distribution for PostgreSQL inside. The following command will do this, naming the new Pod `pg-client`:

```
$ kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-
postgresql:16.8 --restart=Never -- bash -il
```

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

4. Run a container with `psql` tool and connect its console output to your terminal. The following command will connect you as a `cluster1` user to a `cluster1` database via the PostgreSQL interactive terminal.

```
[postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-
pgbouncer.postgres-operator.svc -p 5432 -U cluster1 cluster1
```

Sample output

```
psql (16.8)
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression:
off)
Type "help" for help.
pgdb=>
```

Using PostgreSQL tablespaces with Percona Operator for PostgreSQL

Tablespaces allow DBAs to store a database on multiple file systems within the same server and to control where (on which file systems) specific parts of the database are stored. You can think about it as if you were giving names to your disk mounts and then using those names as additional parameters when creating database objects.

PostgreSQL supports this feature, allowing you to *store data outside of the primary data directory*, and Percona Operator for PostgreSQL is a good option to bring this to your Kubernetes environment when needed.

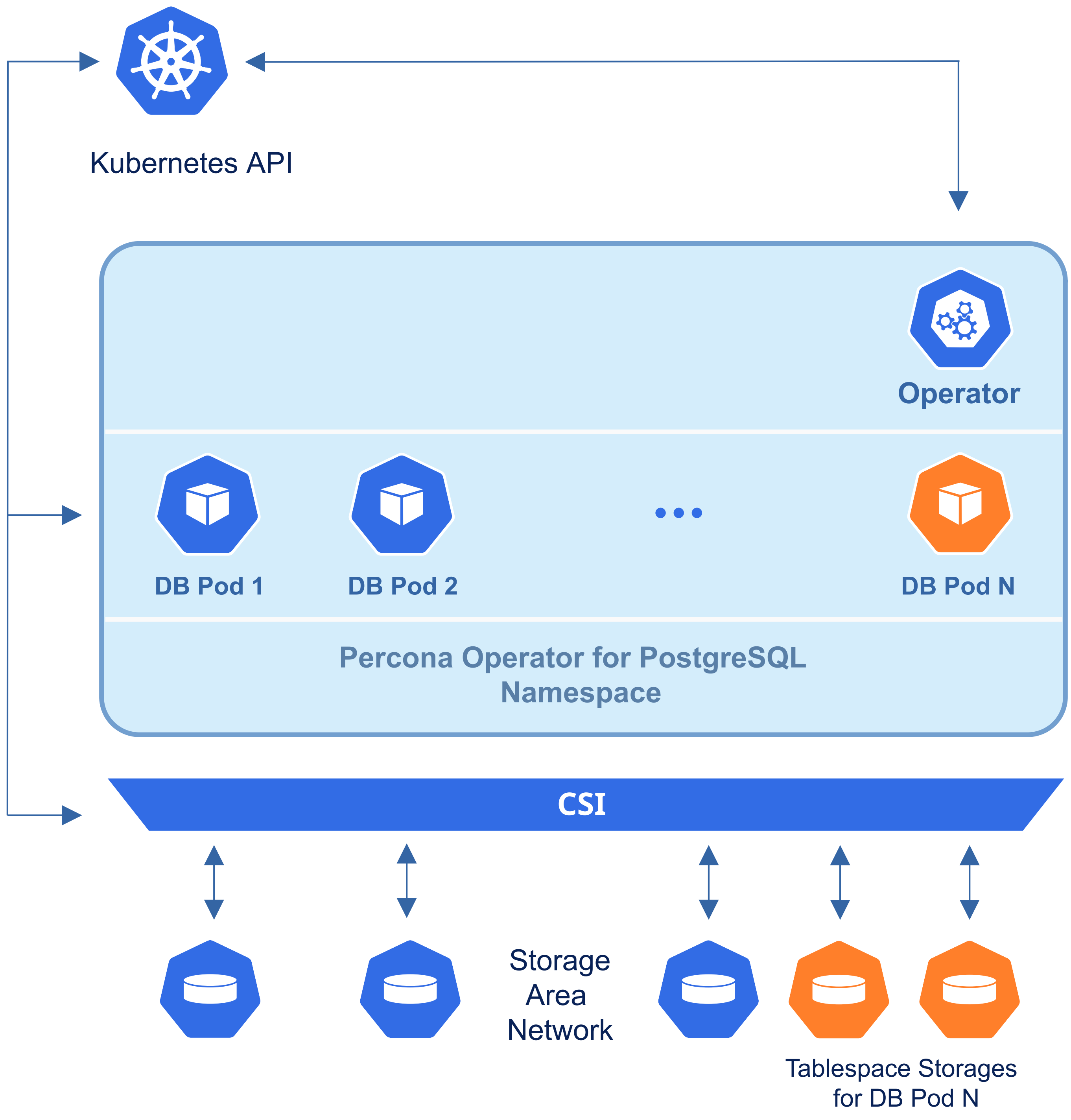
Possible use cases

The most obvious use case for tablespaces is performance optimization. You place appropriate parts of the database on fast but expensive storage and engage slower but cheaper storage for lesser-used database objects. The classic example would be using an SSD for heavily-used indexes and using a large slow HDD for archive data.

Of course, the Operator already provides you with traditional Kubernetes approaches to achieve this on a per-Pod basis (Tolerations, etc.). But if you would like to go deeper and make such differentiation at the level of your database objects (tables and indexes), tablespaces are exactly what you would need for that.

Another well-known use case for tablespaces is quickly adding a new partition to the database cluster when you run out of space on the initially used one and cannot extend it (which may look less typical for cloud storage). Finally, you may need tablespaces when migrating your existing architecture to the cloud.

Each tablespace created by Percona Operator for PostgreSQL corresponds to a separate Persistent Volume, mounted in a container to the `/tablespaces` directory.


Kubernetes API
Operator
DB Pod 1
DB Pod 2
DB Pod N
Percona Operator for PostgreSQL Namespace
CSI
Storage Area Network
Tablespace Storages for DB Pod N

Creating a new tablespace

Providing a new tablespace for your database in Kubernetes involves two parts:

1. Configure the new tablespace storage with the Operator,
2. Create database objects in this tablespace with PostgreSQL.

The first part is done in the traditional way of Percona Operators, by modifying Custom Resource via the `deploy/cr.yaml` configuration file. It has a special spec.tablespaceStorages section for tablespaces.

The example already present in `deploy/cr.yaml` shows how to create tablespace storage 1Gb in size (you can see official Kubernetes documentation on Persistent Volumes for details):

```
spec:
  instances:
    ...
    tablespaceVolumes:
      - name: user
        dataVolumeClaimSpec:
          accessModes:
            - 'ReadWriteOnce'
          resources:
            requests:
              storage: 1Gi
```

After you apply this by running the `kubectl apply -f deploy/cr.yaml` command, the new `/tablespaces/user/` mountpoint will appear for your database. Please take into account that if you add your new tablespace to the already existing PostgreSQL cluster, it may take time for the Operator to create Persistent Volume Claims and get Persistent Volumes actually mounted.

Now you should actually create your tablespace on this volume with the `CREATE TABLESPACE <tablespace name> LOCATION <mount point>` command, and then create objects in it (of course, your user should have appropriate `CREATE` privileges to make it possible):

```
CREATE TABLESPACE user121
LOCATION '/tablespaces/user/data';
```

Now when the tablespace is created you can append `TABLESPACE <tablespace_name>` to your `CREATE` SQL statements to implicitly create tables, indexes, or even entire databases in specific tablespace.

Let's create an example table in the already mentioned `user121` tablespace:

```
CREATE TABLE products (
    product_sku character(10),
    quantity int,
    manufactured_date timestamptz)
TABLESPACE user121;
```

It is also possible to set a default tablespace with the `SET default_tablespace = <tablespace_name>;` statement. It will affect all further `CREATE TABLE` and `CREATE INDEX` commands without an explicit tablespace specifier, until you unset it with an empty string.

As you can see, Percona Operator for PostgreSQL simplifies tablespace creation by carrying on all necessary modifications with Persistent Volumes and Pods. The same would not be true for the deletion of an already existing tablespace, which is not automated, neither by the Operator nor by PostgreSQL.

Deleting an existing tablespace

Deleting an existing tablespace from your database in Kubernetes also involves two parts:

- Delete related database objects and tablespace with PostgreSQL,
- Delete tablespace storage in Kubernetes.

To make tablespace deletion with PostgreSQL possible, you should make this tablespace empty (it is impossible to drop a tablespace until *all objects in all databases using this tablespace* have been removed). Tablespaces are listed in the `pg_tablespace` table, and you can use it to find out which objects are stored in a specific tablespace. The example command for the `lake` tablespace will look as follows:

```
SELECT relname FROM pg_class WHERE reltablespace=(SELECT oid FROM pg_tablespace
WHERE spcname='user121');
```

When your tablespace is empty, you can log in to the *PostgreSQL Primary instance* as a *superuser*, and then execute the `DROP TABLESPACE <tablespace_name>;` command.

Now, when the PostgreSQL part is finished, you can remove the tablespace entry from the `tablespaceStorages` section (don't forget to run the `kubectl apply -f deploy/cr.yaml` command to apply changes).

Delete Percona Operator for PostgreSQL

When cleaning up your Kubernetes environment (e.g., moving from a trial deployment to a production one, or testing experimental configurations), you may need to remove some (or all) of the following objects:

- Percona Distribution for PosgreSQL cluster managed by the Operator
- Percona Operator for PostgreSQL itself
- Custom Resource Definition deployed with the Operator

Delete a database cluster

You can delete the Percona Distribution for PosgreSQL cluster managed by the Operator by deleting the appropriate Custom Resource.

Note

There are two finalizers defined in the Custom Resource, which define whether TLS-related objects and data volumes should be deleted or preserved when the cluster is deleted.

- `finalizers.percona.com/delete-ssl`: if present, objects, created for SSL (Secret, certificate, and issuer) are deleted when the cluster deletion occurs.
- `finalizers.percona.com/delete-pvc`: if present, Persistent Volume Claims for the database cluster Pods are deleted when the cluster deletion occurs.

Both finalizers are off by default in the `deploy/cr.yaml` configuration file, and this allows you to recreate the cluster without losing data, credentials for the system users, etc.

Here's a sequence of steps to follow:

1. List Custom Resources, replacing the `<namespace>` placeholder with your namespace.

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

Sample output

```
NAME       ENDPOINT                         STATUS   POSTGRES   PGBOUNCER   AGE
cluster1   cluster1-pgbouncer.default.svc   ready    3          3           30m
```

2. Delete the Custom Resource with the name of your cluster (for example, let's use the default `cluster1` name).

```
$ kubectl delete pg cluster1 -n <namespace>
```

Sample output

```
perconapgcluster.pgv2.percona.com "cluster1" deleted
```

3. Check that the cluster is deleted by listing the available Custom Resources once again.

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

Sample output

```
No resources found in <namespace> namespace.
```

Delete the Operator

You can uninstall the Operator by deleting the Deployments related to it.

1. List the deployments. Replace the `<namespace>` placeholder with your namespace.

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

Sample output

```
NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
percona-postgresql-operator   1/1     1            1           13m
```

2. Delete the `percona-*` deployment

```
$ kubectl delete deploy percona-postgresql-operator -n <namespace>
```

3. Check that the Operator is deleted by listing the Pods. As a result you should have no Pods related to it.

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

Sample output

```
No resources found in <namespace> namespace.
```

Delete Custom Resource Definition

If you are not just deleting the Operator and PostgreSQL cluster from a specific namespace, but want to clean up your entire Kubernetes environment, you can also delete the CustomRecourceDefinitions (CRDs).

Warning

CRDs in Kubernetes are non-namespaced but are available to the whole environment. This means that you shouldn't delete CRD if you still have the Operator and database cluster in some namespace.

You can delete CRD as follows:

1. List the CRDs:

```
$ kubectl get crd
```

Sample output

```
allowlistedv2workloads.auto.gke.io                     2023-09-07T14:15:30Z
allowlistedworkloads.auto.gke.io                       2023-09-07T14:15:29Z
audits.warden.gke.io                                   2023-09-07T14:15:32Z
backendconfigs.cloud.google.com                        2023-09-07T14:15:41Z
capacityrequests.internal.autoscaling.gke.io           2023-09-07T14:15:25Z
frontendconfigs.networking.gke.io                      2023-09-07T14:15:41Z
managedcertificates.networking.gke.io                  2023-09-07T14:15:41Z
memberships.hub.gke.io                                 2023-09-07T14:15:30Z
perconapgbackups.pgv2.percona.com                      2023-09-07T14:28:59Z
perconapgclusters.pgv2.percona.com                     2023-09-07T14:29:02Z
perconapgrestores.pgv2.percona.com                     2023-09-07T14:29:03Z
postgresclusters.postgres-operator.crunchydata.com     2023-09-07T14:29:06Z
serviceattachments.networking.gke.io                   2023-09-07T14:15:44Z
servicenetworkendpointgroups.networking.gke.io         2023-09-07T14:15:43Z
storagestates.migration.k8s.io                         2023-09-07T14:15:53Z
storageversionmigrations.migration.k8s.io              2023-09-07T14:15:53Z
updateinfos.nodemanagement.gke.io                      2023-09-07T14:15:55Z
volumesnapshotclasses.snapshot.storage.k8s.io          2023-09-07T14:15:52Z
volumesnapshotcontents.snapshot.storage.k8s.io         2023-09-07T14:15:52Z
volumesnapshots.snapshot.storage.k8s.io                2023-09-07T14:15:52Z
```

2. Now delete the `percona*.pgv2.percona.com` CRDs:

```
$ kubectl delete crd perconapgbackups.pgv2.percona.com
perconapgclusters.pgv2.percona.com perconapgrestores.pgv2.percona.com
```

Sample output

```
customresourcedefinition.apiextensions.k8s.io "perconapgbackups.pgv2.percona.com" deleted
customresourcedefinition.apiextensions.k8s.io "perconapgclusters.pgv2.percona.com" deleted
customresourcedefinition.apiextensions.k8s.io "perconapgrestores.pgv2.percona.com" deleted
```

Monitor Kubernetes

Monitoring the state of the database is crucial to timely identify and react to performance issues. Percona Monitoring and Management (PMM) solution enables you to do just that.

However, the database state also depends on the state of the Kubernetes cluster itself. Hence it's important to have metrics that can depict the state of the Kubernetes cluster.

This document describes how to set up monitoring of the Kubernetes cluster health. This setup has been tested with the PMM Server as the centralized data storage and the Victoria Metrics Kubernetes monitoring stack as the metrics collector. These steps may also apply if you use another Prometheus-compatible storage.

Pre-requisites

To set up monitoring of Kubernetes, you need the following:

1. PMM Server up and running. You can run PMM Server as a Docker image, a virtual appliance, or on an AWS instance. Please refer to the official PMM documentation for the installation instructions.
2. Helm v3.
3. kubectl.
4. The PMM Server API key. The key must have the role "Admin".

   Get the PMM API key:

From PMM UI

Generate the PMM API key

From command line

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

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

Note

The API key is not rotated.

Install the Victoria Metrics Kubernetes monitoring stack

Quick install

1. To install the Victoria Metrics Kubernetes monitoring stack with the default parameters, use the quick install command. Replace the following placeholders with your values:

    - `API-KEY` - The [API key of your PMM Server]()
    - `PMM-SERVER-URL` - The URL to access the PMM Server
    - `UNIQUE-K8s-CLUSTER-IDENTIFIER` - Identifier for the Kubernetes cluster. It can be the name you defined during the cluster creation.

    You should use a unique identifier for each Kubernetes cluster. The use of the same identifer for more than one Kubernetes cluster will result in the conflicts during the metrics collection.

    - `NAMESPACE` - The namespace where the Victoria metrics Kubernetes stack will be installed. If you haven't created the namespace before, it will be created during the command execution.

    We recommend to use a separate namespace like `monitoring-system`.

    ```
    $ curl -fsL  https://raw.githubusercontent.com/Percona-Lab/k8s-monitoring/refs/tags/v0.1.1/vm-operator-k8s-stack/quick-install.sh | bash -s -- --api-key <API-KEY> --pmm-server-url <PMM-SERVER-URL> --k8s-cluster-id <UNIQUE-K8s-CLUSTER-IDENTIFIER> --namespace <NAMESPACE>
    ```

    Note

    The Prometheus node exporter is not installed by default since it requires privileged containers with the access to the host file system. If you need the metrics for Nodes, add the `--node-exporter-enabled` flag as follows:

    ```
    $ curl -fsL  https://raw.githubusercontent.com/Percona-Lab/k8s-monitoring/refs/tags/v0.1.1/vm-operator-k8s-stack/quick-install.sh | bash -s -- --api-key <API-KEY> --pmm-server-url <PMM-SERVER-URL> --k8s-cluster-id <UNIQUE-K8s-CLUSTER-IDENTIFIER> --namespace <NAMESPACE> --node-exporter-enabled
    ```

Install manually

You may need to customize the default parameters of the Victoria metrics Kubernetes stack.

- Since we use the PMM Server for monitoring, there is no need to store the data in Victoria Metrics Operator. Therefore, the Victoria Metrics Helm chart is installed with the `vmsingle.enabled` and `vmcluster.enabled` parameters set to `false` in this setup.
- [Check all the role-based access control (RBAC) rules]() of the `victoria-metrics-k8s-stack` chart and the dependencies chart, and modify them based on your requirements.

Configure authentication in PMM

To access the PMM Server resources and perform actions on the server, configure authentication.

1. Encode the PMM Server API key with base64.

   Linux

   ```
   $ echo -n <API-key> | base64 --wrap=0
   ```

   macOS

   ```
   $ echo -n <API-key> | base64
   ```

2. Create the Namespace where you want to set up monitoring. The following command creates the Namespace `monitoring-system`. You can specify a different name. In the latter steps, specify your namespace instead of the `<namespace>` placeholder.

   ```
   $ kubectl create namespace monitoring-system
   ```

3. Create the YAML file for the Kubernetes Secrets and specify the base64-encoded API key value within. Let's name this file `pmm-api-vmoperator.yaml`.

   pmm-api-vmoperator.yaml

   ```
   apiVersion: v1
   data:
     api_key: <base-64-encoded-API-key>
   kind: Secret
   metadata:
     name: pmm-token-vmoperator
     #namespace: default
   type: Opaque
   ```

4. Create the Secrets object using the YAML file you created previously. Replace the `<filename>` placeholder with your value.

   ```
   $ kubectl apply -f pmm-api-vmoperator.yaml -n <namespace>
   ```

5. Check that the secret is created. The following command checks the secret for the resource named `pmm-token-vmoperator` (as defined in the `metadata.name` option in the secrets file). If you defined another resource name, specify your value.

```
$ kubectl get secret pmm-token-vmoperator -n <namespace>
```

Create a ConfigMap to mount for `kube-state-metrics`

The `kube-state-metrics` (KSM) is a simple service that listens to the Kubernetes API server and generates metrics about the state of various objects - Pods, Deployments, Services and Custom Resources.

To define what metrics the `kube-state-metrics` should capture, create the ConfigMap and mount it to a container.

Use the example `configmap.yaml` configuration file to create the ConfigMap.

```
$ kubectl apply -f https://raw.githubusercontent.com/Percona-Lab/k8s-monitoring/
refs/tags/v0.1.1/vm-operator-k8s-stack/ksm-configmap.yaml -n <namespace>
```

As a result, you have the `customresource-config-ksm` ConfigMap created.

Install the Victoria Metrics Kubernetes monitoring stack

1. Add the dependency repositories of victoria-metrics-k8s-stack chart.

   ```
   $ helm repo add grafana https://grafana.github.io/helm-charts
   $ helm repo add prometheus-community https://prometheus-community.github.io/
   helm-charts
   ```

2. Add the Victoria Metrics Kubernetes monitoring stack repository.

   ```
   $ helm repo add vm https://victoriametrics.github.io/helm-charts/
   ```

3. Update the repositories.

   ```
   $ helm repo update
   ```

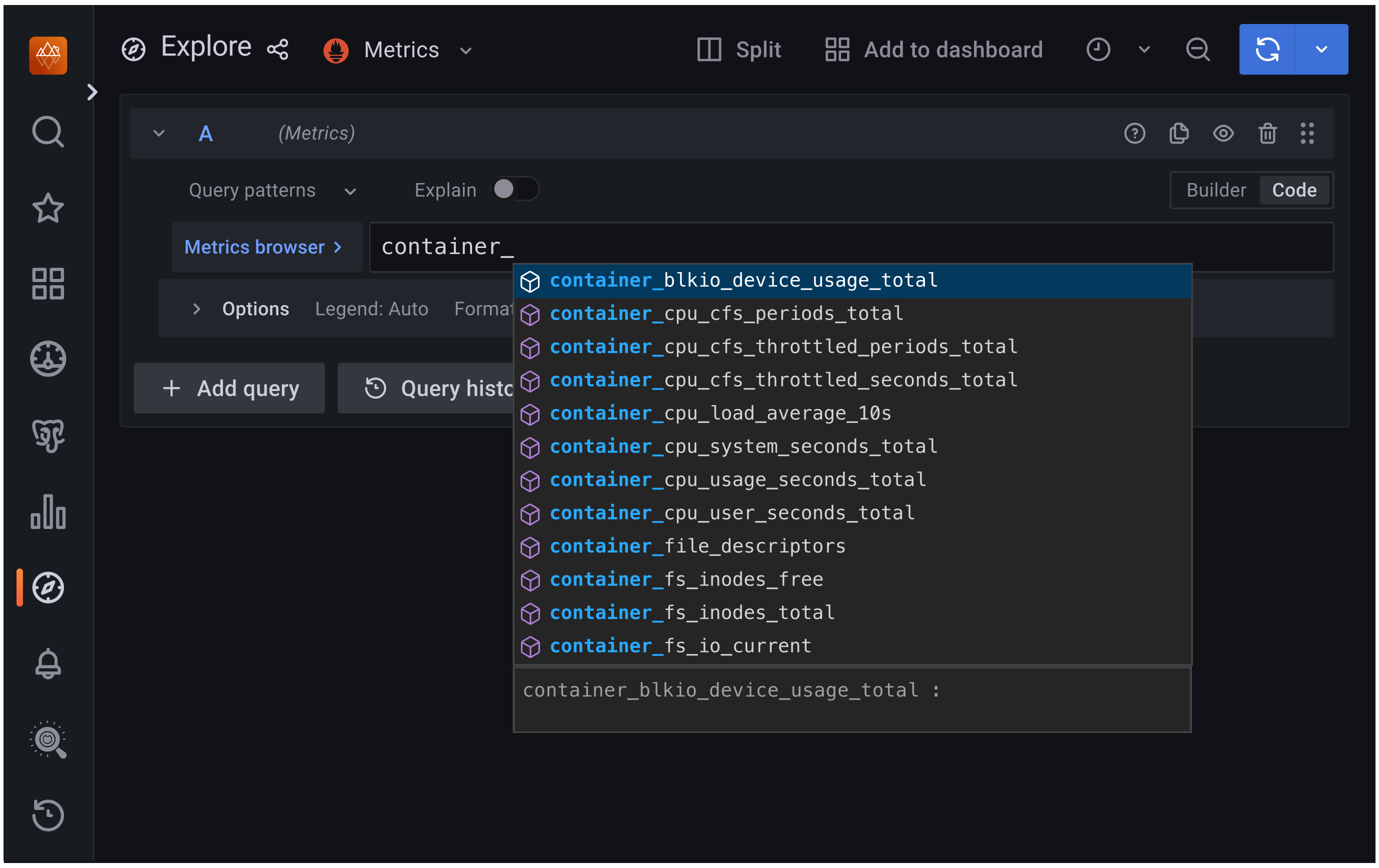
4. Install the Victoria Metrics Kubernetes monitoring stack Helm chart. You need to specify the following configuration:

   - the URL to access the PMM server in the `externalVM.write.url` option in the format `<PMM-SERVER-URL>/victoriametrics/api/v1/write`. The URL can contain either the IP address or the hostname of the PMM server.
   - the unique name or an ID of the Kubernetes cluster in the `vmagent.spec.externalLabels.k8s_cluster_id` option. Ensure to set different values if you are sending metrics from multiple Kubernetes clusters to the same PMM Server.
   - the `<namespace>` placeholder with your value. The Namespace must be the same as the Namespace for the Secret and ConfigMap

```
{.bash data-prompt="$" }
    $ helm install vm-k8s vm/victoria-metrics-k8s-stack \
     -f https://raw.githubusercontent.com/Percona-Lab/k8s-monitoring/refs/tags/
v0.1.1/vm-operator-k8s-stack/values.yaml \
     --set externalVM.write.url=<PMM-SERVER-URL>/victoriametrics/api/v1/write \
     --set vmagent.spec.externalLabels.k8s_cluster_id=<UNIQUE-CLUSTER-IDENTIFER/
NAME> \
     -n <namespace>
```

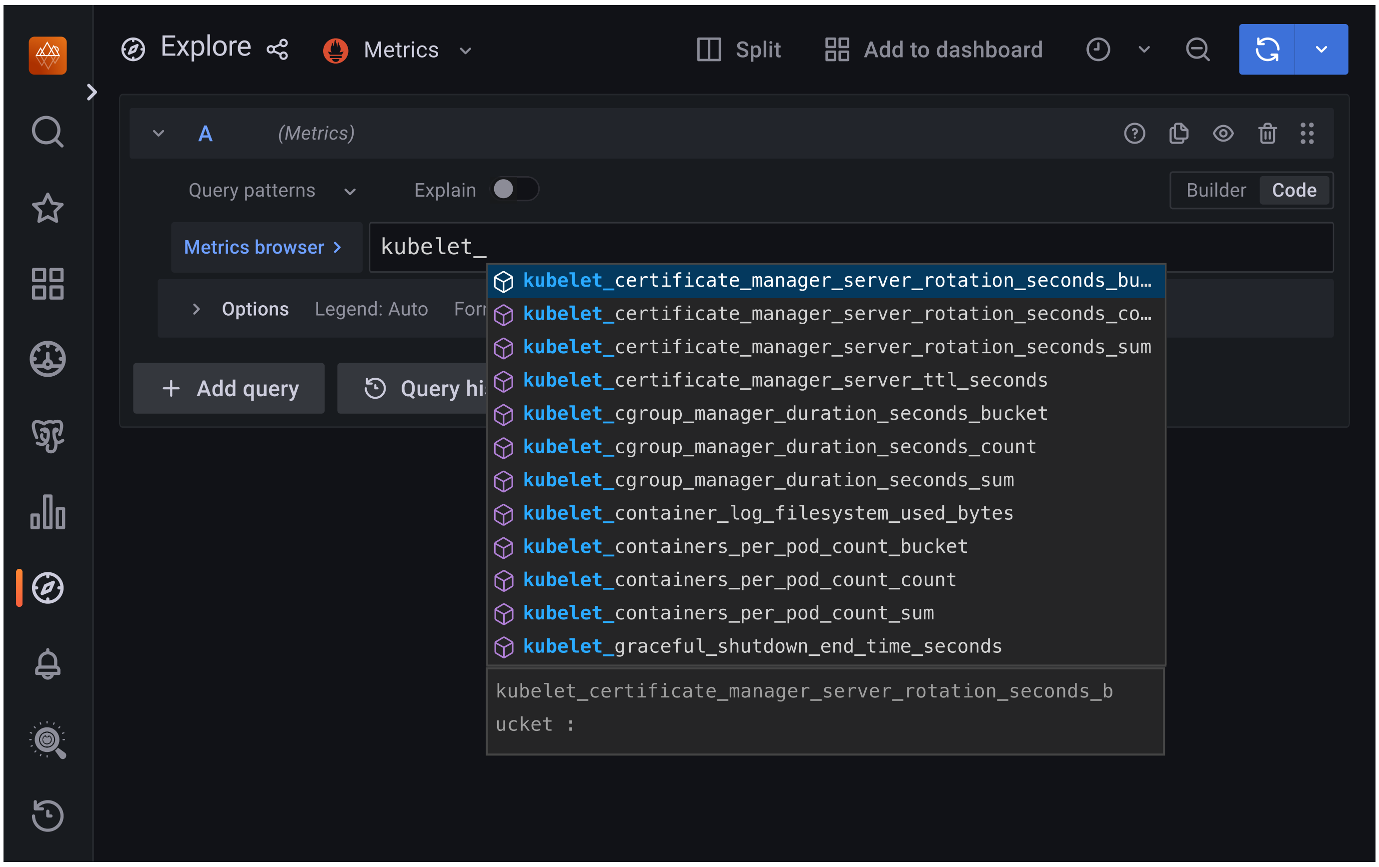
To illustrate, say your PMM Server URL is `https://pmm-example.com`, the cluster ID is `test-cluster` and the Namespace is `monitoring-system`. Then the command would look like this:

```{.bash .no-copy } $ helm install vm-k8s vm/victoria-metrics-k8s-stack \ -f https://raw.githubusercontent.com/Percona-Lab/k8s-monitoring/refs/tags/v0.1.1/vm-operator-k8s-stack/values.yaml \ –set externalVM.write.url=https://pmm-example.com/victoriametrics/api/v1/write \ –set vmagent.spec.externalLabels.k8s_cluster_id=test-cluster \ -n monitoring-system

## Validate the successful installation

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

**Sample output**

```
vm-k8s-stack-kube-state-metrics-d9d85978d-9pzbs                1/1     Running   0
28m
vm-k8s-stack-victoria-metrics-operator-844d558455-gvg4n        1/1     Running   0
28m
vmagent-vm-k8s-stack-victoria-metrics-k8s-stack-55fd8fc4fbcxwhx   2/2     Running   0
28m
```

What Pods are running depends on the configuration chosen in values used while installing `victoria-metrics-k8s-stack` chart.

Verify metrics capture

1. Connect to the PMM server.
2. Click **Explore** and switch to the **Code** mode.
3. Check that the required metrics are captured, type the following in the Metrics browser dropdown:
    - cadvisor:
```



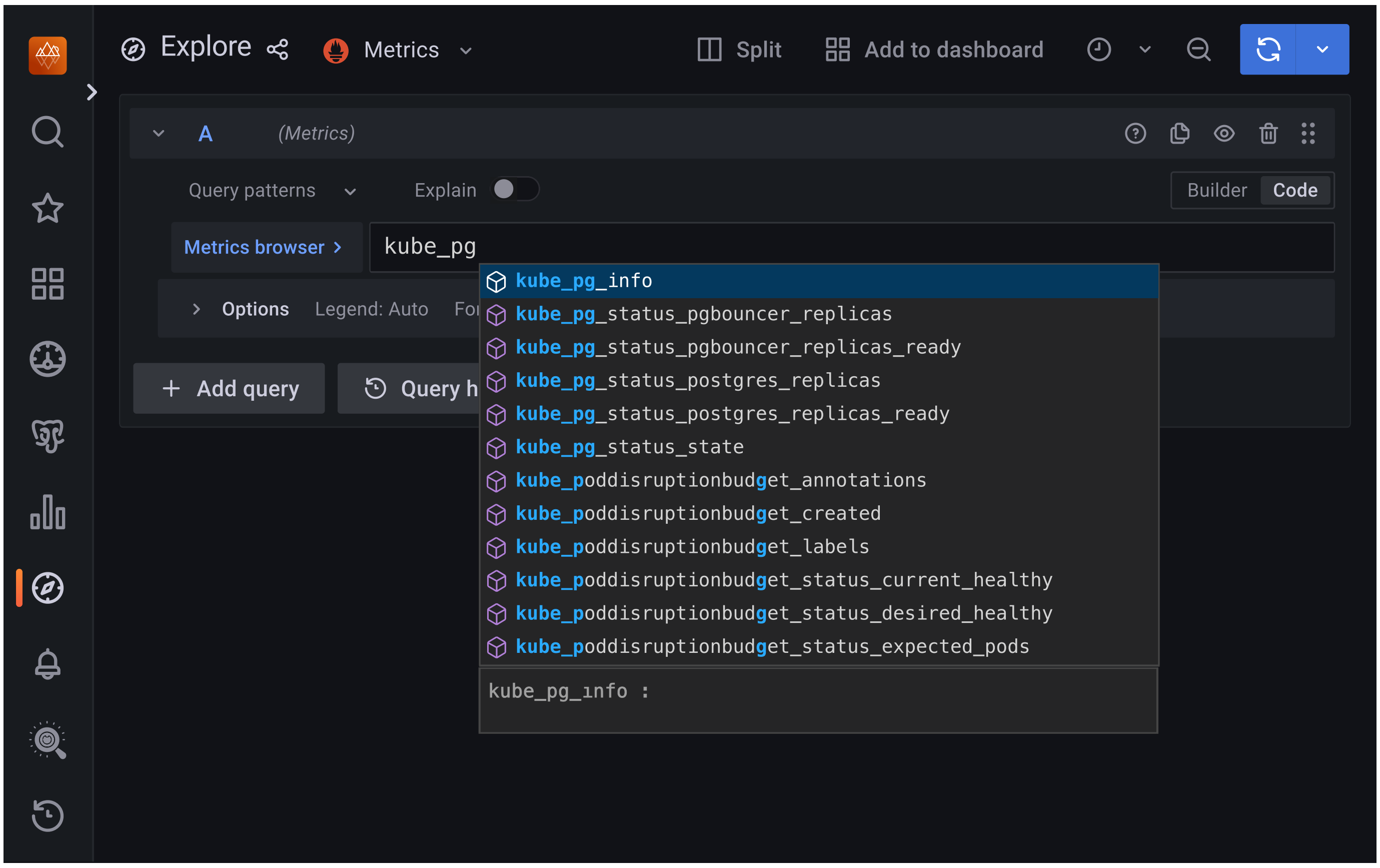
```
container_
container_blkio_device_usage_total
container_cpu_cfs_periods_total
container_cpu_cfs_throttled_periods_total
container_cpu_cfs_throttled_seconds_total
container_cpu_load_average_10s
container_cpu_system_seconds_total
container_cpu_usage_seconds_total
container_cpu_user_seconds_total
container_file_descriptors
container_fs_inodes_free
container_fs_inodes_total
container_fs_io_current
container_blkio_device_usage_total :
```

- kubelet:



```
kubelet_
kubelet_certificate_manager_server_rotation_seconds_bu…
kubelet_certificate_manager_server_rotation_seconds_co…
kubelet_certificate_manager_server_rotation_seconds_sum
kubelet_certificate_manager_server_ttl_seconds
kubelet_cgroup_manager_duration_seconds_bucket
kubelet_cgroup_manager_duration_seconds_count
kubelet_cgroup_manager_duration_seconds_sum
kubelet_container_log_filesystem_used_bytes
kubelet_containers_per_pod_count_bucket
kubelet_containers_per_pod_count_count
kubelet_containers_per_pod_count_sum
kubelet_graceful_shutdown_end_time_seconds
kubelet_certificate_manager_server_rotation_seconds_b
ucket :
```

- kube-state-metrics metrics that also include Custom resource metrics for the Operator and database deployed in your Kubernetes cluster:



```
kube_pg
kube_pg_info
kube_pg_status_pgbouncer_replicas
kube_pg_status_pgbouncer_replicas_ready
kube_pg_status_postgres_replicas
kube_pg_status_postgres_replicas_ready
kube_pg_status_state
kube_poddisruptionbudget_annotations
kube_poddisruptionbudget_created
kube_poddisruptionbudget_labels
kube_poddisruptionbudget_status_current_healthy
kube_poddisruptionbudget_status_desired_healthy
kube_poddisruptionbudget_status_expected_pods
kube_pg_info :
```

Uninstall Victoria metrics Kubernetes stack

To remove Victoria metrics Kubernetes stack used for Kubernetes cluster monitoring, use the cleanup script. By default, the script removes all the Custom Resource Definitions(CRD) and Secrets associated with the Victoria metrics Kubernetes stack. To keep the CRDs, run the script with the `--keep-crd` flag.

Remove CRDs

Replace the `<NAMESPACE>` placeholder with the namespace you specified during the Victoria metrics Kubernetes stack installation:

```
$ bash <(curl -fsL https://raw.githubusercontent.com/Percona-Lab/k8s-monitoring/
refs/tags/v0.1.1/vm-operator-k8s-stack/cleanup.sh) --namespace <NAMESPACE>
```

Keep CRDs

Replace the `<NAMESPACE>` placeholder with the namespace you specified during the Victoria metrics Kubernetes stack installation:

```
$ bash <(curl -fsL https://raw.githubusercontent.com/Percona-Lab/k8s-monitoring/
refs/tags/v0.1.1/vm-operator-k8s-stack/cleanup.sh) --namespace <NAMESPACE> --keep-
crd
```

Check that the Victoria metrics Kubernetes stack is deleted:

```
$ helm list -n <namespace>
```

The output should provide the empty list.

If you face any issues with the removal, uninstall the stack manually:

```
$ helm uninstall vm-k8s-stack -n < namespace>
```

Use PostGIS extension with Percona Distribution for PostgreSQL

PostGIS is a PostgreSQL extension that adds GIS capabilities to this database.

Starting from the Operator version 2.3.0 it became possible to deploy and manage PostGIS-enabled PostgreSQL.

Due to the large size and domain specifics of this extension, Percona provides separate PostgreSQL Distribution images with it.

Deploy the Operator with PostGIS-enabled database cluster

Following steps will allow you to deploy PostgreSQL cluster with these images.

1. Clone the percona-postgresql-operator repository:

   ```
   $ git clone -b v2.5.1 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. The Custom Resource Definition for Percona Distribution for PostgreSQL should be created from the `deploy/crd.yaml` file. Custom Resource Definition extends the standard set of resources which Kubernetes “knows” about with the new items (in our case ones which are the core of the Operator). Apply it as follows:

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

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

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

4. The role-based access control (RBAC) for Percona Distribution for PostgreSQL is configured with the `deploy/rbac.yaml` file. Role-based access is based on defined roles and the available actions which correspond to each role. The role and actions are defined for Kubernetes resources in the yaml file. Further details about users and roles can be found in Kubernetes documentation.

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

Note

Setting RBAC requires your user to have cluster-admin role privileges. For example, those using Google Kubernetes Engine can grant user needed privileges with the following command:

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

5. Start the Operator within Kubernetes:

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

6. After the Operator is started, modify the `deploy/cr.yaml` configuration file with PostGIS-enabled image - use `percona/percona-postgresql-operator:2.5.1-ppg16.8-postgres-gis` instead of `percona/percona-postgresql-operator:2.5.1-ppg16.8-postgres`

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGCluster
metadata:
  name: cluster1
spec:
  ...
  image: percona/percona-postgresql-operator:2.5.1-ppg16.8-postgres-gis
  ...
```

When done, Percona Distribution for PostgreSQL cluster can be created at any time with the following command:

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

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

```
$ kubectl get pg -n postgres-operator
```

Expected output

```
NAME       ENDPOINT                         STATUS   POSTGRES   PGBOUNCER   AGE
cluster1   cluster1-pgbouncer.default.svc   ready    3          3           30m
```

Check PostGIS extension

To use PostGIS extension you should enable it for a specific database.

For example, you can create the new database named `mygisdata` with the `psql` tool as follows:

```
CREATE database mygisdata;
\c mygisdata;
CREATE SCHEMA gis;
```

Next, enable the `postgis` extension. Make sure you are connected to the database you created earlier and run the following command:

```
CREATE EXTENSION postgis;
```

Finally, check that the extension is enabled:

```
SELECT postgis_full_version();
```

The output should resemble the following:

```
postgis_full_version

---------------------------------------------------------------------------------------
----------------------------------------------------------------------------
 POSTGIS="3.3.3" [EXTENSION] PGSQL="140" GEOS="3.10.2-CAPI-1.16.0" PROJ="8.2.1"
LIBXML="2.9.13" LIBJSON="0.15" LIBPROTOBUF="1.3.3" WAGYU="0.5.0 (Internal)"
```

You can find more about using PostGIS in the official Percona Distribution for PostgreSQL documentation, as well as in this blogpost.

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 pgBackRest used to backup PostgreSQL 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           30m
```

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    pgv2.percona.com/v2          true
PerconaPGBackup
perconapgclusters         pg           pgv2.percona.com/v2          true
PerconaPGCluster
perconapgrestores         pg-restore   pgv2.percona.com/v2          true
PerconaPGRestore
```

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.

Troubleshooting

Check Storage-related objects

Storage-related objects worth to check in case of problems are the following ones:

- Persistent Volume Claims (PVC) and Persistent Volumes (PV), which are playing a key role in stateful applications.
- Storage Class, which automates the creation of Persistent Volumes and the underlying storage.

It is important to remember that PVC is namespace-scoped, but PV and Storage Class are cluster-scoped.

Check the PVC

You can check all the PVC with the following command (use different namespace name instead of `postgres-operator`, if needed):

```
$ kubectl get pvc -n postgres-operator
```

Expected output

```
NAME                             STATUS    VOLUME                                     CAPACITY
ACCESS MODES   STORAGECLASS   AGE
cluster1-instance1-4xkv-pgdata   Bound     pvc-2d20abb7-5350-4810-a098-fbdfbffda041   1Gi
RWO            standard       11h
cluster1-instance1-njt9-pgdata   Bound     pvc-f2e9a722-fd30-435b-ade4-9edf20b2104b   1Gi
RWO            standard       11h
cluster1-instance1-qhh6-pgdata   Bound     pvc-7228300b-81de-4a60-a615-8ca935c95139   1Gi
RWO            standard       11h
cluster1-repo1                   Bound     pvc-b2e0bac3-993d-499e-b311-3aa7b9851bc2   1Gi
RWO            standard       11h
```

- **STATUS**: shows the state of the PVC:
  - For normal working of an application, the status should be `Bound`.
  - If the status is not `Bound`, further investigation is required.
- **VOLUME**: is the name of the Persistent Volume with which PVC is Bound to. Obviously, this field will be occupied only when a PVC is Bound.
- **CAPACITY**: it is the size of the volume claimed.
- **STORAGECLASS**: it indicates the Kubernetes storage class used for dynamic provisioning of Volume.
- **ACCESS MODES**: Access mode indicates how Volume is used with the Pods. Access modes should have write permission if the application needs to write data, which is obviously true in case of databases.

Now you can check a specific PVC for more details using its name as follows:

```
$ kubectl get pvc cluster1-instance1-4xkv-pgdata -n postgres-operator -oyaml #
output stripped for brevity, name of PVC may vary
```

Expected output

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  ...
  name: cluster1-instance1-4xkv-pgdata
  namespace: postgres-operator
  ...
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1G
  storageClassName: standard
  volumeMode: Filesystem
  volumeName: pvc-2d20abb7-5350-4810-a098-fbdfbffda041
status:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 24Gi
  phase: Bound
```

Check the PV

It is important to remember that PV is a cluster-scoped Object. If you see any issues with attaching a Volume to a Pod, PV and PVC might be looked upon.

Check all the PV present in the Kubernetes cluster as follows:

```
$ kubectl get pv
```

Expected output

```
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
CLAIM                                              STORAGECLASS   REASON   AGE
pvc-2d20abb7-5350-4810-a098-fbdfbffda041   1Gi        RWO            Delete           Bound
postgres-operator/cluster1-instance1-4xkv-pgdata   standard                11h
pvc-7228300b-81de-4a60-a615-8ca935c95139   1Gi        RWO            Delete           Bound
postgres-operator/cluster1-instance1-qhh6-pgdata   standard                11h
pvc-b2e0bac3-993d-499e-b311-3aa7b9851bc2   1Gi        RWO            Delete           Bound
postgres-operator/cluster1-repo1                   standard                11h
pvc-f2e9a722-fd30-435b-ade4-9edf20b2104b   1Gi        RWO            Delete           Bound
postgres-operator/cluster1-instance1-njt9-pgdata   standard                11h
```

Now you can check a specific PV for more details using its name as follows:

```
$ kubectl get pv pvc-2d20abb7-5350-4810-a098-fbdfbffda041 -oyaml
```

Expected output

```
apiVersion: v1
kind: PersistentVolume
metadata:
  ...
  name: pvc-f3e7097f-accd-4f5d-9c9d-6f29b54a368b
  ...
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 1Gi
  claimRef:
    apiVersion: v1
    kind: PersistentVolumeClaim
    name: cluster1-instance1-4xkv-pgdata
    namespace: postgres-operator
    resourceVersion: "912868"
    uid: f3e7097f-accd-4f5d-9c9d-6f29b54a368b
 gcePersistentDisk:
    fsType: ext4
    pdName: pvc-f3e7097f-accd-4f5d-9c9d-6f29b54a368b
 nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: topology.kubernetes.io/zone
          operator: In
          values:
          - us-central1-a
        - key: topology.kubernetes.io/region
          operator: In
          values:
          - us-central1
  persistentVolumeReclaimPolicy: Delete
  storageClassName: standard
  volumeMode: Filesystem
status:
  phase: Bound
```

Fields to check if there are any issues in binding with PVC, are the `claimRef` and `nodeAffinity`.

The `claimRef` one indicates to which PVC this volume is bound to. This means that if by any chance PVC is deleted (e.g. by the appropriate finalizer), this section needs to be modified so that it can bind to a new PVC.

The `spec.nodeAffinity` field may influence the PV availability as well: for example, it can make Volume accessed in one availability zone only.

Check the StorageClass

StorageClass is also a cluster-scoped object, and it indicates what type of underlying storage is used for the Volumes.

Check all the storage class present in the Kubernetes cluster, and check which storage class is the default one:

```
$ kubectl get sc
```

Expected output

```
NAME                 PROVISIONER             RECLAIMPOLICY   VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION   AGE
premium-rwo          pd.csi.storage.gke.io   Delete          WaitForFirstConsumer   true
44d
standard (default)   kubernetes.io/gce-pd    Delete          Immediate              true
44d
standard-rwo         pd.csi.storage.gke.io   Delete          WaitForFirstConsumer   true
44d
```

If some PVC does not refer any storage class explicitly, it means that the default storage class is used. Ensure there is only one default Storage class.

You can check a specific storage class as follows:

```
$ kubectl get sc standard -oyaml
```

Expected output

```
allowVolumeExpansion: true
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
 annotations:
   storageclass.kubernetes.io/is-default-class: "true"
 creationTimestamp: "2022-10-09T06:28:03Z"
 labels:
   addonmanager.kubernetes.io/mode: EnsureExists
 name: standard
 resourceVersion: "906"
 uid: 933d37db-990b-4b2d-bf3a-dd091d0b00ae
parameters:
 type: pd-standard
provisioner: kubernetes.io/gce-pd
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

Important things to observe here are the following ones:

- Check if the provisioner and parameters are indicating the type of storage you intend to provision.
- Check the volumeBindingMode especially if the storage cannot be accessed across availability zones. “WaitForFirstConsumer” volumeBindingMode ensures volume is provisioned only after a Pod requesting the Volume is created.

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/pg16/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/pg16/log/
  postgresql-Wed.log
  bash-4.4$ exit
  exit
  $
  ```

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
  ```

Increase pgBackRest log verbosity

The pgBackRest tool used for backups supports different log verbosity levels. By default, it logs warnings and errors, but sometimes fixing backup/restore issues can be simpler when you get more debugging information from it.

Log verbosity is controlled by pgBackRest –log-level-stderr option.

You can add it to the `deploy/backup.yaml` file to use it with on-demand backups as follows:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGBackup
metadata:
  name: backup1
spec:
  pgCluster: cluster1
  repoName: repo1
  options:
  - --log-level-stderr=debug
```

Custom Resource options

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

`metadata`

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 the database cluster Pods after the deletion event (off by default).

Toplevel `spec` elements

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

`crVersion`

Version of the Operator the Custom Resource belongs to.

Value type	Example
S string	`2.5.1`

`standby.enabled`

Enables or disables running the cluster in a standby mode (read-only copy of an existing cluster, useful for disaster recovery, etc).

Value type	Example
boolean	`false`

`standby.host`

Host address of the primary cluster this standby cluster connects to.

Value type	Example
S string	`"<primary-ip>"`

`standby.port`

Port number used by a standby copy to connect to the primary cluster.

Value type	Example
S string	`"<primary-port>"`

`openshift`

Set to `true` if the cluster is being deployed on OpenShift, set to `false` otherwise, or unset it for autodetection.

Value type	Example
boolean	`true`

`standby.repoName`

Name of the pgBackRest repository in the primary cluster this standby cluster connects to.

Value type	Example
S string	`repo1`

`secrets.customRootCATLSSecret.name`

Name of the secret with the custom root CA certificate and key for secure connections to the PostgreSQL server, see Transport Layer Security (TLS) for details.

Value type	Example
🅂 string	`cluster1-ca-cert`

`secrets.customRootCATLSSecret.items`

Key-value pairs of the `key` (a key from the `secrets.customRootCATLSSecret.name` secret) and the `path` (name on the file system) for the custom root certificate and key. See Transport Layer Security (TLS) for details.

Value type	Example

≣ subdoc	`- key: “tls.crt”`
	`  path: “root.crt”`
	`- key: “tls.key”`
	`  path: “root.key”`

`secrets.customTLSSecret.name`

A secret with TLS certificate generated for *external* communications, see Transport Layer Security (TLS) for details.

Value type	Example
🅂 string	`cluster1-cert`

`secrets.customReplicationTLSSecret.name`

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

Value type	Example
🅂 string	`replication1-cert`

`users.name`

The name of the PostgreSQL user.

Value type	Example
S string	`rhino`

`users.databases`

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).

Value type	Example
S string	`zoo`

`users.password.type`

The set of characters used for password generation: can be either `ASCII` (default) or `AlphaNumeric`.

Value type	Example
S string	`ASCII`

`users.options`

The `ALTER ROLE` options other than password (the option is ignored for `postgres` user).

Value type	Example
S string	`"SUPERUSER"`

`users.secretName`

The custom name of the user’s Secret; if not specified, the default `<clusterName>-pguser-<userName>` variant will be used.

Value type	Example
S string	`"rhino-credentials"`

`databaseInitSQL.key`

Data key for the [Custom configuration options ConfigMap](#) with the init SQL file, which will be executed at cluster creation time.

Value type	Example
S string	`init.sql`

`databaseInitSQL.name`

Name of the [ConfigMap](#) with the init SQL file, which will be executed at cluster creation time.

Value type	Example
S string	`cluster1-init-sql`

`pause`

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.

Value type	Example
S string	`false`

`unmanaged`

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.

Value type	Example
S string	`false`

`dataSource.postgresCluster.clusterName`

Name of an existing cluster to use as the data source when restoring backup to a new cluster.

Value type	Example
S string	`cluster1`

`dataSource.postgresCluster.clusterNamespace`

Namespace of an existing cluster used as a data source (is needed if the new cluster will be created in a different namespace; needs the Operator deployed in multi-namespace/cluster-wide mode).

Value type	Example
S string	`cluster1-namespace`

`dataSource.postgresCluster.repoName`

Name of the pgBackRest repository in the source cluster that contains the backup to be restored to a new cluster.

Value type	Example
S string	`repo1`

`dataSource.postgresCluster.options`

The pgBackRest command-line options for the pgBackRest restore command.

Value type	Example
S string	

`dataSource.postgresCluster.tolerations.effect`

The Kubernetes Pod tolerations effect for data migration.

Value type	Example
S string	`NoSchedule`

`dataSource.postgresCluster.tolerations.key`

The Kubernetes Pod tolerations key for data migration.

Value type	Example
S string	`role`

`dataSource.postgresCluster.tolerations.operator`

The Kubernetes Pod tolerations operator for data migration.

Value type	Example
S string	`Equal`

`dataSource.postgresCluster.tolerations.value`

The Kubernetes Pod tolerations value for data migration.

Value type	Example
S string	`connection-poolers`

`dataSource.pgbackrest.stanza`

Name of the pgBackRest stanza to use as the data source when restoring backup to a new cluster.

Value type	Example
S string	`db`

`dataSource.pgbackrest.configuration.secret.name`

Name of the Kubernetes Secret object with custom pgBackRest configuration, which will be added to the pgBackRest configuration generated by the Operator.

Value type	Example

Value type	Example
S string	`pgo-s3-creds`

`dataSource.pgbackrest.global`

Settings, which are to be included in the `global` section of the pgBackRest configuration generated by the Operator.

Value type	Example
≡ subdoc	`/pgbackrest/postgres-operator/hippo/repo1`

`dataSource.pgbackrest.repo.name`

Name of the pgBackRest repository.

Value type	Example
S string	`repo1`

`dataSource.pgbackrest.repo.s3.bucket`

The Amazon S3 bucket or Google Cloud Storage bucket name used for backups. Bucket name should follow Amazon naming rules or Google naming rules, and additionally, it can't contain dots.

Value type	Example
S string	`"my-bucket"`

`dataSource.pgbackrest.repo.s3.endpoint`

The endpoint URL of the S3-compatible storage to be used for backups (not needed for the original Amazon S3 cloud).

Value type	Example
S string	`"s3.ca-central-1.amazonaws.com"`

`dataSource.pgbackrest.repo.s3.region`

The AWS region to use for Amazon and all S3-compatible storages.

Value type	Example
boolean	`"ca-central-1"`

`dataSource.pgbackrest.tolerations.effect`

The Kubernetes Pod tolerations effect for pgBackRest at data migration.

Value type	Example
string	`NoSchedule`

`dataSource.pgbackrest.tolerations.key`

The Kubernetes Pod tolerations key for pgBackRest at data migration.

Value type	Example
string	`role`

`dataSource.pgbackrest.tolerations.operator`

The Kubernetes Pod tolerations operator for pgBackRest at data migration.

Value type	Example
string	`Equal`

`dataSource.pgbackrest.tolerations.value`

The Kubernetes Pod tolerations value for pgBackRest at data migration.

Value type	Example

Value type	Example
S string	`connection-poolers`

`dataSource.volumes.pgDataVolume.pvcName`

The PostgreSQL data volume name for the Persistent Volume Claim used for data migration.

Value type	Example
S string	`cluster1`

`dataSource.volumes.pgDataVolume.directory`

The mount point for PostgreSQL data volume used for data migration.

Value type	Example
S string	`cluster1`

`dataSource.volumes.pgDataVolume.tolerations.effect`

The Kubernetes Pod tolerations effect for PostgreSQL data volume used for data migration.

Value type	Example
S string	`NoSchedule`

`dataSource.volumes.pgDataVolume.tolerations.key`

The Kubernetes Pod tolerations key for PostgreSQL data volume used for data migration.

Value type	Example
S string	`role`

`dataSource.volumes.pgDataVolume.tolerations.operator`

The Kubernetes Pod tolerations operator for PostgreSQL data volume used for data migration.

Value type	Example
string	`Equal`

`dataSource.volumes.pgDataVolume.tolerations.value`

The Kubernetes Pod tolerations value for PostgreSQL data volume used for data migration.

Value type	Example
string	`connection-poolers`

`dataSource.volumes.pgDataVolume.annotations`

The Kubernetes annotations metadata for PostgreSQL data volume used for data migration.

Value type	Example
label	`test-annotation: value`

`dataSource.volumes.pgDataVolume.labels`

The Kubernetes labels for PostgreSQL data volume used for data migration.

Value type	Example
label	`test-label: value`

`dataSource.volumes.pgWALVolume.pvcName`

The PostgreSQL write-ahead logs volume name for the Persistent Volume Claim used for data migration.

Value type	Example
string	`cluster1`

`dataSource.volumes.pgWALVolume.directory`

The mount point for PostgreSQL write-ahead logs volume used for data migration.

Value type	Example
S string	`cluster1`

`dataSource.volumes.pgWALVolume.tolerations.effect`

The [Kubernetes Pod tolerations](#) effect for PostgreSQL write-ahead logs volume used for data migration.

Value type	Example
S string	`NoSchedule`

`dataSource.volumes.pgWALVolume.tolerations.key`

The [Kubernetes Pod tolerations](#) key for PostgreSQL write-ahead logs volume used for data migration.

Value type	Example
S string	`role`

`dataSource.volumes.pgWALVolume.tolerations.operator`

The [Kubernetes Pod tolerations](#) operator for PostgreSQL write-ahead logs volume used for data migration.

Value type	Example
S string	`Equal`

`dataSource.volumes.pgWALVolume.tolerations.value`

The [Kubernetes Pod tolerations](#) value for PostgreSQL write-ahead logs volume used for data migration.

Value type	Example

Value type	Example
string	`connection-poolers`

`dataSource.volumes.pgWALVolume.annotations`

The Kubernetes annotations metadata for PostgreSQL write-ahead logs volume used for data migration.

Value type	Example
label	`test-annotation: value`

`dataSource.volumes.pgWALVolume.labels`

The Kubernetes labels for PostgreSQL write-ahead logs volume used for data migration.

Value type	Example
label	`test-label: value`

`dataSource.volumes.pgBackRestVolume.pvcName`

The pgBackRest volume name for the Persistent Volume Claim used for data migration.

Value type	Example
string	`cluster1`

`dataSource.volumes.pgBackRestVolume.directory`

The mount point for pgBackRest volume used for data migration.

Value type	Example
string	`cluster1`

`dataSource.volumes.pgBackRestVolume.tolerations.effect`

The Kubernetes Pod tolerations effect pgBackRest volume used for data migration.

Value type	Example
S string	`NoSchedule`

`dataSource.volumes.pgBackRestVolume.tolerations.key`

The Kubernetes Pod tolerations key for pgBackRest volume used for data migration.

Value type	Example
S string	`role`

`dataSource.volumes.pgBackRestVolume.tolerations.operator`

The Kubernetes Pod tolerations operator for pgBackRest volume used for data migration.

Value type	Example
S string	`Equal`

`dataSource.volumes.pgBackRestVolume.tolerations.value`

The Kubernetes Pod tolerations value for pgBackRest volume used for data migration.

Value type	Example
S string	`connection-poolers`

`dataSource.volumes.pgBackRestVolume.annotations`

The Kubernetes annotations metadata for pgBackRest volume used for data migration.

Value type	Example
label	`test-annotation: value`

`dataSource.volumes.pgBackRestVolume.labels`

The Kubernetes labels for pgBackRest volume used for data migration.

Value type	Example
label	`test-label: value`

`image`

The PostgreSQL Docker image to use.

Value type	Example
string	`perconalab/percona-postgresql-operator:2.5.1-ppg16.8-postgres`

`imagePullPolicy`

This option is used to set the policy for updating PostgreSQL images.

Value type	Example
string	`Always`

`postgresVersion`

The major version of PostgreSQL to use.

Value type	Example
int	`16`

`port`

The port number for PostgreSQL.

Value type	Example

Value type	Example
int	`5432`

`expose.annotations`

The [Kubernetes annotations](#) metadata for PostgreSQL primary.

Value type	Example
label	`my-annotation: value1`

`expose.labels`

Set [labels](#) for the PostgreSQL primary.

Value type	Example
label	`my-label: value2`

`expose.type`

Specifies the type of [Kubernetes Service](#) for PostgreSQL primary.

Value type	Example
string	`LoadBalancer`

`expose.loadBalancerSourceRanges`

The range of client IP addresses from which the load balancer should be reachable (if not set, there is no limitations).

Value type	Example
string	`"10.0.0.0/8"`

`exposeReplicas.annotations`

The [Kubernetes annotations](#) metadata for PostgreSQL replicas.

Value type	Example
label	`my-annotation: value1`

`exposeReplicas.labels`

Set [labels](#) for the PostgreSQL replicas.

Value type	Example
label	`my-label: value2`

`exposeReplicas.type`

Specifies the type of [Kubernetes Service](#) for PostgreSQL replicas.

Value type	Example
string	`LoadBalancer`

`exposeReplicas.loadBalancerSourceRanges`

The range of client IP addresses from which the load balancer should be reachable (if not set, there is no limitations).

Value type	Example
string	`"10.0.0.0/8"`

Instances section

The `instances` section in the [deploy/cr.yaml](#) file contains configuration options for PostgreSQL instances. This section contains at least one *cluster instance* with a number of *PostgreSQL instances* in it (cluster instances are groups of PostgreSQL instances used for fine-grained resources assignment).

`instances.metadata.labels`

Set labels for PostgreSQL Pods.

Value type	Example
label	`pg-cluster-label: cluster1`

`instances.name`

The name of the PostgreSQL instance.

Value type	Example
string	`rs 0`

`instances.replicas`

The number of Replicas to create for the PostgreSQL instance.

Value type	Example
int	`3`

`instances.resources.limits.cpu`

Kubernetes CPU limits for a PostgreSQL instance.

Value type	Example
string	`2.0`

`instances.resources.limits.memory`

The Kubernetes memory limits for a PostgreSQL instance.

Value type	Example

Value type	Example
S string	`4Gi`

`instances.containers.replicaCertCopy.resources.limits.cpu`

Kubernetes CPU limits for `replica-cert-copy` sidecar container.

Value type	Example
S string	`1.0`

`instances.containers.replicaCertCopy.resources.limits.memory`

The Kubernetes memory limits for `replica-cert-copy` sidecar container.

Value type	Example
S string	`1Gi`

`instances.topologySpreadConstraints.maxSkew`

The degree to which Pods may be unevenly distributed under the Kubernetes Pod Topology Spread Constraints.

Value type	Example
1 int	`1`

`instances.topologySpreadConstraints.topologyKey`

The key of node labels for the Kubernetes Pod Topology Spread Constraints.

Value type	Example
S string	`my-node-label`

`instances.topologySpreadConstraints.whenUnsatisfiable`

What to do with a Pod if it doesn’t satisfy the Kubernetes Pod Topology Spread Constraints.

Value type	Example
string	`DoNotSchedule`

`instances.topologySpreadConstraints.labelSelector.matchLabels`

The Label selector for the Kubernetes Pod Topology Spread Constraints.

Value type	Example
label	`postgres-operator.crunchydata.com/instance-set: instance1`

`instances.tolerations.effect`

The Kubernetes Pod tolerations effect for the PostgreSQL instance.

Value type	Example
string	`NoSchedule`

`instances.tolerations.key`

The Kubernetes Pod tolerations key for the PostgreSQL instance.

Value type	Example
string	`role`

`instances.tolerations.operator`

The Kubernetes Pod tolerations operator for the PostgreSQL instance.

Value type	Example

Value type	Example
S string	`Equal`

`instances.tolerations.value`

The [Kubernetes Pod tolerations](#) value for the PostgreSQL instance.

Value type	Example
S string	`connection-poolers`

`instances.priorityClassName`

The [Kuberentes Pod priority class](#) for PostgreSQL instance Pods.

Value type	Example
S string	`high-priority`

‘instances.securityContext’

A custom [Kubernetes Security Context for a Pod](#) to be used instead of the default one.

Value type	Example
≡ subdoc	

Value type	Example

```
fsGroup: 1001
runAsUser: 1001
runAsNonRoot: true
fsGroupChangePolicy: “OnRootMismatch”
runAsGroup: 1001
seLinuxOptions:
  type: spc_t
  level: s0:c123,c456
seccompProfile:
  type: Localhost
  localhostProfile: localhost/profile.json
supplementalGroups:
- 1001
sysctls:
- name: net.ipv4.tcp_keepalive_time
  value: “600”
- name: net.ipv4.tcp_keepalive_intvl
  value: “60”
```

`instances.walVolumeClaimSpec.accessModes`

The Kubernetes PersistentVolumeClaim access modes for the PostgreSQL Write-ahead Log storage.

Value type	Example
S string	`ReadWriteOnce`

`instances.walVolumeClaimSpec.resources.requests.storage`

The Kubernetes storage requests for the storage the PostgreSQL instance will use.

Value type	Example
S string	`1Gi`

`instances.dataVolumeClaimSpec.accessModes`

The Kubernetes PersistentVolumeClaim access modes for the PostgreSQL storage.

Value type	Example

Value type	Example
string	`ReadWriteOnce`

`instances.dataVolumeClaimSpec.storageClassName`

Set the Kubernetes storage class to use with PosgreSQL Cluster PersistentVolumeClaim.

Value type	Example
string	`premium-rwo`

`instances.dataVolumeClaimSpec.resources.requests.storage`

The Kubernetes storage requests for the storage the PostgreSQL instance will use.

Value type	Example
string	`1Gi`

`instances.dataVolumeClaimSpec.resources.limits.storage`

The Kubernetes storage limits for the storage the PostgreSQL instance will use.

Value type	Example
string	`5Gi`

`instances.tablespaceVolumes.name`

Name for the custom tablespace volume.

Value type	Example
string	`user`

`instances.tablespaceVolumes.dataVolumeClaimSpec.accessModes`

The Kubernetes PersistentVolumeClaim access modes for the tablespace volume.

Value type	Example
S string	`ReadWriteOnce`

`instances.tablespaceVolumes.dataVolumeClaimSpec.resources.requests.storage`

The Kubernetes storage requests for the tablespace volume.

Value type	Example
S string	`1Gi`

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.

`instances.sidecars.image`

Image for the custom sidecar container for PostgreSQL Pods.

Value type	Example
S string	`mycontainer1:latest`

`instances.sidecars.name`

Name of the custom sidecar container for PostgreSQL Pods.

Value type	Example
S string	`testcontainer`

`instances.sidecars.imagePullPolicy`

This option is used to set the policy for the PostgreSQL Pod sidecar container.

Value type	Example
string	`Always`

`instances.sidecars.env`

The environment variables set as key-value pairs for the custom sidecar container for PostgreSQL Pods.

Value type	Example
subdoc	

`instances.sidecars.envFrom`

The environment variables set as key-value pairs in ConfigMaps for the custom sidecar container for PostgreSQL Pods.

Value type	Example
subdoc	

`instances.sidecars.command`

Command for the custom sidecar container for PostgreSQL Pods.

Value type	Example
array	`["/bin/sh"]`

`instances.sidecars.args`

Command arguments for the custom sidecar container for PostgreSQL Pods.

Value type	Example
array	`["-c", "while true; do trap 'exit 0' SIGINT SIGTERM SIGQUIT SIGKILL; done;"]`

Backup section

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

`backups.trackLatestRestorableTime`

Enables or disables tracking the latest restorable time for latest successful backup (on by default). It can be turned off to reduced S3 API usage.

Value type	Example
boolean	`true`

`backups.pgbackrest.metadata.labels`

Set labels for pgBackRest Pods.

Value type	Example
label	`pg-cluster-label: cluster1`

`backups.pgbackrest.image`

The Docker image for pgBackRest.

Value type	Example
string	`perconalab/percona-postgresql-operator:2.5.1-ppg16.8-pgbackrest`

`backups.pgbackrest.containers.pgbackrest.resources.limits.cpu`

Kubernetes CPU limits for a pgBackRest container.

Value type	Example
string	`1.0`

`backups.pgbackrest.containers.pgbackrest.resources.limits.memory`

The Kubernetes memory limits for a pgBackRest container.

Value type	Example
S string	`1Gi`

`backups.pgbackrest.containers.pgbackrestConfig.resources.limits.cpu`

Kubernetes CPU limits for `pgbackrest-config` sidecar container.

Value type	Example
S string	`1.0`

`backups.pgbackrest.containers.pgbackrestConfig.resources.limits.memory`

The Kubernetes memory limits for `pgbackrest-config` sidecar container.

Value type	Example
S string	`1Gi`

`backups.pgbackrest.configuration.secret.name`

Name of the Kubernetes Secret object with custom pgBackRest configuration, which will be added to the pgBackRest configuration generated by the Operator.

Value type	Example
S string	`cluster1-pgbackrest-secrets`

`backups.pgbackrest.jobs.priorityClassName`

The Kuberentes Pod priority class for pgBackRest jobs.

Value type	Example

Value type	Example
S string	`high-priority`

`backups.pgbackrest.jobs.resources.limits.cpu`

[Kubernetes CPU limits](#) for a pgBackRest job.

Value type	Example
1 int	`200`

`backups.pgbackrest.jobs.resources.limits.memory`

The [Kubernetes memory limits](#) for a pgBackRest job.

Value type	Example
S string	`128Mi`

`backups.pgbackrest.jobs.tolerations.effect`

The [Kubernetes Pod tolerations](#) effect for a backup job.

Value type	Example
S string	`NoSchedule`

`backups.pgbackrest.jobs.tolerations.key`

The [Kubernetes Pod tolerations](#) key for a backup job.

Value type	Example
S string	`role`

`backups.pgbackrest.jobs.tolerations.operator`

The Kubernetes Pod tolerations operator for a backup job.

Value type	Example
string	`Equal`

`backups.pgbackrest.jobs.tolerations.value`

The Kubernetes Pod tolerations value for a backup job.

Value type	Example
string	`connection-poolers`

`backups.pgbackrest.jobs.securityContext`

A custom Kubernetes Security Context for a Pod to be used instead of the default one.

Value type	Example

subdoc	
	```
	fsGroup: 1001
	runAsUser: 1001
	runAsNonRoot: true
	fsGroupChangePolicy: “OnRootMismatch”
	runAsGroup: 1001
	seLinuxOptions:
	  type: spc_t
	  level: s0:c123,c456
	seccompProfile:
	  type: Localhost
	  localhostProfile: localhost/profile.json
	supplementalGroups:
	- 1001
	sysctls:
	- name: net.ipv4.tcp_keepalive_time
	  value: “600”
	- name: net.ipv4.tcp_keepalive_intvl
	  value: “60”
	```

`backups.pgbackrest.global`

Settings, which are to be included in the `global` section of the pgBackRest configuration generated by the Operator.

Value type	Example
≡ subdoc	`repo1-path: /pgbackrest/postgres-operator/cluster1/repo1`

`backups.pgbackrest.repoHost.priorityClassName`

The Kuberentes Pod priority class for pgBackRest repo.

Value type	Example
S string	`high-priority`

`backups.pgbackrest.repoHost.topologySpreadConstraints.maxSkew`

The degree to which Pods may be unevenly distributed under the Kubernetes Pod Topology Spread Constraints.

Value type	Example
1 int	`1`

`backups.pgbackrest.repoHost.topologySpreadConstraints.topologyKey`

The key of node labels for the Kubernetes Pod Topology Spread Constraints.

Value type	Example
S string	`my-node-label`

`backups.pgbackrest.repoHost.topologySpreadConstraints.whenUnsatisfiable`

What to do with a Pod if it doesn’t satisfy the Kubernetes Pod Topology Spread Constraints.

Value type	Example
S string	`ScheduleAnyway`

`backups.pgbackrest.repoHost.topologySpreadConstraints.labelSelector.matchLabels`

The Label selector for the Kubernetes Pod Topology Spread Constraints.

Value type	Example
label	`postgres-operator.crunchydata.com/pgbackrest: ""`

`backups.pgbackrest.repoHost.affinity.podAntiAffinity`

Pod anti-affinity, allows setting the standard Kubernetes affinity constraints of any complexity.

Value type	Example
subdoc	

`backups.pgbackrest.repoHost.tolerations.effect`

The Kubernetes Pod tolerations effect for pgBackRest repo.

Value type	Example
string	`NoSchedule`

`backups.pgbackrest.repoHost.tolerations.key`

The Kubernetes Pod tolerations key for pgBackRest repo.

Value type	Example
string	`role`

`backups.pgbackrest.repoHost.tolerations.operator`

The Kubernetes Pod tolerations operator for pgBackRest repo.

Value type	Example

Value type	Example
string	`Equal`

`backups.pgbackrest.repoHost.tolerations.value`

The Kubernetes Pod tolerations value for pgBackRest repo.

Value type	Example
string	`connection-poolers`

‘backups.pgbackrest.repoHost.securityContext’

A custom Kubernetes Security Context for a Pod to be used instead of the default one.

<table><tr><th>Value type</th><th>Example</th></tr><tr><td>subdoc</td><td><pre>fsGroup: 1001
runAsUser: 1001
runAsNonRoot: true
fsGroupChangePolicy: “OnRootMismatch”
runAsGroup: 1001
seLinuxOptions:
  type: spc_t
  level: s0:c123,c456
seccompProfile:
  type: Localhost
  localhostProfile: localhost/profile.json
supplementalGroups:
- 1001
sysctls:
- name: net.ipv4.tcp_keepalive_time
  value: “600”
- name: net.ipv4.tcp_keepalive_intvl
  value: “60”</pre></td></tr></table>

`backups.pgbackrest.manual.repoName`

Name of the pgBackRest repository for on-demand backups.

Value type	Example

Value type	Example
S string	`repo1`

`backups.pgbackrest.manual.options`

The on-demand backup command-line options which will be passed to pgBackRest for on-demand backups.

Value type	Example
S string	`--type=full`

`backups.pgbackrest.repos.name`

Name of the pgBackRest repository for backups.

Value type	Example
S string	`repo1`

`backups.pgbackrest.repos.schedules.full`

Scheduled time to make a full backup specified in the crontab format.

Value type	Example
S string	`0 0 \* \* 6`

`backups.pgbackrest.repos.schedules.differential`

Scheduled time to make a differential backup specified in the crontab format.

Value type	Example
S string	`0 0 \* \* 6`

`backups.pgbackrest.repos.volume.volumeClaimSpec.accessModes`

The Kubernetes PersistentVolumeClaim access modes for the pgBackRest Storage.

Value type	Example
S string	`ReadWriteOnce`

`backups.pgbackrest.repos.volume.volumeClaimSpec.storageClassName`

Set the Kubernetes Storage Class to use with the Percona Operator for PosgreSQL backups stored on Persistent Volume.

Value type	Example
S string	`premium-rwo`

`backups.pgbackrest.repos.volume.volumeClaimSpec.resources.requests.storage`

The Kubernetes storage requests for the pgBackRest storage.

Value type	Example
S string	`1Gi`

`backups.pgbackrest.repos.s3.bucket`

The Amazon S3 bucket name used for backups

Value type	Example
S string	`"my-bucket"`
.	

`backups.pgbackrest.repos.s3.endpoint`

The endpoint URL of the S3-compatible storage to be used for backups (not needed for the original Amazon S3 cloud).

Value type	Example
S string	`"s3.ca-central-1.amazonaws.com"`

`backups.pgbackrest.repos.s3.region`

The AWS region to use for Amazon and all S3-compatible storages.

Value type	Example
S string	`"ca-central-1"`

`backups.pgbackrest.repos.gcs.bucket`

The Google Cloud Storage bucket name used for backups.

Value type	Example
S string	`"my-bucket"`

`backups.pgbackrest.repos.azure.container`

Name of the Azure Blob Storage container for backups.

Value type	Example
S string	`my-container`

`backups.restore.tolerations.effect`

The Kubernetes Pod tolerations effect for the backup restore job.

Value type	Example
S string	`NoSchedule`

`backups.restore.tolerations.key`

The Kubernetes Pod tolerations key for the backup restore job.

Value type	Example
string	`role`

`backups.restore.tolerations.operator`

The Kubernetes Pod tolerations operator for the backup restore job.

Value type	Example
string	`Equal`

`backups.restore.tolerations.value`

The Kubernetes Pod tolerations value for the backup restore job.

Value type	Example
string	`connection-poolers`

PMM section

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

`pmm.enabled`

Enables or disables monitoring Percona Distribution for PostgreSQL cluster with PMM.

Value type	Example
boolean	`false`

`pmm.image`

Percona Monitoring and Management (PMM) Client Docker image.

Value type	Example
S string	`percona/pmm-client:2.44.0`

`pmm.imagePullPolicy`

This option is used to set the policy for updating PMM Client images.

Value type	Example
S string	`IfNotPresent`

`pmm.pmmSecret`

Name of the Kubernetes Secret object for the PMM Server password.

Value type	Example
S string	`cluster1-pmm-secret`

`pmm.serverHost`

Address of the PMM Server to collect data from the cluster.

Value type	Example
S string	`monitoring-service`

`pmm.querySource`

Query source to track PostgreSQL statistics. Either pg_stat_monitor (`pgstatmonitor`, the default value) or pg_stat_statements (`pgstatstatements`) can be used.

Value type	Example
S string	`pgstatmonitor`

Proxy section

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

`proxy.pgBouncer.metadata.labels`

Set labels for pgBouncer Pods.

Value type	Example
label	`pg-cluster-label: cluster1`

`proxy.pgBouncer.replicas`

The number of the pgBouncer Pods to provide connection pooling.

Value type	Example
int	`3`

`proxy.pgBouncer.image`

Docker image for the pgBouncer connection pooler.

Value type	Example
string	`perconalab/percona-postgresql-operator:2.5.1-ppg16.8-pgbouncer`

`proxy.pgBouncer.exposeSuperusers`

Enables or disables exposing superuser user through pgBouncer.

Value type	Example
boolean	`false`

`proxy.pgBouncer.resources.limits.cpu`

Kubernetes CPU limits for a pgBouncer container.

Value type	Example
S string	`200m`

`proxy.pgBouncer.resources.limits.memory`

The Kubernetes memory limits for a pgBouncer container.

Value type	Example
S string	`128Mi`

`proxy.pgBouncer.containers.pgbouncerConfig.resources.limits.cpu`

Kubernetes CPU limits for `pgbouncer-config` sidecar container.

Value type	Example
S string	`1.0`

`proxy.pgBouncer.containers.pgbouncerConfig.resources.limits.memory`

The Kubernetes memory limits for `pgbouncer-config` sidecar container.

Value type	Example
S string	`1Gi`

`proxy.pgBouncer.expose.type`

Specifies the type of Kubernetes Service for pgBouncer.

Value type	Example
S string	`ClusterIP`

`proxy.pgBouncer.expose.annotations`

The Kubernetes annotations metadata for pgBouncer.

Value type	Example
label	`pg-cluster-annot: cluster1`

`proxy.pgBouncer.expose.labels`

Set labels for the pgBouncer Service.

Value type	Example
label	`pg-cluster-label: cluster1`

`proxy.pgBouncer.expose.loadBalancerSourceRanges`

The range of client IP addresses from which the load balancer should be reachable (if not set, there is no limitations).

Value type	Example
string	`"10.0.0.0/8"`

`proxy.pgBouncer.affinity.podAntiAffinity`

Pod anti-affinity, allows setting the standard Kubernetes affinity constraints of any complexity.

Value type	Example
subdoc	

‘proxy.pgBouncer.securityContext’

A custom Kubernetes Security Context for a Pod to be used instead of the default one.

Value type	Example

Value type	Example

≣ subdoc	`fsGroup: 1001`
	`runAsUser: 1001`
	`runAsNonRoot: true`
	`fsGroupChangePolicy: “OnRootMismatch”`
	`runAsGroup: 1001`
	`seLinuxOptions:`
	`  type: spc_t`
	`  level: s0:c123,c456`
	`seccompProfile:`
	`  type: Localhost`
	`  localhostProfile: localhost/profile.json`
	`supplementalGroups:`
	`- 1001`
	`sysctls:`
	`- name: net.ipv4.tcp_keepalive_time`
	`  value: “600”`
	`- name: net.ipv4.tcp_keepalive_intvl`
	`  value: “60”`

`proxy.pgBouncer.config`

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.

Value type	Example

≣ subdoc	`global:`
	`pool_mode: transaction`

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.

`proxy.pgBouncer.sidecars.image`

Image for the custom sidecar container for pgBouncer Pods.

Value type	Example

Value type	Example
string	`mycontainer1:latest`

`proxy.pgBouncer.sidecars.name`

Name of the custom sidecar container for pgBouncer Pods.

Value type	Example
string	`testcontainer`

`proxy.pgBouncer.sidecars.imagePullPolicy`

This option is used to set the policy for the pgBouncer Pod sidecar container.

Value type	Example
string	`Always`

`proxy.pgBouncer.sidecars.env`

The environment variables set as key-value pairs for the custom sidecar container for pgBouncer Pods.

Value type	Example
subdoc	

`proxy.pgBouncer.sidecars.envFrom`

The environment variables set as key-value pairs in ConfigMaps for the custom sidecar container for pgBouncer Pods.

Value type	Example
subdoc	

`proxy.pgBouncer.sidecars.command`

Command for the custom sidecar container for pgBouncer Pods.

Value type	Example
array	`["/bin/sh"]`

`proxy.pgBouncer.sidecars.args`

Command arguments for the custom sidecar container for pgBouncer Pods.

Value type	Example
array	`["-c", "while true; do trap 'exit 0' SIGINT SIGTERM SIGQUIT SIGKILL; done;"]`

Patroni Section

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

Value type	Example
int	`3`

`patroni.syncPeriodSeconds`

How often to perform liveness/readiness probes for the patroni container (in seconds).

Value type	Example
int	`3`

`patroni.leaderLeaseDurationSeconds`

Initial delay for liveness/readiness probes for the patroni container (in seconds).

`patroni.dynamicConfiguration`

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.

Value type	Example

≡ subdoc	
	```
	postgresql:
	  parameters:
	    max_parallel_workers: 2
	    max_worker_processes: 2
	    shared_buffers: 1GB
	    work_mem: 2MB
	```

`patroni.switchover.enabled`

Enables or disables manual change of the cluster primary instance.

Value type	Example
S string	`true`

`patroni.switchover.targetInstance`

The name of the Pod that should be set as the new primary. When not specified, the new primary will be selected randomly.

Value type	Example
S string	

Custom extensions Section

The `extensions` section in the deploy/cr.yaml file contains configuration options to manage PostgreSQL extensions.

`extensions.image`

Image for the custom PostgreSQL extension loader sidecar container.

Value type	Example

Value type	Example
S string	`percona/percona-postgresql-operator:2.5.1`

`extensions.imagePullPolicy`

Policy for the custom extension sidecar container.

Value type	Example
S string	`Always`

`extensions.storage.type`

The cloud storage type used for backups. Only `s3` type is currently supported.

Value type	Example
S string	`s3`

`extensions.storage.bucket`

The Amazon S3 bucket name for prepackaged PostgreSQL custom extensions.

Value type	Example
S string	`pg-extensions`

`extensions.storage.region`

The AWS region to use.

Value type	Example
S string	`eu-central-1`

`extensions.storage.endpoint`

The S3 endpoint to use.

Value type	Example
string	`s3.eu-central-1.amazonaws.com`

`extensions.storage.secret.name`

The Kubernetes secret for the custom extensions storage. It should contain `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` keys.

Value type	Example
string	`cluster1-extensions-secret`

`extensions.builtin`

The key-value pairs which enable or disable Percona Distribution for PostgreSQL builtin extensions.

Value type	Example

subdoc	`pg_stat_monitor: true`
	`pg_audit: true`

`extensions.custom.name`

Name of the PostgreSQL custom extension.

Value type	Example
string	`pg_cron`

`extensions.custom.version`

Version of the PostgreSQL custom extension.

Value type	Example

Value type	Example
S string	`1.6.1`

Reference

Percona certified images

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

Image	Digest
Images added for the Operator version 2.5.1:	
percona/percona-postgresql-operator:2.5.1 (x86_64)	e2697ebaae5c98100d86f0521f1b660d933d0df339ecd16f2384b141b5d2bdfa
percona/percona-postgresql-operator:2.5.1 (ARM64)	1203d46708f867653739c100c4f55fec310ee1e58fefaa8e73fe8b2baf90eea3
percona/percona-postgresql-operator:2.5.1-ppg16.8-postgres	7dc40246ace22dbc5e84b27d756ac9d412ad5ebba2f99889644c427801a34a18
percona/percona-postgresql-operator:2.5.1-ppg16.8-postgres-gis3.3.8	346d47f722a0a9bf11623331d5d09cad90d3aec586e7e1d10bc3d3cc9ef99cca
percona/percona-postgresql-operator:2.5.1-ppg16.8-pgbouncer1.24.0	7167e6c595e1bffdcea5df1c3ab5693adff5a1f647e3e5a72c3400ad2c7aa323
percona/percona-postgresql-operator:2.5.1-ppg16.8-pgbackrest2.54.2	9e973e9a93ae9ea2babb836c43c193f19abc406ca8ddb7a9e3c6d61b2a16a47b
percona/pmm-client:2.44.0	19a07dfa8c12a0554308cd11d7d38494ea02a14cfac6c051ce8ff254b7d0a4a7
Images released with the Operator version 2.5.0:	
percona/percona-postgresql-operator:2.5.0 (x86_64)	bebe17db0f2a33a23424b67c5d41654c16e546b6545a8cb4c2b9af1f4d73bd7b
percona/percona-postgresql-operator:2.5.0 (ARM64)	d22b19c5ab0cd7267bb013f98b40465bc8669a533aeb571a98899c233b92e9e4
percona/percona-postgresql-operator:2.5.0-ppg12.20-postgres	f053767e38a5889ed3b5cbb07ccb1c05d81f63faaa4b648d61373b6d57e6fe0c

Image	Digest
percona/percona-postgresql-operator:2.5.0-ppg13.16-postgres	8e1b90876038fdc2301f20996aa7d35eb299c85597d7025489aa3ca47ec6f9d4
percona/percona-postgresql-operator:2.5.0-ppg14.13-postgres	69d6c44b89399fc89384125392228c3992267a930802ec2c4e57f9ce03a0719f
percona/percona-postgresql-operator:2.5.0-ppg15.8-postgres	595a5f83d83d5a8e811aa9ab275ad932898766e2b3287b517b84d366608617f7
percona/percona-postgresql-operator:2.5.0-ppg16.4-postgres	d5758f6172f7715c7644eaecf7f9d079b6516bff0a98f3fd9cd5a29af90d6698
percona/percona-postgresql-operator:2.5.0-ppg12.20-postgres-gis3.3.6	592c5166f3d758829ab465cf9b98c009b325cbf9cf0cfa47b7f8a88df5dd3211
percona/percona-postgresql-operator:2.5.0-ppg13.16-postgres-gis3.3.6	316c9f18c24971153cef9cd613a7de9c67e23836a8e16eef2957341776cffd98
percona/percona-postgresql-operator:2.5.0-ppg14.13-postgres-gis3.3.6	2c4b646738b56a6af4c0b60110a44ebad924828441fe5aa8014bec39e5b4b70e
percona/percona-postgresql-operator:2.5.0-ppg15.8-postgres-gis3.3.6	64962a4157d1537b851aed0992e0c74a17514643d244a8dd5609681407f6c211
percona/percona-postgresql-operator:2.5.0-ppg16.4-postgres-gis3.3.6	5ad2e95ac178f21a540d75d3d857007d96ef2846d558cf584fbdaa0a704051ad
percona/percona-postgresql-operator:2.5.0-ppg12.20-pgbouncer1.23.1	d0300a4c17745d9f964b7ceb887272203e62730df7782c2b401ee10f42e35cbe
percona/percona-postgresql-operator:2.5.0-ppg13.16-pgbouncer1.23.1	e1692b9b9ec02ee232449e5da6df464708673b637c66b109db4c21836d5cae8f
percona/percona-postgresql-operator:2.5.0-ppg14.13-pgbouncer1.23.1	1652e89ad49ab50242b683b5107d70d28b6f1b1cadc3566411dc58902c146aa2

Image	Digest
percona/percona-postgresql-operator:2.5.0-ppg15.8-pgbouncer1.23.1	c048dbd2cdf86c759d2a5e44028455c4a7b7f951892e390b41491769034b99b7
percona/percona-postgresql-operator:2.5.0-ppg16.4-pgbouncer1.23.1	b360a78b992f3741f63003234e3832bf073cbb8f29e3ebae5e453fa0e3ca84df
percona/percona-postgresql-operator:2.5.0-ppg12.20-pgbackrest2.53-1	57a3b15f482adba4c7d3689080f41578a42cede1939bbe0817de957f9049c93e
percona/percona-postgresql-operator:2.5.0-ppg13.16-pgbackrest2.53-1	c3240d08ec86d3dcf7a30e500b23ada7c766b6785a5360f1d28fb1fb02d0a940
percona/percona-postgresql-operator:2.5.0-ppg14.13-pgbackrest2.53-1	28416ccb433600c4948ebc7b033d4f5348d92b2211fef513ec279a7a55ac7969
percona/percona-postgresql-operator:2.5.0-ppg15.8-pgbackrest2.53-1	6db1b9ef0a305524a1ec8658437919daf89c0c37d96273d32c3d438cf880fade
percona/percona-postgresql-operator:2.5.0-ppg16.4-pgbackrest2.53-1	d636aa73c6f74075d8c2b9bbf51892db94e8d698e66d23ce2b1f612df5a0ddde
percona/pmm-client:2.43.1	aad0a51ec5dadc80b1821964045a67c367dc0e75c17885961b5a4937f409490c

Versions compatibility

Versions of the cluster components and platforms tested with different Operator releases are shown below. Other version combinations may also work but have not been tested.

Cluster components:

Operator	PostgreSQL	pgBackRest	pgBouncer
2.5.1	16	2.54.2	1.24.0
2.5.0	12 - 16	2.53-1	1.23.1
2.4.1	12 - 16	2.51	1.22.1
2.4.0	12 - 16	2.51	1.22.1
2.3.1	12 - 16	2.48	1.18.0
2.3.0	12 - 16	2.48	1.18.0
2.2.0	12 - 15	2.43	1.18.0
2.1.0	12 - 15	2.43	1.18.0
2.0.0	12 - 14	2.41	1.17.0
1.6.0	12 - 14	2.50	1.22.0
1.5.1	12 - 14	2.47	1.20.0
1.5.0	12 - 14	2.47	1.20.0
1.4.0	12 - 14	2.43	1.18.0
1.3.0	12 - 14	2.38	1.17.0
1.2.0	12 - 14	2.37	1.16.1
1.1.0	12 - 14	2.34	1.16.0 for PostgreSQL 12, 1.16.1 for other versions
1.0.0	12 - 13	2.33	1.13.0

Platforms:

Operator	GKE	EKS	Openshift	Azure Kubernetes Service (AKS)	Minikube
2.5.1	1.28 - 1.30	1.28 - 1.30	4.13.46 - 4.16.7	1.28 - 1.30	1.33.1
2.5.0	1.28 - 1.30	1.28 - 1.30	4.13.46 - 4.16.7	1.28 - 1.30	1.33.1
2.4.1	1.27 - 1.29	1.27 - 1.30	4.12.59 - 4.15.18	-	1.33.1
2.4.0	1.27 - 1.29	1.27 - 1.30	4.12.59 - 4.15.18	-	1.33.1
2.3.1	1.24 - 1.28	1.24 - 1.28	4.11.55 - 4.14.6	-	1.32
2.3.0	1.24 - 1.28	1.24 - 1.28	4.11.55 - 4.14.6	-	1.32
2.2.0	1.23 - 1.26	1.23 - 1.27	-	-	1.30.1
2.1.0	1.23 - 1.25	1.23 - 1.25	-	-	-
2.0.0	1.22 - 1.25	-	-	-	-
1.6.0	1.26 - 1.29	1.26 - 1.29	4.12.57 - 4.15.13	-	1.33
1.5.1	1.24 - 1.28	1.24 - 1.28	4.11 - 4.14	-	1.32
1.5.0	1.24 - 1.28	1.24 - 1.28	4.11 - 4.14	-	1.32
1.4.0	1.22 - 1.25	1.22 - 1.25	4.10 - 4.12	-	1.28
1.3.0	1.21 - 1.24	1.20 - 1.22	4.7 - 4.10	-	-
1.2.0	1.19 - 1.22	1.19 - 1.21	4.7 - 4.10	-	-
1.1.0	1.19 - 1.22	1.18 - 1.21	4.7 - 4.9	-	-

Operator	GKE	EKS	Openshift	Azure Kubernetes Service (AKS)	Minikube
1.0.0	1.17 - 1.21	1.21	4.6 - 4.8	-	-

Copyright and licensing information

Documentation licensing

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.

Percona Operator for PostgreSQL Release Notes

Release Notes

Percona Operator for PostgreSQL 2.5.1

- **Date**

  March 03, 2025

- **Installation**

  Installing Percona Operator for PostgreSQL

Release highlights

This release fixes the CVE-2025-1094, vulnerability in the libpq PostgreSQL client library, which made images used by the Operator vulnerable to SQL injection within the PostgreSQL interactive terminal due to the lack of neutralizing quoting. For now, the fix includes the image of PostgreSQL 16.8 and other database cluster images based on PostgreSQL 16.8. Fixed images for other PostgreSQL versions are to follow in the upcoming days.

Supported platforms

The Operator was developed and tested with PostgreSQL version 16.8. Other options may also work but have not been tested. The Operator 2.5.1 provides connection pooling based on pgBouncer 1.24.0 and high-availability implementation based on Patroni 3.3.2.

The following platforms were tested and are officially supported by the Operator 2.5.1:

- Google Kubernetes Engine (GKE) 1.28 - 1.30
- Amazon Elastic Container Service for Kubernetes (EKS) 1.28 - 1.30
- OpenShift 4.13.46 - 4.16.7
- Azure Kubernetes Service (AKS) 1.28 - 1.30
- Minikube 1.34.0 with Kubernetes 1.31.0

This list only includes the platforms that the Percona Operators are specifically tested on as part of the release process. Other Kubernetes flavors and versions depend on the backward compatibility offered by Kubernetes itself.

Percona Operator for PostgreSQL 2.5.0

- **Date**

  October 08, 2024

- **Installation**

  Installing Percona Operator for PostgreSQL

Release Highlights

Automated storage scaling

Starting from this release, the Operator is able to detect if the storage usage on the PVC reaches a certain threshold, and trigger the PVC resize. Such autoscaling needs the upstream auto-growable disk feature turned on when deploying the Operator. This is done via the `PGO_FEATURE_GATES` environment variable set in the `deploy/operator.yaml` manifest (or in the appropriate part of `deploy/bundle.yaml`):

```
- name: PGO_FEATURE_GATES
  value: "AutoGrowVolumes=true"
```

When the support for auto-growable disks is turned on, the `spec.instances[].dataVolumeClaimSpec.resources.limits.storage` Custom Resource option sets the maximum value available for the Operator to scale up.

See official documentation for more details and limitations of the feature.

Major versions upgrade improvements

Major version upgrade, introduced in the Operator version 2.4.0 as a tech preview, had undergone some improvements. Now it is possible to upgrade from one PostgreSQL major version to another with custom images for the database cluster components (PostgreSQL, pgBouncer, and pgBackRest). The upgrade is still triggered by applying the YAML manifest with the information about the existing and desired major versions, which now includes image names. The resulting manifest may look as follows:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGUpgrade
metadata:
  name: cluster1-15-to-16
spec:
  postgresClusterName: cluster1
  image: percona/percona-postgresql-operator:2.4.1-upgrade
  fromPostgresVersion: 15
  toPostgresVersion: 16
  toPostgresImage: percona/percona-postgresql-operator:2.5.0-ppg16.4-postgres
  toPgBouncerImage: percona/percona-postgresql-operator:2.5.0-ppg16.4-
pgbouncer1.23.1
  toPgBackRestImage: percona/percona-postgresql-operator:2.5.0-ppg16.4-
pgbackrest2.53-1
```

Azure Kubernetes Service and Azure Blob Storage support

Azure Kubernetes Service (AKS) is now officially supported platform, so developers and vendors of the solutions based on the Azure platform can take advantage of the official support from Percona or just use officially certified Percona Operator for PostgreSQL images; also, Azure Blob Storage can now be used for backups.

New features

- K8SPG-227 and K8SPG-157: Add support for the Azure Kubernetes Service (AKS) platform and allow using Azure Blob Storage for backups
- K8SPG-244: Automated storage scaling is now supported

Improvements

- K8SPG-630: A new `backups.trackLatestRestorableTime` Custom Resource option allows to disable latest restorable time tracking for users who need reducing S3 API calls usage
- K8SPG-605 and K8SPG-593: Documentation now includes information about upgrading the Operator via Helm and using databaseInitSQL commands
- K8SPG-598: Database major version upgrade now supports custom images
- K8SPG-560: A `pg-restore` Custom Resource is now automatically created at bootstrapping a new cluster from an existing backup
- K8SPG-555: The Operator now creates separate Secret with CA certificate for each cluster
- K8SPG-553: Users can provide the Operator with their own root CA certificate
- K8SPG-454: Cluster status obtained with `kubectl get pg` command is now “ready” not only when all

Pods are ready, but also takes into account if all StatefulSets are up to date

- K8SPG-577: A new `pmm.querySource` Custom Resource option allows to set PMM query source

Bugs Fixed

- K8SPG-629: Fix a bug where the Operator was not deleting backup Pods when cleaning outdated backups according to the retention policy
- K8SPG-499: Fix a bug where cluster was getting stuck in the init state if pgBackRest secret didn't exist
- K8SPG-588: Fix a bug where the Operator didn't stop WAL watcher if the namespace and/or cluster were deleted
- K8SPG-644: Fix a bug in the `pg-db` Helm chart which prevented from setting more than one Toleration

Deprecation, Change, Rename and Removal

With the Operator versions prior to 2.5.0, autogenerated TLS certificates for all database clusters were based on the same generated root CA. Starting from 2.5.0, the Operator creates root CA on a per-cluster basis.

Supported platforms

The Operator was developed and tested with PostgreSQL versions 12.20, 13.16, 14.13, 15.8, and 16.4. Other options may also work but have not been tested. The Operator 2.5.0 provides connection pooling based on pgBouncer 1.23.1 and high-availability implementation based on Patroni 3.3.2.

The following platforms were tested and are officially supported by the Operator 2.5.0:

- Google Kubernetes Engine (GKE) 1.28 - 1.30
- Amazon Elastic Container Service for Kubernetes (EKS) 1.28 - 1.30
- OpenShift 4.13.46 - 4.16.7
- Azure Kubernetes Service (AKS) 1.28 - 1.30
- Minikube 1.34.0 with Kubernetes 1.31.0

This list only includes the platforms that the Percona Operators are specifically tested on as part of the release process. Other Kubernetes flavors and versions depend on the backward compatibility offered by Kubernetes itself.

Percona Operator for PostgreSQL 2.4.1

- **Date**

  August 6, 2024

- **Installation**

  Installing Percona Operator for PostgreSQL

Bugs Fixed

- K8SPG-616: Fix a bug where it was not possible to create a new cluster after deleting the previous one with the `kubectl delete pg` command

Supported platforms

The Operator was developed and tested with PostgreSQL versions 12.19, 13.15, 14.12, 15.7, and 16.3. Other options may also work but have not been tested. The Operator 2.4.1 provides connection pooling based on pgBouncer 1.22.1 and high-availability implementation based on Patroni 3.3.0.

The following platforms were tested and are officially supported by the Operator 2.4.1:

- Google Kubernetes Engine (GKE) 1.27 - 1.29
- Amazon Elastic Container Service for Kubernetes (EKS) 1.27 - 1.30
- OpenShift 4.12.59 - 4.15.18
- Minikube 1.33.1

This list only includes the platforms that the Percona Operators are specifically tested on as part of the release process. Other Kubernetes flavors and versions depend on the backward compatibility offered by Kubernetes itself.

Percona Operator for PostgreSQL 2.4.0

- **Date**

  June 26, 2024

- **Installation**

  Installing Percona Operator for PostgreSQL

Release Highlights

Major versions upgrade (tech preview)

Starting from this release Operator users can automatically upgrade from one PostgreSQL major version to another. Upgrade is triggered by applying the yaml file with the information about the existing and desired major versions, with an example present in `deploy/upgrade.yaml`:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGUpgrade
metadata:
  name: cluster1-15-to-16
spec:
  postgresClusterName: cluster1
  image: perconalab/percona-postgresql-operator:main-upgrade
  fromPostgresVersion: 15
  toPostgresVersion: 16
```

After applying it as usual, by running `kubectl apply -f deploy/upgrade.yaml` command, the actual upgrade takes place as follows:

1. The cluster is paused for a while,
2. The cluster is specially annotated with `pgv2.percona.com/allow-upgrade: <PerconaPGUpgrade.Name>` annotation,
3. Jobs are created to migrate the data,
4. The cluster starts up after the upgrade finishes.

Check official documentation for more details, including ones about tracking the upgrade process and side effects for users with custom extensions.

Supporting PostgreSQL tablespaces

Tablespaces allow DBAs to store a database on multiple file systems within the same server and to control where (on which file systems) specific parts of the database are stored. You can think about it as if you were giving names to your disk mounts and then using those names as additional parameters when creating database objects.

PostgreSQL supports this feature, allowing you to store data outside of the primary data directory. Tablespaces support was present in Percona Operator for PostgreSQL 1.x, and starting from this version, Percona Operator for PostgreSQL 2.x can also bring this feature to your Kubernetes environment, when needed.

Using cloud roles to authenticate on the object storage for backups

Percona Operator for PostgreSQL has introduced a new feature that allows users to authenticate to AWS S3 buckets via IAM roles. Now Operator This enhancement significantly improves security by eliminating the need to manage S3 access keys directly, while also streamlining the configuration process for easier backup and restore operations.

To use this feature, add annotation to the `spec` part of the Custom Resource and also add pgBackRest custom configuration option to the `backups` subsection:

```
spec:
  crVersion: 2.4.0
  metadata:
    annotations:
      eks.amazonaws.com/role-arn: arn:aws:iam::1191:role/role-pgbackrest-access-s3-
bucket
  ...
  backups:
    pgbackrest:
      image: percona/percona-postgresql-operator:2.4.0-ppg16-pgbackrest
      global:
        repo1-s3-key-type: web-id
        ...
```

New features

- K8SPG-138: Users are now able to use AWS IAM role to provide access to the S3 bucket used for backups
- K8SPG-254: Now the Operator automates upgrading PostgreSQL major versions
- K8SPG-459: PostgreSQL tablespaces are now supported by the Operator
- K8SPG-479 and K8SPG-492: It is now possible to specify tolerations for the backup restore jobs as well as

for the data move jobs created when the Operator 1.x is upgraded to 2.x; this is useful in environments with dedicated Kubernetes worker nodes protected by taints

- K8SPG-503 and K8SPG-513: It is now possible to specify resources for the sidecar containers of database instance Pods

Improvements

- K8SPG-259: Users can now change the default level for log messages for pgBackRest to simplify fixing backup and restore issues
- K8SPG-542: Documentation now includes HowTo on creating a disaster recovery cluster using streaming replication
- K8SPG-506: The `pg-backup` objects now have a new `backupName` status field, which allows users to obtain the backup name for restore simpler
- K8SPG-514: The new `securityContext` Custom Resource subsections allow to configure securityContext for PostgreSQL instances, pgBouncer, and pgBackRest Pods
- K8SPG-518: The `kubectl get pg-backup` command now shows the latest restorable time to make it easier to pick a point-in-time recovery target
- K8SPG-519: The new `extensions.storage.endpoint` Custom Resource option allows specifying a custom S3 object storage endpoint for installing custom extensions
- K8SPG-549: It is now possible to expose replica nodes through a separate Service, useful if you want to balance the load and separate reads and writes traffic
- K8SPG-550: The default size for `/tmp` mount point in PMM container was increased from 1.5G to 2G
- K8SPG-585: The namespace field was added to the Operator and database Helm chart templates

Bugs Fixed

- K8SPG-462: Fixed a bug where backups could not start if a previous backup had the same name
- K8SPG-470: Liveness and Readiness probes timeouts are now configurable through Custom Resource
- K8SPG-559: Fix a bug where the first full backup was incorrectly marked as incremental in the status field
- K8SPG-490: Fixed broken replication that occurred after the network loss of the primary Pod with PostgreSQL 14 and older versions
- K8SPG-502: Fix a bug where backup jobs were not cleaned up after completion
- K8SPG-510: Fix a bug where pausing the cluster immediately set its state to “paused” instead of “stopping” while Pods were still running
- K8SPG-531: Fix a bug where scheduled backups did not work for a second database with the same name

in cluster-wide mode

- K8SPG-535: Fix a bug where the Operator crashed when attempting to run a backup with a non-existent repository
- K8SPG-540: Fix a bug in the pg-db Helm chart readme where the key to set the backup secret was incorrectly specified (Thanks to Abhay Tiwari for contribution)
- K8SPG-543: Fix a bug where applying a cr.yaml file with an empty `spec.proxy` field caused the Operator to crash
- K8SPG-547: Fix dependency issue that made pgbackrest-repo container incompatible with pgBackRest 2.50, resulting in the older 2.48 version being used instead

Deprecation and removal

- The `plpythonu` extension was removed from the list of built-in PostgreSQL extensions; users who still need it can enable it for their databases via custom extensions functionality

Supported platforms

The Operator was developed and tested with PostgreSQL versions 12.19, 13.15, 14.12, 15.7, and 16.3. Other options may also work but have not been tested. The Operator 2.4.0 provides connection pooling based on pgBouncer 1.22.1 and high-availability implementation based on Patroni 3.3.0.

The following platforms were tested and are officially supported by the Operator 2.4.0:

- Google Kubernetes Engine (GKE) 1.27 - 1.29
- Amazon Elastic Container Service for Kubernetes (EKS) 1.27 - 1.30
- OpenShift 4.12.59 - 4.15.18
- Minikube 1.33.1

This list only includes the platforms that the Percona Operators are specifically tested on as part of the release process. Other Kubernetes flavors and versions depend on the backward compatibility offered by Kubernetes itself.

Percona Operator for PostgreSQL 2.3.1

- **Date**

  January 23, 2024
- **Installation**

  Installing Percona Operator for PostgreSQL

Release Highlights

This release provides a number of bug fixes, including fixes for the following vulnerabilities in PostgreSQL, pgBackRest, and pgBouncer images used by the Operator:

- OpenSSH could cause remote code execution by ssh-agent if a user establishes an SSH connection to a compromised or malicious SSH server and has agent forwarding enabled (CVE-2023-38408). This vulnerability affects pgBackRest and PostgreSQL images.
- The c-ares library could cause a Denial of Service with 0-byte UDP payload (CVE-2023-32067). This vulnerability affects pgBouncer image.

Both Operator 1.x (including version 1.5.0) and Operator 2.x (including version 2.3.0) are affected. The 2.x versions upgrade to 2.3.1 is recommended to resolve these issues.

Bugs Fixed

- K8SPG-493: Fix a regression due to which the Operator could run scheduled backup only one time
- K8SPG-494: Fix vulnerabilities in PostgreSQL, pgBackRest, and pgBouncer images
- K8SPG-496: Fix the bug where setting the `pause` Custom Resource option to `true` for the cluster with a backup running would not take effect even after the backup completed

Supported platforms

The Operator was developed and tested with PostgreSQL versions 12.17, 13.13, 14.10, 15.5, and 16.1. Other options may also work but have not been tested. The Operator 2.3.1 provides connection pooling based on pgBouncer 1.21.0 and high-availability implementation based on Patroni 3.1.0.

The following platforms were tested and are officially supported by the Operator 2.3.1:

- Google Kubernetes Engine (GKE) 1.24 - 1.28
- Amazon Elastic Container Service for Kubernetes (EKS) 1.24 - 1.28

- OpenShift 4.11.55 - 4.14.6
- Minikube 1.32

This list only includes the platforms that the Percona Operators are specifically tested on as part of the release process. Other Kubernetes flavors and versions depend on the backward compatibility offered by Kubernetes itself.

Percona Operator for PostgreSQL 2.3.0

- **Date**

  December 21, 2023

- **Installation**

  Installing Percona Operator for PostgreSQL

Release Highlights

PostGIS support

Modern businesses heavily rely on location-based data to gain valuable insights and make data-driven decisions. However, integrating geospatial functionality into the existing database systems has often posed a challenge for enterprises. PostGIS, an open-source software extension for PostgreSQL, addresses this difficulty by equipping users with extensive geospatial operations for handling geographic data efficiently. Percona Operator now supports PostGIS, available through a separate container image. You can read more about PostGIS and how to use it with the Operator in our documentation.

OpenShift and PostgreSQL 16 support

The Operator is now compatible with the OpenShift platform empowering enterprise customers with seamless on-premise or cloud deployments on the platform of their choice. Also, PostgreSQL 16 was added to the range of supported database versions and is used by default starting with this release.

Experimental support for custom PostgreSQL extensions

One of great features of PostgreSQL is support for Extensions, which allow adding new functionality to the database on a plugin basis. Starting from this release, users can add custom PostgreSQL extensions dynamically, without the need to rebuild the container image (see this HowTo on how to create and connect yours).

New features

- K8SPG-311 and K8SPG-389: A new `loadBalancerSourceRanges` Custom Resource option allows to customize the range of IP addresses from which the load balancer should be reachable
- K8SPG-375: Experimental support for custom PostgreSQL extensions was added to the Operator
- K8SPG-391: The Operator is now compatible with the OpenShift platform

- K8SPG-434: The Operator now supports Percona Distribution for PostgreSQL version 16 and uses it as default database version

Improvements

- K8SPG-413: The Operator documentation now includes a comptibility matrix for each Operator version, specifying exact versions of all core components as well as supported versions of the database and platforms
- K8SPG-332: Creating backups and pausing the cluster do not interfere with each other: the Operator either postpones the pausing until the active backup ends, or postpones the scheduled backup on the paused cluster
- K8SPG-370: Logging management is now aligned with other Percona Operators, allowing to use structured logging and to control log level
- K8SPG-372: The multi-namespace (cluster-wide) mode of the Operator was improved, making it possible to customize the list of Kubernetes namespaces under the Operator's control
- K8SPG-400: The documentation now explains how to allow application users to connect to a database cluster without TLS (for example, for testing or demonstration purposes)
- K8SPG-410: Scheduled backups now create `pg-backup` object to simplify backup management and tracking
- K8SPG-416: PostgreSQL custom configuration is now supported in the Helm chart
- K8SPG-422 and K8SPG-447: The user can now see backup type and status in the output of `kubectl get pg-backup` and `kubectl get pg-restore` commands
- K8SPG-458: Affinity configuration examples were added to the `default/cr.yaml` configuration file

Bugs Fixed

- K8SPG-435: Fix a bug with insufficient size of /tmp filesystem which caused PostgreSQL Pods to be recreated every few days due to running out of free space on it
- K8SPG-453: Bug in `pg_stat_monitor` PostgreSQL extensions could hang PostgreSQL
- K8SPG-279: Fix regression which made the Operator to crash after creating a backup if there was no backups.pgbackrest.manual section in the Custom Resource
- K8SPG-310: Documentation didn't explain how to apply pgBackRest `verifyTLS` option which can be used to explicitly enable or disable TLS verification for it
- K8SPG-432: Fix a bug due to which backup jobs and Pods were not deleted on deleting the backup object
- K8SPG-442: The Operator didn't allow to append custom items to the PostgreSQL `shared_preload_libraries` option

- K8SPG-443: Fix a bug due to which only English locale was installed in the PostgreSQL image, missing other languages support
- K8SPG-450: Fix a bug which prevented PostgreSQL to initialize the database on Kubernetes working nodes with enabled huge memory pages if Pod resource limits didn't allow using them
- K8SPG-401: Fix a bug which caused Operator crash if deployed with no `pmm` section in the `deploy/cr.yaml` configuration file

Supported platforms

The Operator was developed and tested with PostgreSQL versions 12.17, 13.13, 14.10, 15.5, and 16.1. Other options may also work but have not been tested. The Operator 2.3.0 provides connection pooling based on pgBouncer 1.21.0 and high-availability implementation based on Patroni 3.1.0.

The following platforms were tested and are officially supported by the Operator 2.3.0:

- Google Kubernetes Engine (GKE) 1.24 - 1.28
- Amazon Elastic Container Service for Kubernetes (EKS) 1.24 - 1.28
- OpenShift 4.11.55 - 4.14.6
- Minikube 1.32

This list only includes the platforms that the Percona Operators are specifically tested on as part of the release process. Other Kubernetes flavors and versions depend on the backward compatibility offered by Kubernetes itself.

Percona Operator for 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.

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

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.

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.

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.

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`

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

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

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

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.

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.

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.

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

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.

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.