

Percona Operator for PostgreSQL documentation

2.1.0 (May 04, 2023)

Percona Technical Documentation Team

Percona LLC and/or its affiliates, © 2009 - 2023

Table of contents

1. Percona Operator for PostgreSQL	3
2. Features	3
3. Quickstart	3
4. Installation	3
5. Configuration	3
6. Management	3
7. Reference	4
8. Features	6
8.1 System Requirements	6
8.2 Design overview	7
9. Quickstart	10
9.1 Install Percona Distribution for PostgreSQL using Helm	10
9.2 Install Percona Distribution for PostgreSQL using kubectl	12
10. Installation	17
10.1 Install Percona Distribution for PostgreSQL on Google Kubernetes Engine (GKE)	17
10.2 Install Percona Distribution for PostgreSQL on Kubernetes	22
11. Configuration	24
11.1 Users	24
11.2 Exposing cluster	26
11.3 Binding Percona Distribution for PostgreSQL components to Specific Kubernetes/OpenShift Nodes	28
11.4 Transport Layer Security (TLS)	30
11.5 Telemetry	32
12. Management	33
12.1 Providing Backups	33
12.2 High availability and scaling	46
12.3 Monitoring	49
12.4 Using sidecar containers	51
12.5 Pause/resume PostgreSQL Cluster	53
13. Reference	54
13.1 Custom Resource options	54
13.2 Percona certified images	74
14. Release Notes	76
14.1 Percona Operator for PostgreSQL Release Notes	76
14.2 Percona Operator for PostgreSQL 2.1.0 (Tech preview)	77
14.3 Percona Operator for PostgreSQL 2.0.0 (Tech preview)	79

1. Percona Operator for PostgreSQL

 **Note**

This is version 2.0.0 of the Percona Operator for PostgreSQL. It 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.

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

This automation adds a level of complexity to the container-based architecture and stateful applications, such as a database. A Kubernetes Operator is a special type of controller introduced to simplify complex deployments. The Operator extends the Kubernetes API with custom resources.

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

2. Features

- [System Requirements](#)
- [Design and architecture](#)

3. Quickstart

- [Install with Helm](#)
- [Install with kubectl](#)

4. Installation

- [Install on Google Kubernetes Engine \(GKE\)](#)
- [Generic Kubernetes installation](#)

5. Configuration

- [Application and system users](#)
- [Exposing the cluster](#)
- [Anti-affinity and tolerations](#)
- [Telemetry](#)

6. Management

- [Backup and restore](#)
- [High availability and scaling](#)

- [Monitor with Percona Monitoring and Management \(PMM\)](#)
- [Add sidecar containers](#)
- [Restart or pause the cluster](#)

7. Reference

- [Custom Resource options](#)
- [Percona certified images](#)
- [Release Notes](#)

Contact Us

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#) , contact [Percona Sales](#).

Last update: 2023-05-04

8. Features

8.1 System Requirements

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

8.1.1 Officially 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

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

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

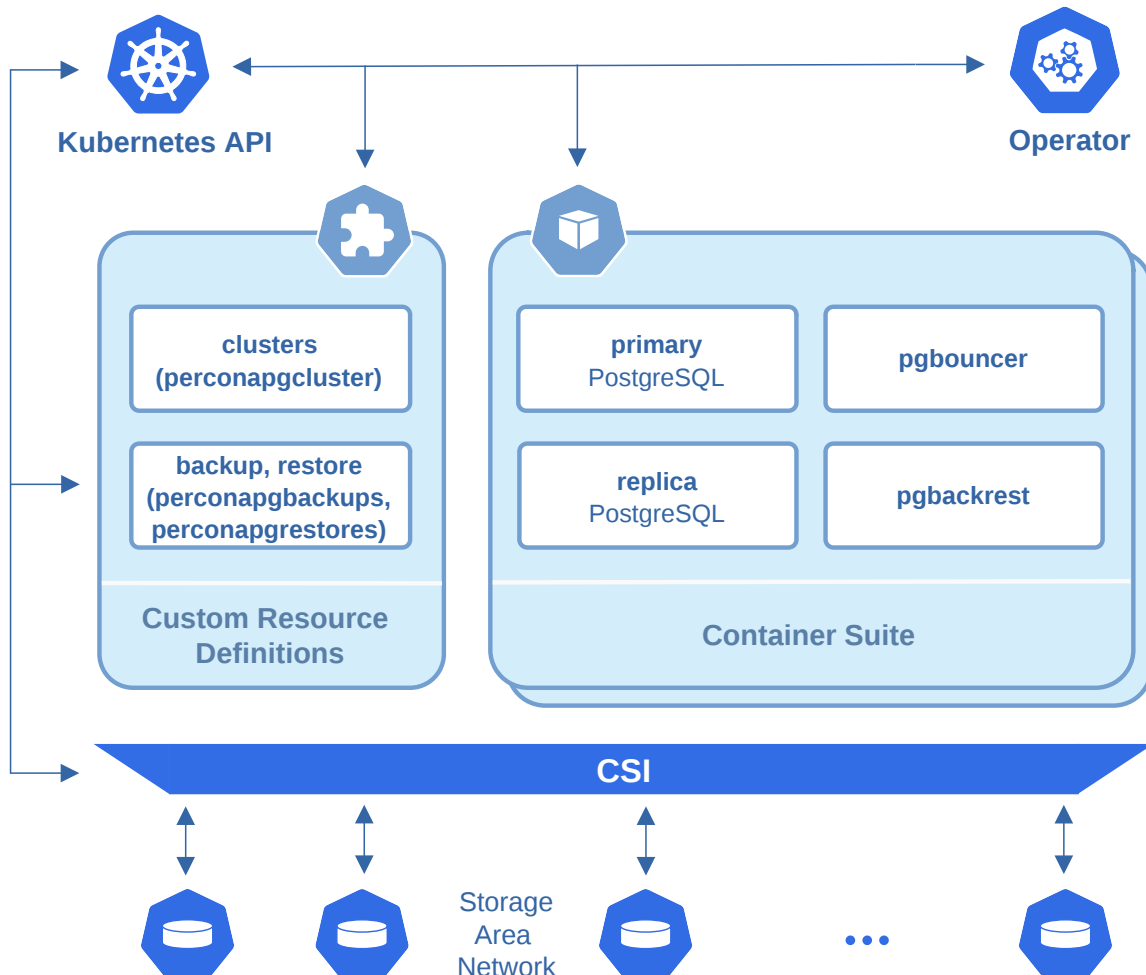
To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#), contact [Percona Sales](#).

Last update: 2023-05-04

8.2 Design overview

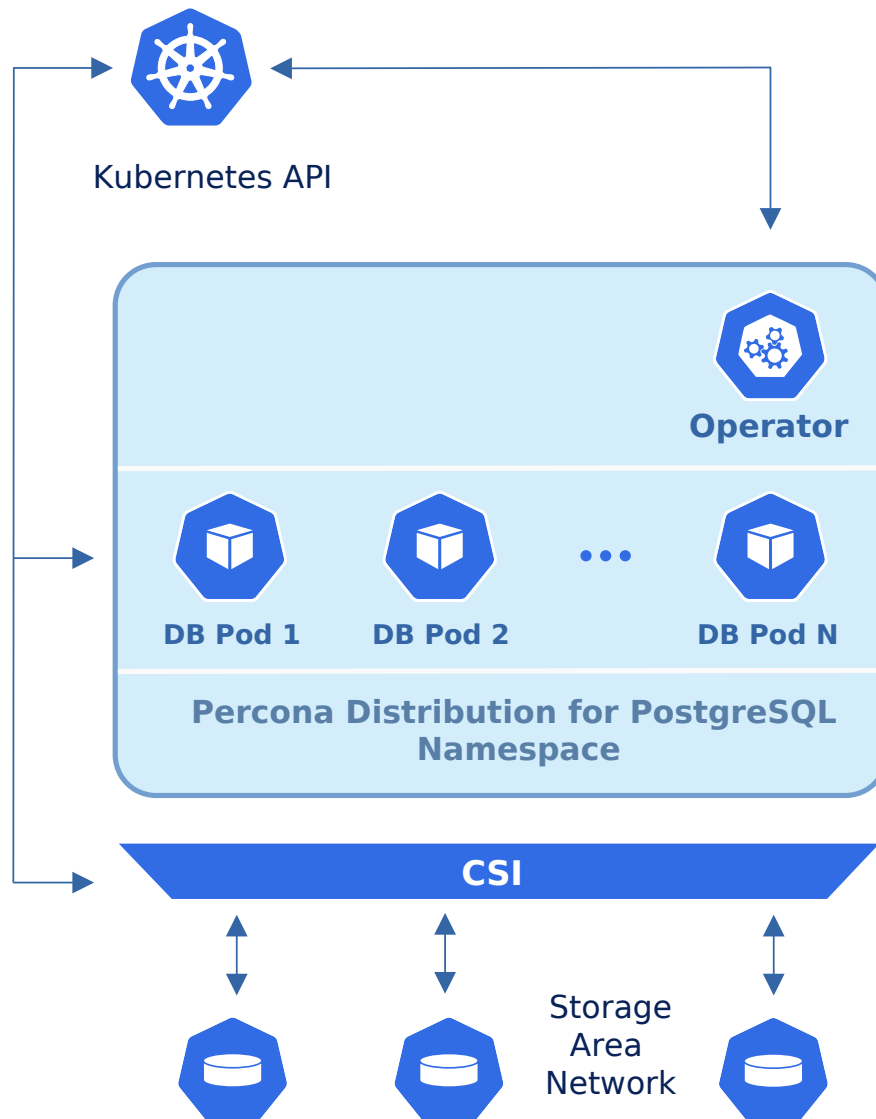
The Percona Operator for PostgreSQL automates and simplifies deploying and managing open source PostgreSQL clusters on Kubernetes. The Operator is based on [CrunchyData's PostgreSQL Operator](#).



PostgreSQL containers deployed with the Operator include the following components:

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

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



To provide data storage for stateful applications, Kubernetes uses Persistent Volumes. A *PersistentVolumeClaim* (PVC) is used to implement the automatic storage provisioning to pods. If a failure occurs, the Container Storage Interface (CSI) should be able to re-mount storage on a different node.

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

When a new Custom Resource is created or an existing one undergoes some changes or deletion, the Operator automatically creates/changes/deletes all needed Kubernetes objects with the appropriate settings to provide a proper Percona PostgreSQL Cluster operation.

Following CRDs are created while the Operator installation:

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

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#) , contact [Percona Sales](#).

Last update: 2023-04-14

9. Quickstart

9.1 Install Percona Distribution for PostgreSQL using Helm

Helm is the package manager for Kubernetes. Percona Helm charts can be found in [percona/percona-helm-charts](https://github.com/percona/percona-helm-charts) repository in Github.

9.1.1 Pre-requisites

Install Helm following its [official installation instructions](#).

Note

Helm v3 is needed to run the following steps.

9.1.2 Installation

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

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

2. Install the Percona Operator for PostgreSQL:

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

The `my-operator` parameter in the above example is the name of [a new release object](#) which is created for the Operator when you install its Helm chart (use any name you like).

Note

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

- To use different namespace, provide its name with the following additional parameter: `--namespace my-namespace`.
- To use different Helm chart version, provide it as follows: `--version 2.1.0`

3. Install PostgreSQL:

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

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

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#) , contact [Percona Sales](#).

Last update: 2023-03-03

9.2 Install Percona Distribution for PostgreSQL using kubectl

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

9.2.1 Pre-requisites

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

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

9.2.2 Install the Operator and Percona Distribution for PostgreSQL

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

1. Add the `postgres-operator` namespace to Kubernetes, not forgetting to set the correspondent context for further steps:

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

 **Note**

To use different namespace, you should edit *all occurrences* of the `namespace: postgres-operator` line in both `deploy/cr.yaml` and `deploy/bundle.yaml` configuration files.

2. Deploy the Operator with the following command:

```
$ kubectl apply --server-side -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/v2.1.0/deploy/bundle.yaml
```

 **Expected output**

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

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

3. Deploy Percona Distribution for PostgreSQL:

```
$ kubectl apply -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/v2.1.0/deploy/cr.yaml
```

 **Expected output**

```
perconapgcluster.pg.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 v2.1.0 https://github.com/percona/percona-postgresql-operator
```

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

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

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

```
$ kubectl get pods
```

 **Expected output** 

NAME	READY	STATUS	RESTARTS	AGE
cluster1-backup-7hsq-9ch48	0/1	Completed	0	35s
cluster1-instance1-mtnz-0	4/4	Running	0	87s
cluster1-pgbouncer-f4dcffc8-lrs2d	2/2	Running	0	87s
cluster1-repo-host-0	2/2	Running	0	87s
percona-postgresql-operator-75fd989d98-wvx4h	1/1	Running	0	109s

9.2.3 Verifying the cluster operation

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

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

Use `kubectl get secrets` command to see the list of Secrets objects. The Secrets object you are interested in is named as `<cluster_name>-pguser-<cluster_name>`, so the default variant will be `cluster1-pguser-cluster1`. You can use the following command to get the password of this user:

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

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

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

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

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

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

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#), contact [Percona Sales](#).

Last update: 2023-05-01

10. Installation

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

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

10.1.1 Prerequisites

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

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

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

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

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

10.1.2 Create and configure the GKE cluster

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

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

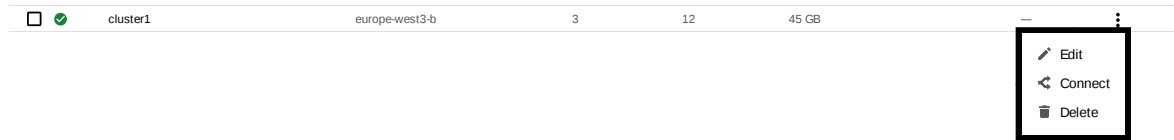
Note

You must edit the above command and other command-line statements to replace the `<project name>` placeholder with your project name. You may also be required to edit the *zone location*, which is set to `us-central1` in the above example. Other parameters specify that we are creating a cluster with 3 nodes and with machine type of 4 vCPUs and 45 GB memory.

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

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

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



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
```

10.1.3 Install the Operator and deploy your PostgreSQL cluster

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

```
$ git clone -b v2.1.0 https://github.com/percona/percona-postgresql-operator
$ cd percona-postgresql-operator
```

2. Add the `postgres-operator` namespace to Kubernetes, not forgetting to set the correspondent context for further steps:

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

 **Note**

To use different namespace, you should edit *all occurrences* of the `namespace: postgres-operator` line in both `deploy/cr.yaml` and `deploy/bundle.yaml` configuration files.

3. Deploy the operator with the following command:

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

Expected output

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

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

4. Deploy Percona Distribution for PostgreSQL:

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

Expected output

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

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

```
$ kubectl get pods
```

Expected output

NAME	READY	STATUS	RESTARTS	AGE
cluster1-backup-7hsq-9ch48	0/1	Completed	0	35s
cluster1-instance1-mtnz-0	4/4	Running	0	87s
cluster1-pgbouncer-f4dcfffc8-lrs2d	2/2	Running	0	87s
cluster1-repo-host-0	2/2	Running	0	87s
percona-postgresql-operator-75fd989d98-wvx4h	1/1	Running	0	109s

 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

10.1.4 Verifying the cluster operation

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

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

Use `kubectl get secrets` command to see the list of Secrets objects. The Secrets object you are interested in is named as `<cluster_name>-pguser-<cluster_name>`, so the default variant will be `cluster1-pguser-cluster1`. You can use the following command to get the password of this user:

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

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

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

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

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

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

10.1.5 Removing the GKE cluster

There are several ways that you can delete the cluster.


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




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

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

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

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

<input type="checkbox"/>	<input checked="" type="checkbox"/>	cluster1	eu-west-3-b	3	12	45 GB	
--------------------------	-------------------------------------	----------	-------------	---	----	-------	---

-  Edit
-  Connect
-  Delete

The cluster deletion may take time.

 **Warning**

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

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#), contact [Percona Sales](#).

Last update: 2023-05-01

10.2 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 v2.1.0 https://github.com/percona/percona-postgresql-operator
$ cd percona-postgresql-operator
```

Note

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

2. The next thing to do is to add the `postgres-operator` namespace to Kubernetes, not forgetting to set the correspondent context for further steps:

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

Note

To use different namespace, you should edit *all occurrences* of the `namespace: postgres-operator` line in both `deploy/cr.yaml` and `deploy/bundle.yaml` configuration files.

3. Deploy the operator with the following command:

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

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

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

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

```
$ kubectl get pods
```

Expected output

NAME	READY	STATUS	RESTARTS	AGE
cluster1-backup-7hsq-9ch48	0/1	Completed	0	35s
cluster1-instance1-mtnz-0	4/4	Running	0	87s
cluster1-pgbouncer-f4dcfffc8-lrs2d	2/2	Running	0	87s
cluster1-repo-host-0	2/2	Running	0	87s
percona-postgresql-operator-75fd989d98-wvx4h	1/1	Running	0	109s

10.2.1 Verifying the cluster operation

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

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

Use `kubectl get secrets` command to see the list of Secrets objects. The Secrets object you are interested in is named as `<cluster_name>-pguser-<cluster_name>`, so the default variant will be `cluster1-pguser-cluster1`. You can use the following command to get the password of this user:

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

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

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

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

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

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

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#), contact [Percona Sales](#).

Last update: 2023-05-01

11. Configuration

11.1 Users

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

- *application-level users*: the user accounts to be used by the application (probably, the unprivileged ones),
- *system-level users*: the accounts needed to automate the cluster deployment and management tasks.

The Operator creates needed system users at the cluster deployment time with generated random passwords. It can manage additional (application-level) users also if their data are placed into the Custom Resource `users` section. Changes in this section will be tracked and immediately applied by the Operator.

For example, here is a self-explanatory `deploy/cr.yaml` configuration file fragment which would add a new `rhino` user with administrative privileges over the `zoo` database:

```
...
users:
  - name: rhino
    databases:
      - zoo
    options: "SUPERUSER"
    password:
      type: ASCII
...
```

Credentials for users managed by the Operator are stored as [Kubernetes Secrets](#) objects. Each such user has its own dedicated Secret named as `<cluster_name>-<user_name>-<cluster_name>`.

By default, the Operator creates only `pguser` administrative user (the superuser), and it would have a Secret named `cluster1-pguser-cluster1` in case of the default cluster name.

Note

You can connect to PostgreSQL and login as `pguser` to PostgreSQL Pods, but `pgBouncer` (the connection pooler for PostgreSQL) doesn't allow `pguser` user access by default. That's done for security reasons.

Secrets object for each user contains `password` field stored as `data` - i.e., base64-encoded string. You can find out user's password by querying the correspondent Secret as follows (don't forget to use the real user login and cluster name instead of the `<cluster_name>-<user_name>-<cluster_name>` placeholder):

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

Note

The `{{"\n"}}` fragment at the end of the above command provides a newline to improve the readability of the command output. In case of automation (for example, in a script), this fragment can be safely omitted.

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 <cluster_name>-<user_name>-<cluster_name> -p '{"data": {"password":""}}'
```

Also, you can set a custom password for the user. Do it as follows (use the real user login and cluster name instead of the `<cluster_name>-<user_name>-<cluster_name>`, and new password instead of the `<custom_password>` placeholders):

```
$ kubectl patch secret <cluster_name>-<user_name>-<cluster_name> -p '{"stringData": {"password":"<custom_password>", "verifier":""}}'
```

CONTACT US

For free technical help, visit the [Percona Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#), contact [Percona Sales](#).

Last update: 2023-05-04

11.2 Exposing cluster

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

This document describes the usage of [Custom Resource manifest options](#) to expose the clusters deployed with the Operator.

11.2.1 PgBouncer

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

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

```
proxy:
  pgBouncer:
    replicas: 2
    image: percona/percona-postgresql-operator:2.1.0-ppg14-pgbouncer
    expose:
      type: LoadBalancer
```

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

```
$ kubectl get service
```

Expected output

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

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

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

11.2.2 Exposing the cluster without PgBouncer

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

```
$ kubectl get service
```

Expected output

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

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

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

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

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#), contact [Percona Sales](#).

Last update: 2023-05-04

11.3 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.

11.3.1 Affinity and anti-affinity

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

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

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

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

You can see the explanation of these affinity options in [Kubernetes documentation](#).

11.3.2 Topology Spread Constraints

Topology Spread Constraints allow you to control how Pods are distributed across the cluster based on regions, zones, nodes, and other topology specifics. This can be useful for both high availability and resource efficiency.

Pod topology spread constraints are controlled by the `topologySpreadConstraints` subsection, which can be put into `proxy.pgBouncer` and `backups.pgbackrest.repoHost` sections of the `deploy/cr.yaml` configuration file as follows:

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

You can see the explanation of these affinity options in [Kubernetes documentation](#).

11.3.3 Tolerations

Tolerations allow Pods having them to be able to land onto nodes with matching *taints*. Tolerations are expressed as a key with an operator, which is either `exists` or `equal` (the latter variant also requires a value the key is equal to). Moreover, a 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 a 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 Tolerations](#) contains more examples on this topic.

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#), contact [Percona Sales](#).

Last update: 2022-12-30

11.4 Transport Layer Security (TLS)

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

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

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

TLS security can be configured in several ways:

- the Operator can generate certificates automatically at cluster creation time,
- you can also generate certificates manually.

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

11.4.1 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.

11.4.2 Check connectivity to the cluster

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

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

```

- name: ca
  secret:
    secretName: <cluster_name>-ssl-ca
    items:
      - key: ca.crt
        path: ca.crt
        mode: 0777
EOF

```

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

```

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

```

Now you should see the prompt of PostgreSQL interactive terminal:

```

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

```

11.4.3 Keep certificates after deleting the cluster

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

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

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#), contact [Percona Sales](#).

Last update: 2023-05-02

11.5 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 2.1.0 --namespace my-namespace --set
disable_telemetry="true"
```

- if you don't use helm for installation, you have to edit the `operator.yaml` before applying it with the `kubectl apply -f deploy/operator.yaml` command. Open the `operator.yaml` file with your text editor, find the `disable_telemetry` key and set it to `true`:

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

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#), contact [Percona Sales](#).

12. Management

12.1 Providing Backups

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

The Operator uses the open source [pgBackRest](#) backup and restore utility.

Backup repositories

A special *pgBackRest repository* is created by the Operator along with creating a new PostgreSQL cluster to facilitate the usage of the [pgBackRest](#) features in it.

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

- Amazon S3, or [any S3-compatible storage](#),
- [Google Cloud Storage](#),
- [Azure Blob Storage](#)

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

Each [pgBackRest](#) repository consists of the following Kubernetes objects:

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

12.1.1 Backup types

The PostgreSQL Operator supports three types of [pgBackRest](#) backups:

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

12.1.2 Backup retention

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

Backup retention can be controlled by the following [pgBackRest](#) options:

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

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

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

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

12.1.3 Backup storage

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

Configuring the S3-compatible backup storage

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

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

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

- Put your AWS S3 key and AWS S3 key secret into the base64 encoded pgBackRest configuration as follows:

in Linux

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

in macOS

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

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

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



Note

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

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

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

- Update your `deploy/cr.yaml` configuration with the your S3 credentials Secret in the `backups.pgbackrest.configuration` subsection, and put all other S3 related information into the options of one of your repositories in the `backups.pgbackrest.repos` subsection. For example, the S3 storage for the `repo2` repository would look 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>"
```

4. Finally, create or update the cluster:

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

Configuring Google Cloud Storage for backups

You can configure [Google Cloud Storage](#) as an object store for backups similarly to S3 storage.

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

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


1. Create your service account key following the [official Google Cloud instructions](#).
2. Export this key from your Google Cloud account.

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

← my-service-account

DETAILS PERMISSIONS **KEYS** METRICS LOGS

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](#)

ADD KEY ▾

Click the *ADD KEY* button, chose *Create new key* and chose *JSON* as a key type. These actions will result in downloading a file in JSON format with your new private key and related information.

3. Now you should use a base64-encoded version of this file and create the [Kubernetes Secret](#). You can encode the file with the `base64 <filename>` command. When done, create a yaml file with your cluster name and base64-encoded file contents as the following `cluster1-pgbackrest-secrets.yaml` example:

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

Note

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

Create the Secrets object from this yaml file:

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

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

```
...
backups:
  pgbackrest:
    ...
    configuration:
      - secret:
          name: cluster1-pgbackrest-secrets
    ...
  repos:
    ...
    - name: repo3
```

```
gcs:  
  bucket: "<YOUR_GCS_BUCKET_NAME>"
```

5. Finally, create or update the cluster:

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

Configuring Azure Blob Storage for backups

You can configure [Microsoft Azure Blob Storage](#) as an object store for backups similarly to S3 or GCS storage.

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

The Operator will also need a [Kubernetes Secret](#) with your Azure Storage credentials to access the storage.

- Put your Azure storage account name and key into the base64 encoded pgBackRest configuration as follows:

in Linux

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

in macOS

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

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

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



Note

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

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

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

- Update your `deploy/cr.yaml` configuration with the your S3 credentials Secret in the `backups.pgbackrest.configuration` subsection, and put all other S3 related information into the options of one of your repositories in the `backups.pgbackrest.repos` subsection. For example, the S3 storage for the `repo4` repository would look as follows.

```
...
backups:
  pgbackrest:
    ...
    configuration:
      - secret:
          name: cluster1-pgbackrest-secrets
    ...
  repos:
    ...
    - name: repo4
      azure:
        container: "<YOUR_AZURE_CONTAINER>"
```

4. Finally, create or update the cluster:

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

12.1.4 Scheduling backups

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

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

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

The schedule is specified in crontab format as explained in [Custom Resource options](#).

12.1.5 Making on-demand backup

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

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

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

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

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

12.1.6 Restore the cluster from a previously saved backup

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

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

Restore to an existing PostgreSQL cluster

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

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

The following keys are the most important ones:

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

The actual restoration process can be started as follows:

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

Restore the cluster with point-in-time recovery

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

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

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

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

Note

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

Restore to a new PostgreSQL cluster

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

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

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

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

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#), contact [Percona Sales](#).

Last update: 2023-05-03

12.2 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 MySQL nodes; horizontal scaling is about adding more nodes to the cluster. High availability looks technically similar, because it also involves additional nodes, but the reason is maintaining liveness of the system in case of server or network failures.

12.2.1 Vertical scaling

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

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

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

Use our reference documentation for the [Custom Resource options](#) for more details about other components.

12.2.2 High availability

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

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

12.2.3 Using `spec.instances.replicas`

For example, you have the following Custom Resource manifest:

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

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

```
spec:
  ...
```

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

...and applying the Custom Resource:

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

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

12.2.4 Using spec.instances

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

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

For example, you have the following Custom Resource manifest:

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

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

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

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

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#), contact [Percona Sales](#).

Last update: 2023-05-04

12.3 Monitoring

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

Note

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

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

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

12.3.1 Installing the PMM Server

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

12.3.2 Installing the PMM Client

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

1. The PMM client installation is initiated by updating the `pmm` section in the `deploy/cr.yaml` file.

- set `pmm.enabled=true`
- set the `pmm.serverHost` key to your PMM Server hostname,
- check that the `serverUser` key contains your PMM Server user name (`admin` by default),
- make sure the `pmmserver` key in the `deploy/pmm-secret.yaml` secrets file contains the password specified for the PMM Server during its installation.

Apply changes with the `kubectl apply -f deploy/pmm-secret.yaml` command.

Info

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

in Linux

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

in macOS

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

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

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

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

```
$ kubectl get pods
$ kubectl logs cluster1-7b7f7898d5-7f5pz -c pmm-client
```

3. Now you can access PMM via `https` in a web browser, with the login/password authentication, and the browser is configured to show Percona Distribution for PostgreSQL metrics.

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#), contact [Percona Sales](#).

Last update: 2022-12-20

12.4 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.

12.4.1 Adding a sidecar container

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

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

Apply your modifications as usual:

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

Note

More options suitable for the `sidecars` subsection can be found in the [Custom Resource options reference](#).

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

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

Expected output

```
Name:          cluster1-instance1-n8v4-0
....
Containers:
....
my-sidecar-1:
  Container ID:  docker://f0c3437295d0ec819753c581aae174a0b8d062337f80897144eb8148249ba742
  Image:        busybox
  Image ID:     docker-pullable://
busybox@sha256:139abcf41943b8bcd4bc5c42ee71ddc9402c7ad69ad9e177b0a9bc4541f14924
  Port:        <none>
  Host Port:   <none>
  Command:
    /bin/sh
  Args:
    -c
    while true; do echo echo $(date -u) 'test' >> /dev/null; sleep 5; done
  State:       Running
    Started:    Thu, 11 Nov 2021 10:38:15 +0300
  Ready:       True
  Restart Count: 0
  Environment: <none>
  Mounts:
    /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-fbrbn (ro)
....
```

12.4.2 Getting shell access to a sidecar container

You can login to your sidecar container as follows:

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

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#) , contact [Percona Sales](#).

Last update: 2023-05-03

12.5 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
```

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#), contact [Percona Sales](#).

Last update: 2023-05-03

13. Reference

13.1 Custom Resource options

The Cluster is configured via the [deploy/cr.yaml](#) file.

The metadata part of this file contains the following keys:

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

The spec part of the [deploy/cr.yaml](#) file contains the following:

Key	<code>standby.enabled</code>
Value	boolean
Example	<code>false</code>
Description	Enables or disables running the cluster in a standby mode (read-only copy of an existing cluster, useful for disaster recovery, etc)
Key	<code>standby.host</code>
Value	string
Example	<code>"<primary-ip>"</code>
Description	Host address of the primary cluster this standby cluster connects to
Key	<code>standby.port</code>
Value	string
Example	<code>"<primary-port>"</code>
Description	Port number used by a standby copy to connect to the primary cluster
Key	<code>openshift</code>
Value	boolean
Example	<code>true</code>
Description	Set to <code>true</code> if the cluster is being deployed on OpenShift, set to <code>false</code> otherwise, or unset it for autodetection
Key	<code>standby.repoName</code>
Value	string
Example	<code>repo1</code>
Description	Name of the pgBackRest repository in the primary cluster this standby cluster connects to
Key	<code>secrets.customTLSSecret.name</code>
Value	string
Example	<code>cluster1-cert</code>
Description	A secret with TLS certificate generated for <i>external</i> communications, see Transport Layer Security (TLS) for details
Key	<code>secrets.customReplicationTLSSecret.name</code>
Value	string
Example	<code>replication1-cert</code>
Description	A secret with TLS certificate generated for <i>internal</i> communications, see Transport Layer Security (TLS) for details
Key	<code>users.name</code>
Value	string

Example	<code>rhino</code>
Description	The name of the PostgreSQL user
Key	<code>users.databases</code>
Value	string
Example	<code>zoo</code>
Description	Databases accessible by a specific PostgreSQL user with rights to create objects in them (the option is ignored for <code>postgres</code> user; also, modifying it can't be used to revoke the already given access)
Key	<code>users.password.type</code>
Value	string
Example	<code>ASCII</code>
Description	The set of characters used for password generation: can be either <code>ASCII</code> (default) or <code>AlphaNumeric</code>
Key	<code>users.options</code>
Value	string
Example	<code>"SUPERUSER"</code>
Description	The <code>ALTER ROLE</code> options other than password (the option is ignored for <code>postgres</code> user)
Key	<code>databaseInitSQL.key</code>
Value	string
Example	<code>init.sql</code>
Description	Data key for the Custom configuration options ConfigMap with the init SQL file, which will be executed at cluster creation time
Key	<code>databaseInitSQL.name</code>
Value	string
Example	<code>cluster1-init-sql</code>
Description	Name of the ConfigMap with the init SQL file, which will be executed at cluster creation time
Key	<code>pause</code>
Value	string
Example	<code>false</code>
Description	Setting it to <code>true</code> gracefully stops the cluster, scaling workloads to zero and suspending CronJobs; setting it to <code>false</code> after shut down starts the cluster back
Key	<code>unmanaged</code>
Value	string
Example	<code>false</code>

Description	Setting it to <code>true</code> stops the Operator's activity including the rollout and reconciliation of changes made in the Custom Resource; setting it to <code>false</code> starts the Operator's activity back
Key	<code>dataSource.postgresCluster.clusterName</code>
Value	string
Example	<code>cluster1</code>
Description	Name of an existing cluster to use as the data source when restoring backup to a new cluster
Key	<code>dataSource.postgresCluster.repoName</code>
Value	string
Example	<code>repo1</code>
Description	Name of the <code>pgBackRest</code> repository in the source cluster that contains the backup to be restored to a new cluster
Key	<code>dataSource.postgresCluster.options</code>
Value	string
Example	
Description	The <code>pgBackRest</code> command-line options for the <code>pgBackRest</code> restore command
Key	<code>dataSource.pgbackrest.stanza</code>
Value	string
Example	<code>db</code>
Description	Name of the <code>pgBackRest</code> stanza to use as the data source when restoring backup to a new cluster
Key	<code>dataSource.pgbackrest.configuration.secret.name</code>
Value	string
Example	<code>pgo-s3-creds</code>
Description	Name of the <code>Kubernetes Secret</code> object with custom <code>pgBackRest</code> configuration, which will be added to the <code>pgBackRest</code> configuration generated by the Operator
Key	<code>dataSource.pgbackrest.global</code>
Value	subdoc
Example	<code>/pgbackrest/postgres-operator/hippo/repo1</code>
Description	Settings, which are to be included in the <code>global</code> section of the <code>pgBackRest</code> configuration generated by the Operator
Key	<code>dataSource.pgbackrest.repo.name</code>
Value	string
Example	<code>repo1</code>
Description	Name of the <code>pgBackRest</code> repository

Key	dataSource.pgbackrest.repo.s3.bucket
Value	string
Example	"my-bucket"
Description	The Amazon S3 bucket or Google Cloud Storage bucket name used for backups
Key	dataSource.pgbackrest.repo.s3.endpoint
Value	string
Example	"s3.ca-central-1.amazonaws.com"
Description	The endpoint URL of the S3-compatible storage to be used for backups (not needed for the original Amazon S3 cloud)
Key	dataSource.pgbackrest.repo.s3.region
Value	boolean
Example	"ca-central-1"
Description	The AWS region to use for Amazon and all S3-compatible storages
Key	image
Value	string
Example	perconalab/percona-postgresql-operator:main-pgg14-postgres
Description	The PostgreSQL Docker image to use
Key	imagePullPolicy
Value	string
Example	Always
Description	This option is used to set the policy for updating PostgreSQL images
Key	postgresVersion
Value	int
Example	14
Description	The major version of PostgreSQL to use
Key	port
Value	int
Example	5432
Description	The port number for PostgreSQL
Key	expose.annotations
Value	label

Example	<code>my-annotation: value1</code>
Description	The Kubernetes annotations metadata for PostgreSQL
Key	<code>expose.labels</code>
Value	label
Example	<code>my-label: value2</code>
Description	Set labels for the PostgreSQL Service
Key	<code>expose.type</code>
Value	string
Example	<code>LoadBalancer</code>
Description	Specifies the type of Kubernetes Service for PostgreSQL
Key	<code>instances.name</code>
Value	string
Example	<code>rs-0</code>
Description	The name of the PostgreSQL instance
Key	<code>instances.replicas</code>
Value	int
Example	<code>3</code>
Description	The number of Replicas to create for the PostgreSQL instance
Key	<code>instances.resources.limits.cpu</code>
Value	string
Example	<code>2.0</code>
Description	Kubernetes CPU limits for a PostgreSQL instance
Key	<code>instances.resources.limits.memory</code>
Value	string
Example	<code>4Gi</code>
Description	The Kubernetes memory limits for a PostgreSQL instance
Key	<code>instances.sidecars.image</code>
Value	string
Example	<code>mycontainer1:latest</code>
Description	Image for the custom sidecar container for PostgreSQL Pods
Key	<code>instances.sidecars.name</code>
Value	string

Example	<code>testcontainer</code>
Description	Name of the custom sidecar container for PostgreSQL Pods
Key	<code>instances.sidecars.imagePullPolicy</code>
Value	string
Example	<code>Always</code>
Description	This option is used to set the policy for the PostgreSQL Pod sidecar container
Key	<code>instances.sidecars.env</code>
Value	subdoc
Example	
Description	The environment variables set as key-value pairs for the custom sidecar container for PostgreSQL Pods
Key	<code>instances.sidecars.envFrom</code>
Value	subdoc
Example	
Description	The environment variables set as key-value pairs in ConfigMaps for the custom sidecar container for PostgreSQL Pods
Key	<code>instances.sidecars.command</code>
Value	array
Example	<code>["/bin/sh"]</code>
Description	Command for the custom sidecar container for PostgreSQL Pods
Key	<code>instances.sidecars.args</code>
Value	array
Example	<code>["-c", "while true; do trap 'exit 0' SIGINT SIGTERM SIGQUIT SIGKILL; done;"]</code>
Description	Command arguments for the custom sidecar container for PostgreSQL Pods
Key	<code>instances.topologySpreadConstraints.maxSkew</code>
Value	int
Example	<code>1</code>
Description	The degree to which Pods may be unevenly distributed under the Kubernetes Pod Topology Spread Constraints
Key	<code>instances.topologySpreadConstraints.topologyKey</code>
Value	string
Example	<code>my-node-label</code>
Description	The key of node labels for the Kubernetes Pod Topology Spread Constraints
Key	<code>instances.topologySpreadConstraints.whenUnsatisfiable</code>

Value	string
Example	<code>DoNotSchedule</code>
Description	What to do with a Pod if it doesn't satisfy the Kubernetes Pod Topology Spread Constraints
Key	<code>instances.topologySpreadConstraints.labelSelector.matchLabels</code>
Value	label
Example	<code>postgres-operator.crunchydata.com/instance-set: instance1</code>
Description	The Label selector for the Kubernetes Pod Topology Spread Constraints
Key	<code>instances.tolerations.effect</code>
Value	string
Example	<code>NoSchedule</code>
Description	The Kubernetes Pod tolerations effect for the PostgreSQL instance
Key	<code>instances.tolerations.key</code>
Value	string
Example	<code>role</code>
Description	The Kubernetes Pod tolerations key for the PostgreSQL instance
Key	<code>instances.tolerations.operator</code>
Value	string
Example	<code>Equal</code>
Description	The Kubernetes Pod tolerations operator for the PostgreSQL instance
Key	<code>instances.tolerations.value</code>
Value	string
Example	<code>connection-poolers</code>
Description	The Kubernetes Pod tolerations value for the PostgreSQL instance
Key	<code>instances.priorityClassName</code>
Value	string
Example	<code>high-priority</code>
Description	The Kuberentes Pod priority class for PostgreSQL instance Pods
Key	<code>instances.walVolumeClaimSpec.accessModes</code>
Value	string
Example	<code>ReadWriteOnce</code>
Description	The Kubernetes PersistentVolumeClaim access modes for the PostgreSQL Write-ahead Log storage

Key	<code>instances.walVolumeClaimSpec.resources.requests.storage</code>
Value	string
Example	1Gi
Description	The Kubernetes storage requests for the storage the PostgreSQL instance will use
Key	<code>instances.dataVolumeClaimSpec.accessModes</code>
Value	string
Example	ReadWriteOnce
Description	The Kubernetes PersistentVolumeClaim access modes for the PostgreSQL Write-ahead Log storage
Key	<code>instances.dataVolumeClaimSpec.resources.requests.storage</code>
Value	string
Example	1Gi
Description	The Kubernetes storage requests for the storage the PostgreSQL instance will use

13.1.1 Backup Section

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

Key	<code>backups.pgbackrest.image</code>
Value	string
Example	<code>perconalab/percona-postgresql-operator:main-pg14-pgbackrest</code>
Description	The Docker image for pgBackRest
Key	<code>backups.pgbackrest.configuration.secret.name</code>
Value	string
Example	<code>cluster1-pgbackrest-secrets</code>
Description	Name of the Kubernetes Secret object with custom pgBackRest configuration, which will be added to the pgBackRest configuration generated by the Operator
Key	<code>backups.pgbackrest.jobs.priorityClassName</code>
Value	string
Example	<code>high-priority</code>
Description	The Kuberentes Pod priority class for pgBackRest jobs
Key	<code>backups.pgbackrest.jobs.resources.limits.cpu</code>
Value	int
Example	<code>200</code>
Description	Kubernetes CPU limits for a pgBackRest job
Key	<code>backups.pgbackrest.jobs.resources.limits.memory</code>
Value	int
Example	<code>128Mi</code>
Description	The Kubernetes memory limits for a pgBackRest job
Key	<code>backups.pgbackrest.jobs.tolerations.effect</code>
Value	string
Example	<code>NoSchedule</code>
Description	The Kubernetes Pod tolerations effect for a pgBackRest job
Key	<code>backups.pgbackrest.jobs.tolerations.key</code>
Value	string
Example	<code>role</code>
Description	The Kubernetes Pod tolerations key for a pgBackRest job
Key	<code>backups.pgbackrest.jobs.tolerations.operator</code>
Value	string
Example	<code>Equal</code>
Description	The Kubernetes Pod tolerations operator for a pgBackRest job

Key	<code>backups.pgbackrest.jobs.tolerations.value</code>
Value	string
Example	<code>connection-poolers</code>
Description	The Kubernetes Pod tolerations value for a pgBackRest job
Key	<code>backups.pgbackrest.global</code>
Value	subdoc
Example	<code>/pgbackrest/postgres-operator/hippo/repo1</code>
Description	Settings, which are to be included in the <code>global</code> section of the pgBackRest configuration generated by the Operator
Key	<code>backups.pgbackrest.repoHost.priorityClassName</code>
Value	string
Example	<code>high-priority</code>
Description	The Kuberentes Pod priority class for pgBackRest repo
Key	<code>backups.pgbackrest.repoHost.topologySpreadConstraints.maxSkew</code>
Value	int
Example	<code>1</code>
Description	The degree to which Pods may be unevenly distributed under the Kubernetes Pod Topology Spread Constraints
Key	<code>backups.pgbackrest.repoHost.topologySpreadConstraints.topologyKey</code>
Value	string
Example	<code>my-node-label</code>
Description	The key of node labels for the Kubernetes Pod Topology Spread Constraints
Key	<code>backups.pgbackrest.repoHost.topologySpreadConstraints.whenUnsatisfiable</code>
Value	string
Example	<code>ScheduleAnyway</code>
Description	What to do with a Pod if it doesn't satisfy the Kubernetes Pod Topology Spread Constraints
Key	<code>backups.pgbackrest.repoHost.topologySpreadConstraints.labelSelector.matchLabels</code>
Value	label
Example	<code>postgres-operator.crunchydata.com/pgbackrest: ""</code>
Description	The Label selector for the Kubernetes Pod Topology Spread Constraints
Key	<code>backups.pgbackrest.repoHost.affinity.podAntiAffinity</code>
Value	subdoc
Example	

Description	Pod anti-affinity, allows setting the standard Kubernetes affinity constraints of any complexity
Key	<code>backups.pgbackrest.manual.repoName</code>
Value	string
Example	<code>repo1</code>
Description	Name of the pgBackRest repository for on-demand backups
Key	<code>backups.pgbackrest.manual.options</code>
Value	string
Example	<code>--type=full</code>
Description	The on-demand backup command-line options which will be passed to pgBackRest for on-demand backups
Key	<code>backups.pgbackrest.repos.name</code>
Value	string
Example	<code>repo1</code>
Description	Name of the pgBackRest repository for backups
Key	<code>backups.pgbackrest.repos.schedules.full</code>
Value	string
Example	<code>0 0 * * 6</code>
Description	Scheduled time to make a full backup specified in the crontab format
Key	<code>backups.pgbackrest.repos.schedules.differential</code>
Value	string
Example	<code>0 0 * * 6</code>
Description	Scheduled time to make a differential backup specified in the crontab format
Key	<code>backups.pgbackrest.repos.volume.volumeClaimSpec.accessModes</code>
Value	string
Example	<code>ReadWriteOnce</code>
Description	The Kubernetes PersistentVolumeClaim access modes for the pgBackRest Storage
Key	<code>backups.pgbackrest.repos.volume.volumeClaimSpec.resources.requests.storage</code>
Value	string
Example	<code>1Gi</code>
Description	The Kubernetes storage requests for the pgBackRest storage
Key	<code>backups.pgbackrest.repos.s3.bucket</code>
Value	string

Example	<code>"my-bucket"</code>
Description	The Amazon S3 bucket
	name used for backups
Key	<code>backups.pgbackrest.repos.s3.endpoint</code>
Value	string
Example	<code>"s3.ca-central-1.amazonaws.com"</code>
Description	The endpoint URL of the S3-compatible storage to be used for backups (not needed for the original Amazon S3 cloud)
Key	<code>backups.pgbackrest.repos.s3.region</code>
Value	boolean
Example	<code>"ca-central-1"</code>
Description	The AWS region to use for Amazon and all S3-compatible storages
Key	<code>backups.pgbackrest.repos.gcs.bucket</code>
Value	string
Example	<code>"my-bucket"</code>
Description	The Google Cloud Storage bucket
	name used for backups
Key	<code>backups.pgbackrest.repos.azure.container</code>
Value	string
Example	<code>my-container</code>
Description	Name of the Azure Blob Storage container for backups
Key	<code>backups.restore.enabled</code>
Value	boolean
Example	<code>false</code>
Description	Enables or disables restoring a previously made backup
Key	<code>backups.restore.repoName</code>
Value	string
Example	<code>repo1</code>
Description	Name of the pgBackRest repository that contains the backup to be restored
Key	<code>backups.restore.options</code>
Value	string
Example	

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

13.1.2 PMM Section

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

Key	<code>pmm.enabled</code>
Value	boolean
Example	<code>false</code>
Description	Enables or disables monitoring Percona Distribution for PostgreSQL cluster with PMM
Key	<code>pmm.image</code>
Value	string
Example	<code>percona/pmm-client:2.37.0</code>
Description	Percona Monitoring and Management (PMM) Client Docker image
Key	<code>pmm.imagePullPolicy</code>
Value	string
Example	<code>IfNotPresent</code>
Description	This option is used to set the policy for updating PMM Client images
Key	<code>pmm.pmmSecret</code>
Value	string
Example	<code>cluster1-pmm-secret</code>
Description	Name of the Kubernetes Secret object for the PMM Server password
Key	<code>pmm.serverHost</code>
Value	string
Example	<code>monitoring-service</code>
Description	Address of the PMM Server to collect data from the cluster

13.1.3 proxy Section

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

Key	<code>proxy.pgBouncer.replicas</code>
Value	int
Example	3
Description	The number of the pgBouncer Pods to provide connection pooling
Key	<code>proxy.pgBouncer.image</code>
Value	string
Example	<code>perconalab/percona-postgresql-operator:main-pg14-pgbouncer</code>
Description	Docker image for the pgBouncer connection pooler
Key	<code>proxy.pgBouncer.exposePostgresUser</code>
Value	boolean
Example	<code>false</code>
Description	Enables or disables exposing postgres user through pgBouncer
Key	<code>proxy.pgBouncer.resources.limits.cpu</code>
Value	int
Example	<code>200m</code>
Description	Kubernetes CPU limits for a pgBouncer container
Key	<code>proxy.pgBouncer.resources.limits.memory</code>
Value	int
Example	<code>128Mi</code>
Description	The Kubernetes memory limits for a pgBouncer container
Key	<code>proxy.pgBouncer.expose.type</code>
Value	string
Example	<code>ClusterIP</code>
Description	Specifies the type of Kubernetes Service for pgBouncer
Key	<code>proxy.pgBouncer.expose.annotations</code>
Value	label
Example	<code>pg-cluster-annot: cluster1</code>
Description	The Kubernetes annotations metadata for pgBouncer
Key	<code>proxy.pgBouncer.expose.labels</code>
Value	label
Example	<code>pg-cluster-label: cluster1</code>
Description	Set labels for the pgBouncer Service

Value	string
Example	preferred
Description	Pod anti-affinity type, can be either preferred or required
Key	proxy.pgBouncer.sidecars.image
Value	string
Example	mycontainer1:latest
Description	Image for the custom sidecar container for pgBouncer Pods
Key	proxy.pgBouncer.sidecars.name
Value	string
Example	testcontainer
Description	Name of the custom sidecar container for pgBouncer Pods
Key	proxy.pgBouncer.sidecars.imagePullPolicy
Value	string
Example	Always
Description	This option is used to set the policy for the pgBouncer Pod sidecar container
Key	proxy.pgBouncer.sidecars.env
Value	subdoc
Example	
Description	The environment variables set as key-value pairs for the custom sidecar container for pgBouncer Pods
Key	proxy.pgBouncer.sidecars.envFrom
Value	subdoc
Example	
Description	The environment variables set as key-value pairs in ConfigMaps for the custom sidecar container for pgBouncer Pods
Key	proxy.pgBouncer.sidecars.command
Value	array
Example	["/bin/sh"]
Description	Command for the custom sidecar container for pgBouncer Pods
Key	proxy.pgBouncer.sidecars.args
Value	array
Example	["-c", "while true; do trap 'exit 0' SIGINT SIGTERM SIGQUIT SIGKILL; done;"]
Description	Command arguments for the custom sidecar container for pgBouncer Pods

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

13.1.4 patroni Section

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

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

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#), contact [Percona Sales](#).

Last update: 2023-05-04

13.2 Percona certified images

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

Image	Digest
percona/percona-postgresql-operator:2.1.0	538076c55e2fb2e7c8964a793dfaf56dea4ce276e839e0cd8ac7da54966784a8
percona/percona-postgresql-operator:2.1.0-ppg12-postgres	950b63972fb5ddcdb50adca3306b3894ae6c2c64d16c36096101a64a6bc9bbb2
percona/percona-postgresql-operator:2.1.0-ppg13-postgres	e728d6f70b427e7752703cb20b38b456090e47079535d2f62ef224f7544bddfd
percona/percona-postgresql-operator:2.1.0-ppg14-postgres	1f1ae0278071f3331db3fbd368baa0ef2c28fadd497e2d8fb7ad5904f5113945
percona/percona-postgresql-operator:2.1.0-ppg15-postgres	a00da56a3a907b585a6eab8b4522864285878291240e07914c060eefbbc7c543
percona/percona-postgresql-operator:2.1.0-ppg12-pgbouncer	cd52276253214f37f66714dcd5a9fcafa21e2aff02048e8fda2f3d5affba13ca
percona/percona-postgresql-operator:2.1.0-ppg13-pgbouncer	cdeb1de7882067a49d20024a7959cadb5efa627420bad6a95e4036b1643aa7a4
percona/percona-postgresql-operator:2.1.0-ppg14-pgbouncer	c144eb7bf7c0f332785ec49785f5c974325c9bd467488d9e624ca8d744da4cfc
percona/percona-postgresql-operator:2.1.0-ppg15-pgbouncer	ff89002b697ad820e638410e8851d8b71ec2b2c1b008dc18ec0aaf69ff03da91
percona/percona-postgresql-operator:2.1.0-ppg12-pgbackrest	8b3515ad7bd0fd572d2ee3334eeb8f6c213baf39be65ff2c1dde5c1b50fe4ea2
percona/percona-postgresql-operator:2.1.0-ppg13-pgbackrest	05fb839203eccc2f3151bf72df3bccc31d5d6409d6bc7ba288fcdf06896c5029
percona/percona-postgresql-operator:2.1.0-ppg14-pgbackrest	80df91c4d8d9092351ff33f2d476df74366c4eb82c296dbc4edb307fc0407171
percona/percona-postgresql-operator:2.1.0-ppg15-pgbackrest	06ca64615c324ae97c79e95349f33995cc47843caa452ce9740ea4ccd3a455e1
percona/pmm-client:2.37.0	e1e2f4cbfd4ce4be5d883330d810e9962a62531e2da07f1b115077a49ff97ed5

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#) , contact [Percona Sales](#).

Last update: 2023-05-04

14. Release Notes

14.1 Percona Operator for PostgreSQL Release Notes

- *Percona Operator for PostgreSQL 2.1.0 Tech preview (2023-05-04)*
- *Percona Operator for PostgreSQL 2.0.0 Tech preview (2022-12-30)*

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#) , contact [Percona Sales](#).

Last update: 2023-05-04

14.2 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.

14.2.1 Release Highlights

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

14.2.2 New Features

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

14.2.3 Improvements

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

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

14.2.4 Bugs Fixed

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

14.2.5 Supported platforms

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

- [Google Kubernetes Engine \(GKE\)](#) 1.23 - 1.25
- [Amazon Elastic Container Service for Kubernetes \(EKS\)](#) 1.23 - 1.25

This list only includes the platforms that the Percona Operators are specifically tested on as part of the release process. Other Kubernetes flavors and versions depend on the backward compatibility offered by Kubernetes itself.

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#), contact [Percona Sales](#).

Last update: 2023-05-04

14.3 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.

14.3.1 Architecture

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

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

14.3.2 Backups

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

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

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

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

14.3.3 Operations

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

Within the same cluster users can deploy [multiple instances](#). These instances are going to have the same data, but can have different configuration and resources. This can be useful if you plan to migrate to new hardware or need to test the new topology.

Each postgresSQL node can have [sidecar containers](#) now to provide integration with your existing tools or expand the capabilities of the cluster.

14.3.4 Try it out now

Excited with what you read above?

- We encourage you to install the Operator following [our documentation](#).
- Feel free to share feedback with us on the [forum](#) or raise a bug or feature request in [JIRA](#).
- See the source code in our [Github repository](#).

CONTACT US

For free technical help, visit the Percona [Community Forum](#).

To report bugs or submit feature requests, open a [JIRA](#) ticket.

For paid [support](#) and [managed](#) or [consulting services](#) , contact [Percona Sales](#).

Last update: 2022-12-30