



Operator for PostgreSQL 3.0.0

(May 22, 2026)

Documentation

Table of Contents

[Home](#)

[Understand the Operator](#)

[Kubernetes Operator concepts](#)

[Features and capabilities](#)

[How the Operator works](#)

[Cluster architecture](#)

[Comparison with other solutions](#)

[Known limitations](#)

[Get help from Percona](#)

[Quickstart guide](#)

[Overview](#)

[System requirements](#)

[1 Quick install](#)

[With kubectl](#)

[With Helm](#)

[2 Connect to PostgreSQL](#)

[3 Insert data](#)

[4 Make a backup](#)

[5 Monitor the database with PMM](#)

[What's next](#)

[Installation](#)

[Install on Minikube](#)

[Install with OpenEverest](#)

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

[Install on Amazon Elastic Kubernetes Service \(AWS EKS\)](#)

[Install on Microsoft Azure Kubernetes Service \(AKS\)](#)

[Install on OpenShift](#)

[Generic Kubernetes installation](#)

[Configuration](#)

[Application and system users](#)

[LDAP authentication](#)

[About LDAP authentication](#)

[Configure LDAP authentication](#)

[Exposing the cluster](#)

[Changing PostgreSQL options](#)

[Anti-affinity and tolerations](#)

[Labels and annotations](#)

[Transport encryption \(TLS/SSL\)](#)

[About TLS/SSL](#)

[Configure TLS/SSL with the Operator using cert-manager](#)

[Migrate from Operator-generated certificates to cert-manager](#)

[Migrate from cert-manager to custom TLS certificates](#)

[Generate certificates manually](#)

[Update TLS/SSL certificates](#)

[Check TLS communication](#)

[Telemetry](#)

[Configure concurrency for a cluster reconciliation](#)

[Environment variables](#)

[About environment variables](#)

[Operator environment variables](#)

[Cluster component environment variables](#)

[Management](#)

[Back up and restore](#)

[About backups](#)

[Configure storage for backups](#)

[Overview](#)

[Amazon S3 storage](#)

[Google Cloud Storage](#)

[Microsoft Azure Blob Storage](#)

[Persistent Volume](#)

[Make a backup](#)

[Scheduled backup](#)

[On-demand backup](#)

[Restore from a backup](#)

[Restore options](#)

[To the same cluster \(in-place restore\)](#)

[To a new cluster \(cluster clone\)](#)

[PVC snapshots](#)

[About PVC snapshots](#)

[Configure PVC snapshots](#)

[Configure PVC snapshots on EKS](#)

[Use PVC snapshots](#)

[Backup encryption](#)

[Speed up backups](#)

[Backup retention](#)

[Delete the unneeded backup](#)

[Disable backups](#)

[Deploy a standby cluster for Disaster Recovery](#)

[Introduction](#)

[Deploy standby cluster based on backups](#)

[Deploy standby cluster based on streaming replication](#)

[Failover](#)

[Scale your cluster](#)

[High-availability](#)

[Huge pages](#)

[Add sidecar containers](#)

[Restart or pause the cluster](#)

[Monitor the database with PMM](#)

[Upgrade](#)

[About upgrades](#)

[Upgrade the Operator](#)

[Upgrade the database](#)

[About database upgrades](#)

[Minor version upgrade of Percona Distribution for PostgreSQL](#)

[Major version upgrade](#)

[Upgrade PostgreSQL extensions](#)

[Upgrade on Percona Operator for PostgreSQL on OpenShift](#)

[Migrate from Crunchy Operator for PostgreSQL](#)

[About migration from Crunchy to Percona Operator for PostgreSQL](#)

[Migrate using a standby cluster](#)

[Migrate using backup and restore](#)

[Migrate using data volumes](#)

[How-to](#)

[Install the database with customized parameters](#)

[Run Initialization SQL commands at cluster creation time](#)

[Change PostgreSQL primary instance](#)

[How to use private registry](#)

[Manage PostgreSQL extensions](#)

[Provide Percona Operator for PostgreSQL single-namespace and multi-namespace deployment](#)

[Use PostgreSQL tablespaces with Percona Operator for PostgreSQL](#)

[Monitor Kubernetes](#)

[Use PostGIS extension](#)

[Configure DNS suffix for service discovery](#)

[Delete the Operator](#)

[Retrieve Percona certified images](#)

[Troubleshooting](#)

[Troubleshoot Operator installation issues](#)

[Initial troubleshooting](#)

[Check storage](#)

[Exec into the container](#)

[Check the logs](#)

[Manage a database manually](#)

[Reinitialize replicas](#)

[Reference](#)

[Custom Resource options](#)

[Backup resource options](#)

[Restore resource options](#)

[Secrets options](#)

[Upgrade resource options](#)

[Immutable options](#)

[Percona certified images](#)

[Versions compatibility](#)

[About documentation](#)

[Legal](#)

[Copyright and licensing information](#)

[Trademark policy](#)

[Release Notes](#)

[Release notes index](#)

[Percona Operator for PostgreSQL 3.0.0 \(2026-05-22\)](#)

[Percona Operator for PostgreSQL 2.9.0 \(2026-04-01\)](#)

[Percona Operator for PostgreSQL 2.8.2 \(2025-12-25\)](#)
[Percona Operator for PostgreSQL 2.8.1 \(2025-12-16\)](#)
[Percona Operator for PostgreSQL 2.8.0 \(2025-11-13\)](#)
[Percona Operator for PostgreSQL 2.7.0 \(2025-07-18\)](#)
[Percona Operator for PostgreSQL 2.6.0 \(2025-03-17\)](#)
[Percona Operator for PostgreSQL 2.5.1 \(2025-03-03\)](#)
[Percona Operator for PostgreSQL 2.5.0 \(2024-10-08\)](#)
[Percona Operator for PostgreSQL 2.4.1 \(2024-08-06\)](#)
[Percona Operator for PostgreSQL 2.4.0 \(2024-06-24\)](#)
[Percona Operator for PostgreSQL 2.3.1 \(2024-01-23\)](#)
[Percona Operator for PostgreSQL 2.3.0 \(2023-12-21\)](#)
[Percona Operator for PostgreSQL 2.2.0 \(2023-06-30\)](#)
[Percona Operator for PostgreSQL 2.1.0 Tech preview \(2023-05-04\)](#)
[Percona Operator for PostgreSQL 2.0.0 Tech preview \(2022-12-30\)](#)

[Home](#)

[Understand the Operator](#)

[Kubernetes Operator concepts](#)
[Features and capabilities](#)
[How the Operator works](#)
[Cluster architecture](#)
[Comparison with other solutions](#)
[Known limitations](#)

[Get help from Percona](#)

[Quickstart guide](#)

[Overview](#)
[System requirements](#)
[1 Quick install](#)
[With kubectl](#)
[With Helm](#)
[2 Connect to PostgreSQL](#)
[3 Insert data](#)
[4 Make a backup](#)
[5 Monitor the database with PMM](#)
[What's next](#)

[Installation](#)

[Install on Minikube](#)

[Install with OpenEverest](#)

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

[Install on Amazon Elastic Kubernetes Service \(AWS EKS\)](#)

[Install on Microsoft Azure Kubernetes Service \(AKS\)](#)

[Install on OpenShift](#)

[Generic Kubernetes installation](#)

[Configuration](#)

[Application and system users](#)

[LDAP authentication](#)

[About LDAP authentication](#)

[Configure LDAP authentication](#)

[Exposing the cluster](#)

[Changing PostgreSQL options](#)

[Anti-affinity and tolerations](#)

[Labels and annotations](#)

[Transport encryption \(TLS/SSL\)](#)

[About TLS/SSL](#)

[Configure TLS/SSL with the Operator using cert-manager](#)

[Migrate from Operator-generated certificates to cert-manager](#)

[Migrate from cert-manager to custom TLS certificates](#)

[Generate certificates manually](#)

[Update TLS/SSL certificates](#)

[Check TLS communication](#)

[Telemetry](#)

[Configure concurrency for a cluster reconciliation](#)

[Environment variables](#)

[About environment variables](#)

[Operator environment variables](#)

[Cluster component environment variables](#)

[Management](#)

[Back up and restore](#)

[About backups](#)

[Configure storage for backups](#)

[Overview](#)

[Amazon S3 storage](#)

[Google Cloud Storage](#)

[Microsoft Azure Blob Storage](#)

[Persistent Volume](#)

[Make a backup](#)

[Scheduled backup](#)

[On-demand backup](#)

[Restore from a backup](#)

[Restore options](#)

[To the same cluster \(in-place restore\)](#)

[To a new cluster \(cluster clone\)](#)

[PVC snapshots](#)

[About PVC snapshots](#)

[Configure PVC snapshots](#)

[Configure PVC snapshots on EKS](#)

[Use PVC snapshots](#)

[Backup encryption](#)

[Speed up backups](#)

[Backup retention](#)

[Delete the unneeded backup](#)

[Disable backups](#)

[Deploy a standby cluster for Disaster Recovery](#)

[Introduction](#)

[Deploy standby cluster based on backups](#)

[Deploy standby cluster based on streaming replication](#)

[Failover](#)

[Scale your cluster](#)

[High-availability](#)

[Huge pages](#)

[Add sidecar containers](#)

[Restart or pause the cluster](#)

[Monitor the database with PMM](#)

[Upgrade](#)

[About upgrades](#)

[Upgrade the Operator](#)

[Upgrade the database](#)

[About database upgrades](#)

[Minor version upgrade of Percona Distribution for PostgreSQL](#)

[Major version upgrade](#)

[Upgrade PostgreSQL extensions](#)

[Upgrade on Percona Operator for PostgreSQL on OpenShift](#)

[Migrate from Crunchy Operator for PostgreSQL](#)

[About migration from Crunchy to Percona Operator for PostgreSQL](#)

[Migrate using a standby cluster](#)

[Migrate using backup and restore](#)

[Migrate using data volumes](#)

[How-to](#)

[Install the database with customized parameters](#)

[Run Initialization SQL commands at cluster creation time](#)

[Change PostgreSQL primary instance](#)

[How to use private registry](#)

[Manage PostgreSQL extensions](#)

[Provide Percona Operator for PostgreSQL single-namespace and multi-namespace deployment](#)

[Use PostgreSQL tablespaces with Percona Operator for PostgreSQL](#)

[Monitor Kubernetes](#)

[Use PostGIS extension](#)

[Configure DNS suffix for service discovery](#)

[Delete the Operator](#)

[Retrieve Percona certified images](#)

[Troubleshooting](#)

[Troubleshoot Operator installation issues](#)

[Initial troubleshooting](#)

[Check storage](#)

[Exec into the container](#)

[Check the logs](#)

[Manage a database manually](#)

[Reinitialize replicas](#)

[Reference](#)

[Custom Resource options](#)

[Backup resource options](#)

[Restore resource options](#)

[Secrets options](#)

[Upgrade resource options](#)

[Immutable options](#)

[Percona certified images](#)

[Versions compatibility](#)

[About documentation](#)

[Legal](#)

[Copyright and licensing information](#)

[Trademark policy](#)

[Release Notes](#)

[Release notes index](#)

[Percona Operator for PostgreSQL 3.0.0 \(2026-05-22\)](#)

[Percona Operator for PostgreSQL 2.9.0 \(2026-04-01\)](#)

[Percona Operator for PostgreSQL 2.8.2 \(2025-12-25\)](#)

[Percona Operator for PostgreSQL 2.8.1 \(2025-12-16\)](#)

[Percona Operator for PostgreSQL 2.8.0 \(2025-11-13\)](#)

[Percona Operator for PostgreSQL 2.7.0 \(2025-07-18\)](#)

[Percona Operator for PostgreSQL 2.6.0 \(2025-03-17\)](#)

[Percona Operator for PostgreSQL 2.5.1 \(2025-03-03\)](#)

[Percona Operator for PostgreSQL 2.5.0 \(2024-10-08\)](#)

[Percona Operator for PostgreSQL 2.4.1 \(2024-08-06\)](#)

[Percona Operator for PostgreSQL 2.4.0 \(2024-06-24\)](#)

[Percona Operator for PostgreSQL 2.3.1 \(2024-01-23\)](#)

[Percona Operator for PostgreSQL 2.3.0 \(2023-12-21\)](#)

[Percona Operator for PostgreSQL 2.2.0 \(2023-06-30\)](#)

[Percona Operator for PostgreSQL 2.1.0 Tech preview \(2023-05-04\)](#)

[Percona Operator for PostgreSQL 2.0.0 Tech preview \(2022-12-30\)](#)

Percona Operator for PostgreSQL documentation

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

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

This is the documentation for the latest release, **3.0.0** ([Release Notes](#)).

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

Installation guides

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

[Quickstart guides →](#)

Security and encryption

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

[Security measures →](#)

Backup management

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

[Backup management →](#)

Troubleshooting

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

[Diagnostics →](#)

Understand the Operator

Kubernetes Operator concepts

If you already run workloads on Kubernetes, you are used to declaring *desired state* in YAML and letting controllers create Pods, Services, and other objects. A **Kubernetes Operator** applies the same idea to a whole application stack—such as a database cluster—so you do not have to assemble every low-level object by hand.

This page explains what Operators are in general and how **Percona Operator for PostgreSQL** fits in that model. For how reconciliation and Custom Resources work in practice, see [How the Operator works](#). To learn more what the Operator manages, see [Architecture](#).

What is a Kubernetes Operator?

An Operator is a way to **package, deploy, and manage** a complex application like databases on Kubernetes. It extends the Kubernetes API with custom resources so you can create and update instances of that application using the same `kubectl` and GitOps workflows you use elsewhere.

At its core, an Operator is a **custom controller**. It watches [custom resources](#), compares what *should* exist (desired state) with what *does* exist (current state), and takes steps to bring them in line. That pattern is often called a **control loop** or **reconciliation loop**.

Percona Operator for PostgreSQL was originally built upon patterns and components from [CrunchyData's PostgreSQL Operator](#) [↗](#), benefiting from industry best practices for PostgreSQL management on Kubernetes. Since version 3.0.0, Percona Operator for PostgreSQL has evolved into an independent software solution, continuing to develop the Operator as open source software alongside the database and tooling stack. The Operator leverages Percona's expertise in PostgreSQL management, delivering a solution that addresses the unique challenges of cloud-native database operations and lifecycle management.

Percona Operator for PostgreSQL PostgreSQL lifecycle management by automating tasks such as initialization, scaling, backups, updates, and disaster recovery.

Custom Resources explained

How Percona Operator for PostgreSQL works in Kubernetes is defined by the relationship between these core components:

- Custom Resource Definition
- Custom Resource
- Operator Deployment

Custom Resource Definition (CRD)

A **Custom Resource Definition** is a schema that defines a new type of resource in Kubernetes. It tells Kubernetes:

- What fields the custom resource will have
- What types of values those fields can contain
- How the resource should be validated

For example, a Custom Resource Definition for a database cluster called `PostgreSQLCluster` has fields like `replicas`, `storageSize`, `postgresVersion`, etc.

Custom Resource (CR)

A **Custom Resource** is an instance of a CRD. It represents a specific deployment of your application. When you create a Custom Resource, you're essentially declaring your desired state for that application.

For instance, you might create a Custom Resource named `my-postgres-cluster` that specifies that you want a PostgreSQL cluster with 3 replicas, 100GB of storage, and PostgreSQL version 18.

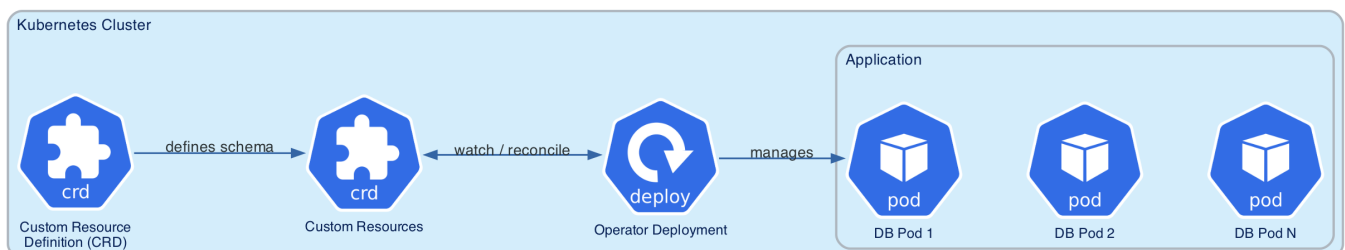
Operator Deployment

The **Operator** itself is deployed as a standard Kubernetes Deployment. It runs as a Pod (or set of Pods) in your cluster and contains the controller logic that:

1. Watches for Custom Resources of the type defined by the CRD
2. Reads the desired state from the Custom Resource
3. Compares it with the current state of the cluster
4. Takes actions to reconcile any differences - to bring the current cluster state to the desired state defined in the Custom Resource

In short: the CRD extends the API with the new resources and defines the *shape* of the resource, the CR is *your* declaration of the resource, and the Operator is the automation that *implements* it.

The following is the components workflow:



1. **Install CRDs** — The new API kinds appear in the cluster.
2. **Install Operator Deployment:** The Operator Pod is deployed, and it starts watching for resources.
3. **Create Custom Resource:** You create and apply a Custom Resource YAML file that describes your desired cluster state
4. **Reconcile:** The Operator detects the new Custom Resource, reads its specification, and creates the necessary Kubernetes resources (StatefulSets, Services, ConfigMaps, Secrets, etc.) to bring the application (PostgreSQL in our case) into existence.
5. **Monitor continuously:** The Operator continuously monitors both the Custom Resource and the actual cluster state, making adjustments whenever they diverge.

Why use an Operator for a database?

The Operator is a game-changer for database management on Kubernetes: it frees administrators and infrastructure teams from tedious, error-prone operational work, allowing them to focus on more important and interesting tasks rather than manual maintenance.

Traditionally, deploying PostgreSQL in Kubernetes means you must create, wire together, and continuously maintain many objects like StatefulSets, Services, PVCs, ConfigMaps, Secrets, backup Jobs, monitoring hooks, and more. Every change or upgrade requires manual intervention and careful coordination.

With an Operator, all you do is describe your desired cluster in a Custom Resource. The Operator reads it and does the works for you:

- It automatically translates your intent into concrete, production-ready objects
- Integrates trusted open source tools like Patroni for high availability pgBackRest for backup/restore operations, and pgBouncer for connection pooling
- Keeps your cluster healthy and up to date, even as requirements change

Day-to-day, the Operator automates the most challenging and repetitive database tasks: seamless failovers, backup management, rolling upgrades, scaling, and self-healing.

By trusting the Operator to manage your database infrastructure, your team can focus on building features and improving your applications rather than deal with unexpected outages.

Next steps

[Features and capabilities](#)

Features and capabilities

Percona Operator for PostgreSQL is a Kubernetes-native controller that automatically manages the full lifecycle of [Percona Distribution for PostgreSQL](#) clusters. The Operator offloads your teams from manual day-to-day database management operations empowering them to focus on tasks that matter instead. To learn how the Operator fits into Kubernetes, see [Kubernetes Operator concepts](#).

Here's what Percona Operator for PostgreSQL brings to your infrastructure:

High availability and failover

Run PostgreSQL with confidence: [Patroni](#) provides automatic leader election, failover, and coordination with Distributed Configuration Store, so your cluster stays available through node and Pod failures. For architecture details, see [Cluster architecture](#).

- **Automatic failover** — Patroni manages leader election and failover to ensure the cluster always has a healthy primary. See [High availability](#).
- **Zero data loss failover** — WAL-based replication limits data loss during failover; synchronous replication is available when you need stronger guarantees.
- **Health monitoring** — Continuous health checks trigger failover when PostgreSQL is not ready to serve traffic.
- **Manual switchover** — [Promote a replica to primary](#) in a controlled way for maintenance.

Automated backup and restore flows

Safeguard your data at any scale: the Operator automates backups and restores using [pgBackRest](#), a robust open source solution trusted for PostgreSQL in production. Read [About backups](#) for the full workflow.

Also, leverage Kubernetes [PersistentVolumeClaim snapshots](#) for rapid, consistent backup and restore operations. It is especially valuable for large database clusters.

- **Full, incremental, and differential backups** — Select the backup strategy that matches your recovery objectives and storage requirements.
- **Point-in-time recovery (PITR)** — Achieve low Recovery Point Objectives (RPO) by [restoring to any specific time](#) using WAL archives.
- **Scheduled backups** — Automate backups on your chosen [schedule with cron-like expressions](#).
- **Flexible storage** — Store backups in S3-compatible object storage or on local PersistentVolumes for hybrid strategies.
- **PVC snapshot support** — Boost backup and restore performance for large datasets with a point-in-time snapshot of your data volume.

- **Encryption** — [Secure backups at rest](#) where your storage backends and configuration allow it.
- **Retention** — Manage backup lifecycle and [automate old backup cleanup](#) to prevent storage sprawl.

Connection pooling with pgBouncer

Reduce connection churn and spread read load without extra operational burden.

- **Efficient pooling** — Lower PostgreSQL connection overhead by pooling client connections
- **Transaction-level pooling** — Manage connections at the transaction level efficiently
- **Read balancing** — Distribute read queries across replicas where configured
- **High availability** — Replica pgBouncer instances provide high availability
- **Integrated lifecycle** — Automatically configured and managed by the Operator

Automated scaling and resource management

Scale your cluster up or down to match demand while keeping changes declarative.

- **Declarative clusters** — Describe desired cluster state in YAML; the Operator automatically reconciles Kubernetes resources to match.
- **Replica scaling** — [Adjust replica count](#) in the Custom Resource to scale horizontally.
- **Dynamic configuration** — [Update PostgreSQL parameters](#) without a full cluster restart.
- **Self-healing** — The Operator automatically detects and recovers from Pod crashes, node issues, and common network problems.
- **Rolling updates** — Apply configuration and image updates with controlled rollouts.
- **Storage expansion** — Automatically [increase storage size](#) for PostgreSQL instances when supported by your environment and configuration.

PostgreSQL-specific features

Use PostgreSQL capabilities that operators expect in production.

- **WAL storage** — Optional dedicated volumes for Write-Ahead Logs when you want to separate I/O.
- **Tablespaces** — Custom [tablespaces](#) with dedicated storage.
- **Extensions** — Built-in support for extensions such as pg_stat_monitor, pgAudit, set_user, wal2json, plus ability to extend PostgreSQL with [custom extensions](#).
- **Users and databases** — Automatically create users, databases, and manage credentials.
- **Init SQL** — Execute [custom SQL scripts during cluster initialization](#).

Standby clusters for disaster recovery

Leverage disaster-recovery topologies that fit your RTO and RPO.

- **Backups or streaming** — Deploy your standby cluster based on backups or streaming replication, depending on your architecture
- **Cross-namespace or cross-cluster** — Primary and standby clusters can run in different namespaces or Kubernetes clusters
- **Promotion** — Promote a standby to primary when you need to recover from an outage or drill a failover

Security and compliance

Keep traffic and data protected with encryption and flexible TLS workflows.

- **TLS for connections** — Encrypt client traffic and traffic between cluster components
- **Certificates** — Comply with your security policy via [custom certificates](#) or automated certificate generation [with cert-manager](#) with configurable lifecycle management.

Monitoring and observability

Understand performance and troubleshoot faster with metrics and optional Percona tooling.

- **PMM integration** — Connect the cluster to [Percona Monitoring and Management \(PMM\)](#) [↗](#) for dashboards and alerting.
- **pg_stat_monitor** — Get query performance insights with fingerprinting when you enable the extension.
- **Broad metrics** — Track connection counts, transaction rates, cache hit ratios, replication lag, and more.
- **Query analytics** — Deeper query analysis in PMM. See [Query Analytics](#) [↗](#) in the PMM documentation.

Operator capabilities

Operate at the scale of your platform with flexible reconciliation scope.

- **Selective namespaces** — Reconcile clusters in a single namespace or multi-namespace mode. See [cluster-wide deployment](#) to learn more.
- **Concurrent reconciliation** — Run [concurrent reconciliations](#) to manage many clusters efficiently.

How the Operator works

This page describes how Percona Operator for PostgreSQL plugs into Kubernetes and how it keeps your database cluster in the state you define.

If you are new to the Operator pattern itself, start with [Kubernetes Operators and Percona Operator](#). For the runtime stack (database processes, Patroni, backups), see [Cluster architecture](#).

How the Operator extends Kubernetes API

Standard Kubernetes resources cover Deployments, Services, and many other kinds. The Operator registers additional Custom Resources, defined via [Custom Resource Definitions \(CRDs\)](#) [↗](#).

These Custom Resources are:

- `PerconaPGCluster` for the database cluster
- `PerconaPGBackup` for backups
- `PerconaPGRestore` for restores
- `PerconaPGUpgrade` for major upgrades

The Operator itself runs as a **Deployment** in the Kubernetes cluster. It has controller logic that watches the Custom Resources that the CRDs define. Whenever you create or change a relevant Custom Resource, the Operator's reconciliation loop automatically does the following:

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

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

To learn more about features and capabilities that the Operator brings in, see [Features and capabilities](#).

Next steps

- [Cluster architecture](#)

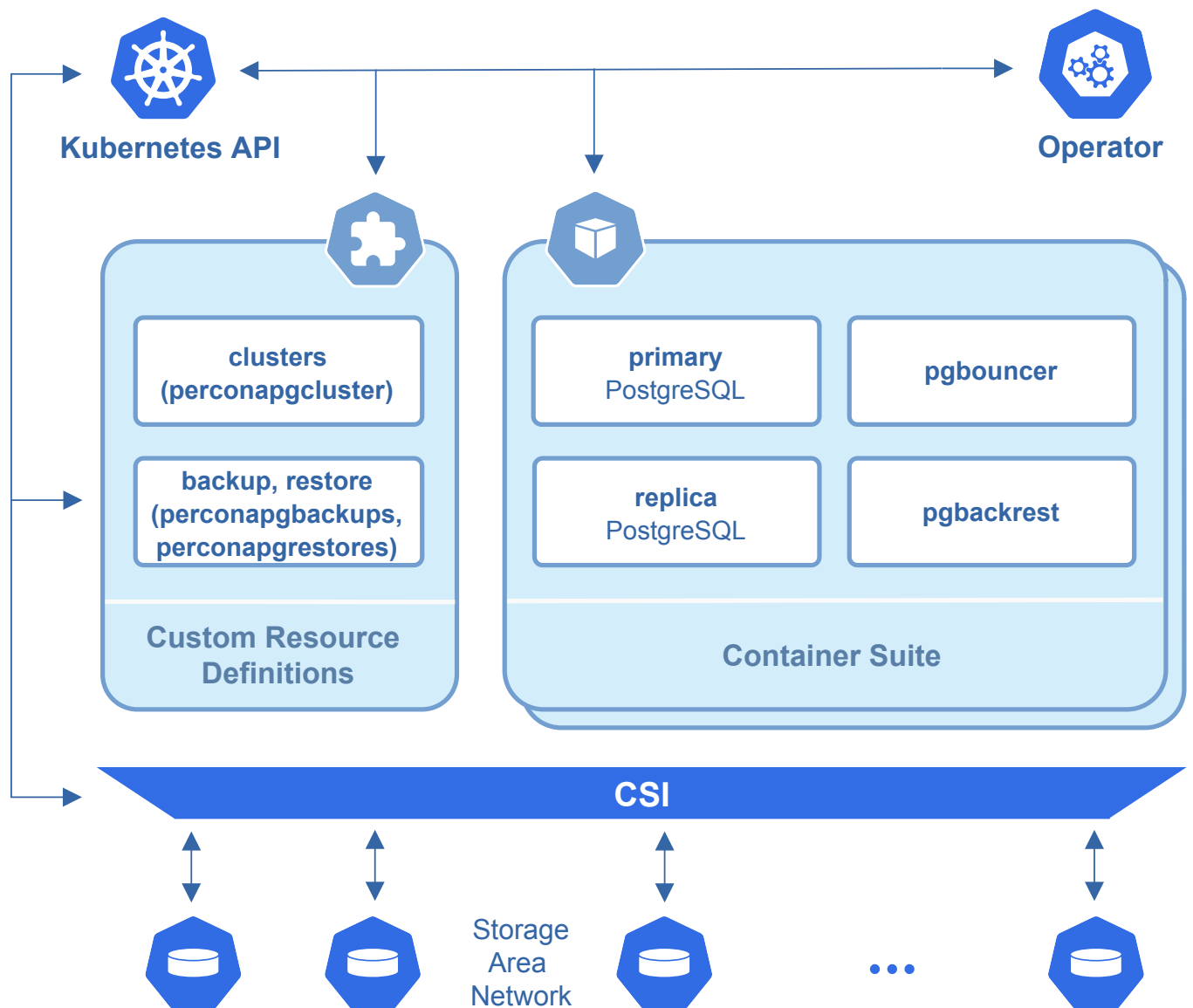
Architecture

This document provides a high-level overview of Percona Operator for PostgreSQL architecture, explaining how the various components connect to create a production-ready PostgreSQL cluster on Kubernetes. See also [How the Operator works](#).

Components

The Operator components are the following:

- [Percona Distribution for PostgreSQL](#)  - a suite of open source software, tools and services required to deploy and maintain a reliable production cluster for PostgreSQL. It includes the set of extensions such as [pgAudit](#)  for audit logging, [pg_stat_monitor](#)  for query performance statistics and [additional supplied modules and extensions](#) . It also comes with LLVM library for JIT compilation.
- [Patroni](#)  - a high-availability solution for PostgreSQL that automates replication and **failover**. It maintains the cluster state and coordinates leader election to ensure that a healthy primary node is always available. Patroni simplifies building and operating resilient PostgreSQL clusters by handling node monitoring, failover, and recovery automatically.
- [pgBouncer](#)  is a **lightweight connection pooler** in front of PostgreSQL. It sits between client applications and the database server to manage and reuse connections efficiently. Instead of each client opening its own database connection, pgBouncer maintains a pool of connections and serves them to clients on demand, significantly reducing connection overhead and improving performance, especially for applications with many short-lived or concurrent connections.
- [pgBackRest](#)  is a backup and restore tool. It handles **full, incremental, and differential** backups, compression and encryption, parallel processing, and point-in-time recovery using WAL archives.
- **PMM Client for observability** – The PMM Client is an optional, yet valuable, component that you can enable to gain deeper insights into your database performance. When monitoring is [configured](#), the PMM Client is deployed as a sidecar container alongside PostgreSQL Pods, empowering you with detailed monitoring and management capabilities.



How components work together

This workflow shows how cluster components work together:

1. Your **application** connects through a Kubernetes **Service** that routes the traffic to pgBouncer.
2. **pgBouncer** accepts many client connections and forwards them through a smaller set of server connections to PostgreSQL Pods.
3. **PostgreSQL** executes queries. **Writes** go to the **primary**. **Reads** can target the primary or **replicas**.
4. Primary streams WAL to replicas via instance Services
5. Patroni monitors the cluster state and coordinates the leader elections if the primary node fails
6. pgBackRest makes backups according the schedule that you defined or when you manually create a backup object. pgBackRest saves backups to the backup storage your configured. To learn more about backups, their workflow and setup, refer to the [About backups](#)
7. PMM Client collects performance metrics and sends them to the PMM Server for you to see and analyze.

See [Monitor the database with PMM](#) to learn more.

Default cluster configuration

The default Percona Distribution for PostgreSQL configuration includes:

- 3 PostgreSQL servers, one primary and two replicas.
- 3 pgBouncer instances.
- a pgBackRest repository host instance – a dedicated instance in your cluster that stores filesystem backups made with pgBackRest.
- (optional) a PMM client instance - a monitoring and management tool for PostgreSQL that provides a way to monitor your database health and performance. PMM Client is disabled by default and runs as a sidecar container in the database Pods when you [configure monitoring](#)

Primary, replicas, and high availability

Each PostgreSQL cluster has **one primary** instance that accepts read/write transactions. **Replicas** are standbys: they replicate from the primary and typically serve **read-only** traffic (depending on how you expose them).

The Operator provides high availability through multiple layers of protection:

Pod distribution

The Operator uses [node affinity and anti-affinity](#)  to distribute PostgreSQL instances across separate worker nodes when possible. This prevents a single node failure from taking down multiple database instances.

Automatic recovery

If a node fails, Kubernetes automatically reschedules the affected Pod on another healthy node. Patroni handles which PostgreSQL instance is primary and ensures replication continuity. For more on HA behavior and operations, see [High-availability](#).

Storage and persistent volumes

Stateful applications require their data to persist even if Pods are restarted or rescheduled. In Kubernetes, this is achieved through **PersistentVolumeClaims (PVCs)**, which request storage resources. The cluster's CSI driver provisions **PersistentVolumes**, and can **reattach** storage if a Pod moves to another node, thereby ensuring data continuity.

If a node fails, the expectation is that the volume can be mounted elsewhere and the Pod recreated, while Patroni and PostgreSQL recover the database layer. For storage troubleshooting, see [Check storage](#).

Next steps

For a comparison of Percona's approach with other deployment models, see [Comparison with other solutions](#).

Compare various solutions to deploy PostgreSQL in Kubernetes

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

- [Crunchy Data PostgreSQL Operator \(PGO\)](#)
- [CloudNative PG](#), vendor-neutral, originally created by Enterprise DB
- [Stackgres](#) from OnGres
- [Zalando Postgres Operator](#)
- [Percona Operator for PostgreSQL](#)

Generic

Feature/Product	Percona Operator for PostgreSQL	Stackgres	CrunchyData	CloudNativePG	Zalando
Open-source license	Apache 2.0	AGPL 3	Apache 2.0, but images are under Developer Program	Apache 2.0	MIT
PostgreSQL versions	13 - 18	14 - 18	14 - 18	14 - 18	13 - 17
Kubernetes conformance	Various versions are tested	Various versions are tested	Various versions are tested	Various versions are tested	AWS EKS
Web-based GUI	Percona Everest	Admin UI	⊘	⊘	Postgres Operator UI

Maintenance

Feature/Product	Percona Operator for PostgreSQL	Stackgres	CrunchyData	CloudNativePG	Zalando
Operator upgrade	✓	✓	✓	✓	✓

Database upgrade	Automated and safe	Automated and safe	Manual	Manual	Manual
Compute scaling	Horizontal and vertical	Horizontal and vertical	Horizontal and vertical	Horizontal and vertical	Horizontal and vertical
Storage scaling	Manual	Manual	Manual	Manual	Manual, automated for AWS EBS

PostgreSQL topologies

Feature/Product	Percona Operator for PostgreSQL	Stackgres	CrunchyData	CloudNativePG	Zalando
Warm standby	✓	✓	✓	✓	✓
Hot standby	✓	✓	✓	✓	✓
Connection pooling	✓	✓	✓	✓	✓
Delayed replica	✗	✗	✗	✓	✗

Backups

Feature/Product	Percona Operator for PostgreSQL	Stackgres	CrunchyData	CloudNativePG	Zalando
Scheduled backups	✓	✓	✓	✓	✓
WAL archiving	✓	✓	✓	✓	✓
PITR	✓	✓	✓	✓	✓
GCS	✓	✓	✓	✓	✓
S3	✓	✓	✓	✓	✓
Azure	✓	✓	✓	✓	✓

Monitoring

Feature/Product	Percona Operator for PostgreSQL	Stackgres	CrunchyData	CloudNativePG	Zalando
Solution	Percona Monitoring and Management and sidecars	Exposing metrics in Prometheus format	Prometheus stack and pgMonitor	Prometheus metrics & Grafana dashboard	Sidecars

Miscellaneous

Feature/Product	Percona Operator for PostgreSQL	Stackgres	CrunchyData	CloudNativePG	Zalando
Customize PostgreSQL configuration	✓	✓	✓	✓	✓
Sidecar containers for customization	✓	✗	✓	✗	✓
Helm	✓	✓	✓	✓	✓
Transport encryption	✓	✓	✓	✓	✓
Data-at-rest encryption	Through storage class	Through storage class	Through storage class	Through storage class	Through storage class
Create users/roles	✓	✓	✓	✓	limited

Known limitations

This page describes known limitations of Percona Operator for PostgreSQL. Understanding these constraints helps you plan deployments and avoid unexpected behavior.

Replica management

- Delayed replicas are not supported
- The Operator doesn't automatically update Patroni configuration inside running instances when `create_replica_methods` is updated. You must manually run `patronictl reload` on each replica to make Patroni aware of the change. See [Override Patroni configuration](#) for details.

Service control

- By default, stopping a PostgreSQL process in the Operator triggers a Pod restart because Kubernetes treats the stopped process as a failed container.

You can shut down the entire cluster gracefully by [pausing](#) it and resuming it later to restore its operation. The Operator handles this process for you.

To have a per-pod service control for maintenance, debugging, or controlled downtime requires manual operation. Refer to the [Manual management of database clusters](#) chapter for details how to do it.

- Due to security considerations, superusers cannot connect to PostgreSQL via `pgBouncer` by default. They can either connect via the primary service bypassing `pgBouncer`, or you must adjust `pgBouncer` configuration exposing it to superusers. Read more about it in the [Superusers and pgBouncer](#) section.

Backups and restores

- PVC snapshots are in tech preview stage. When you use PVC snapshots as scheduled backups, they are not deleted automatically. You must manually delete them. The retention policy for scheduled backups made from PVC snapshots is planned for future releases.
- When you use the `dataSource` option to restore from a backup on a new cluster, you can use either `dataSource.postgresCluster` or `dataSource.pgbackrest`. You cannot use both options. If both are present, `dataSource.postgresCluster` takes precedence. See [Restore the backup to a new cluster](#) to learn more.

Data management

- Tablespace deletion is not automated; you must remove all objects in all databases using the tablespace

before dropping it. For details, see [Delete existing tablespaces](#).

Node memory overcommit

[PostgreSQL](#)  recommends strict virtual memory allocation by setting `vm.overcommit_memory=2` to ensure predictable memory accounting and reduce the risk of the OOM killer terminating the postmaster. However, Kubernetes does not allow Pods or Operators to configure this setting. The kubelet treats `vm.overcommit_memory` as an unsafe `sysctl` and rejects attempts to set it to `2`. Most Kubernetes node OS images also default to `1`.

To avoid node-level OOM events, consider tuning PostgreSQL memory parameters like `shared_buffers`, `work_mem`, `max_connections`, and so on accordingly.

TLS and cert-manager

For the Operator upgraded to version 2.9.0: If you had cert-manager installed and your cluster used the Operator-generated TLS certificates, the Operator switches TLS management to cert-manager. The Operator creates the cert-manager `Issuer` and `Certificate` resources so that the cert-manager can start start issuing and renewing TLS certificates for your cluster.

However, the Operator hasn't verified if the cert-manager owned the root CA certificate. As a result, the CA Secret remained managed by the Operator's internal PKI and the keys inside it (`root.crt` and `root.key`) differ from what the cert-manager expects (`tls.crt` and `tls.key`). This caused the `Issuer` resource to fail and brought the cluster to the inconsistent state: the CA certificate stayed on the internal PKI, while cert-manager attempted to take ownership of the leaf certificates for the cluster, database instances, pgBouncer and pgBackRest. Those leaf certificates then reported the `Not Ready` status because of the failed `Issuer`.

If you are in this situation, you have these options:

- [Switch the cluster to use only the certificates managed by the cert-manager](#).
- Switch the cluster to use custom certificates only. Refer to the [Migrate from cert-manager to custom TLS certificates](#) for the steps.

Switch to use only the certificates managed by the cert-manager

To restore the cluster from the inconsistent state, reconfigure it to use only the certificates managed by the cert-manager:

1. Delete the CA Secret `<cluster-name>-cluster-ca-cert` that still remains on the Operator's internal PKI:

```
kubectl delete Secret <cluster-name>cluster-ca-cert -n <namespace>
```



This makes the Operator to recreate the Secret through the cert-manager, with the correct keys.

2. Trigger the cluster reconciliation. You can do so by deleting the `pgBouncer` Pod:

```
kubectl delete pod <cluster-name>-pgbouncer-<hash> -n <namespace>
```



The Operator detects the missing root CA Secret, creates cert-manager resources for the CA, and lets cert-manager issue new leaf certificates.

3. Check the Issuer and Certificate status:

```
kubectl get issuers -n <namespace>  
kubectl get certificates -n <namespace>
```



Both Issuer and Certificate resources must report the `Ready` status.

In version 3.0.0, the Operator respects the existing TLS configuration, which resolves this limitation.

Upgrades

- A major database upgrade introduces a downtime because the whole cluster is shut down.
- In clusters with very low write traffic, the first restore after a major upgrade can fail with the `could not find common ancestor of the source and target cluster's timelines` error in `pg_rewind`. In this case you must [reinitialize the replica](#) to ensure it starts properly and starts streaming data from the primary.
- PostGIS extension is not updated automatically after the database update. You must [manually update](#) it for each database where it is enabled.
- `pgAudit` extension is not updated automatically during the major upgrade. You must [drop and recreate](#) it manually for each database where it is installed.
- If you have installed [custom PostgreSQL extensions](#), you need to build and package each custom extension for the new PostgreSQL major version. During the upgrade, the Operator will install extensions into the upgrade container.

Get help from Percona

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

Percona's Community Forum

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

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

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

Percona experts

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

Talk to a Percona Expert

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

Service	Description
24/7 Expert Support	Our dedicated team of database experts is available 24/7 to assist you with any database issues. We provide flexible support plans tailored to your specific needs.
Hands-On Database Management	Our managed services team can take over the day-to-day management of your database infrastructure, freeing up your time to focus on other priorities.
Expert Consulting	Our experienced consultants provide guidance on database topics like architecture design, migration planning, performance optimization, and security best practices.

Comprehensive
Training

Our training programs help your team develop skills to manage databases effectively, offering virtual and in-person courses.

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

Quickstart guide

Overview

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

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

Next steps

[Install the Operator →](#)

System requirements

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

Supported versions

The Operator 3.0.0 is developed, tested and based on:

- PostgreSQL 14.23-1, 15.18-1, 16.14-1, 17.10-1, 18.4-1 as the database. Other versions may also work but have not been tested.
- pgBackRest 2.58.0-2 for backup and recovery
- pgBouncer 1.25.2-1 for connection pooling
- Patroni version 4.1.0 for high-availability
- PostGIS version 3.5.5
- PMM Client version 2.44.1-1 and 3.7.1

Supported platforms

The following platforms were tested and are officially supported by the Operator 3.0.0:

- [Google Kubernetes Engine \(GKE\)](#)  1.33 - 1.35
- [Amazon Elastic Container Service for Kubernetes \(EKS\)](#)  1.33 - 1.35
- [OpenShift](#)  4.18 - 4.21
- [Azure Kubernetes Service \(AKS\)](#)  1.33 - 1.35
- [Minikube](#)  1.38.1 with Kubernetes v1.35.1

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

Huge pages

We strongly recommend [enabling huge pages](#) on worker nodes for better stability and performance.

Installation guidelines

Choose how you wish to install Percona Operator for PostgreSQL:

- [with Helm](#)

- [with `kubect1`](#)
- [on Minikube](#)
- [on Google Kubernetes Engine \(GKE\)](#)
- [on Amazon Elastic Kubernetes Service \(AWS EKS\)](#)
- [on Azure Kubernetes Service \(AKS\)](#)
- [in a general Kubernetes-based environment](#)

1 Quick install

Install Percona Operator for PostgreSQL using kubectl

Percona Operator for PostgreSQL is a custom controller that will manage your PostgreSQL clusters inside Kubernetes. The Operator will automatically deploy, maintain, and monitor PostgreSQL databases, so you don't have to manage these tasks manually.

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



Install with kubectl ↓



Install with Helm →

What you will install

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

The default Percona Distribution for PostgreSQL configuration includes:

- 3 PostgreSQL servers, one Primary and two replicas.
- 3 pgBouncer instances - a lightweight connection pooler for PostgreSQL that sits between client applications and the database server to manage and reuse connections efficiently.
- a pgBackRest repository host instance – a dedicated instance in your cluster that stores filesystem backups made with pgBackRest - a backup and restore utility.
- a PMM client instance - a monitoring and management tool for PostgreSQL that provides a way to monitor and manage your database. It runs as a sidecar container in the database Pods.

Read more about the default components in [Architecture](#).

Prerequisites

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

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



See also

- [Set up Minikube](#)
- [Create and configure the GKE cluster](#)
- [Set up Amazon Elastic Kubernetes Service](#)
- [Create and configure the AKS cluster](#)

Procedure

Here's a sequence of steps to follow:

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

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



Expected output



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

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

```
kubectl apply --server-side -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/v3.0.0/deploy/bundle.yaml -n postgres-operator
```



Expected output



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

- 3 Deploy Percona Distribution for PostgreSQL cluster:

```
kubectl apply -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/v3.0.0/deploy/cr.yaml -n postgres-operator
```



Expected output



- 4 Check the Operator and replica set Pods status.

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



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



Expected output



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

Next steps



[Connect to PostgreSQL →](#)

Install Percona Operator for PostgreSQL using Helm

[Helm](#) is the package manager for Kubernetes. A Helm [chart](#) is a package that contains all the necessary files to deploy resources or an application to a Kubernetes cluster.

Helm charts for Percona Operator for PostgreSQL include:

- [Percona Operator for PostgreSQL Deployment](#)
- [Percona Distribution for PostgreSQL](#)

You need to install both of them. First you install the Operator Deployment and then you use that to install Percona Distribution for PostgreSQL cluster.

This guide walks you through installing Percona Operator for PostgreSQL with default parameters and names.

To install the Operator with custom parameters or custom resource names, refer to [Install Percona Operator for PostgreSQL with customized parameters](#).

Prerequisites

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

1. [Helm v3](#). Run `helm version` to check the version
2. [kubectl](#) command line utility.
3. A Kubernetes environment. You can deploy it locally on [Minikube](#) for testing purposes or using any cloud provider of your choice. Check the list of our [officially supported platforms](#).

See also

- [Set up Minikube](#)
- [Create and configure the GKE cluster](#)
- [Set up Amazon Elastic Kubernetes Service](#)

4. Privileges to create Custom Resource Definitions (CRDs), RBAC resources, and deploy the Operator

Installation

Here's a sequence of steps to follow:

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

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



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

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



- 3 Install the Percona Operator for PostgreSQL:

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



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

- 4 Install Percona Distribution for PostgreSQL:

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



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

- 5 Check the Operator and replica set Pods status.

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



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



Expected output



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

Next steps

[Connect to PostgreSQL](#) →

2 Connect to the PostgreSQL cluster

When the [installation](#) is done, we can connect to the cluster.

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

The `pgBouncer` URI is stored in the [Secret](#) object, which the Operator generates during the installation.

To connect to PostgreSQL, do the following:

1 List the Secrets objects

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

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

 **via kubectl**

```
cluster1-pguser-cluster1
```



via Helm

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

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

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

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

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

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

```
kubect1 run -i --rm --tty pg-client --image=percona/percona-distribution-  
postgresql:18.4-1 --restart=Never -- psql $PGBOUNCER_URI
```



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



Expected output



Congratulations! You have connected to your PostgreSQL cluster.

Next steps

 [Insert testing data →](#)

3 Insert sample data

The next step after [connecting to the cluster](#) is to insert some sample data to PostgreSQL.

When you start a PostgreSQL container and connect to the database, a user is created with the username that matches the name of your cluster. Also, a database and a schema named after the name of this user are created so that you can [create a table](#) right away.

Create a schema (for Operator version earlier than 2.6.0)

In Operator versions earlier than 2.6.0, you must create a new schema to insert the data. This is because your user cannot access the default schema called `public` due to PostgreSQL restrictions (introduced starting with PostgreSQL 15).

A schema stores database objects like tables, views, indexes and allows organizing them into logical groups.

Use the following statement to create a schema

```
CREATE SCHEMA demo;
```



Create a table

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

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

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





Tip

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

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



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

Insert the data

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

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

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



This command does the following:

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

Now your cluster has some data in it.

Next steps

 [Make a backup →](#)

4 Make a backup

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

Note

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

Considerations and prerequisites

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

If you don't have access to AWS, you can use any S3-compatible storage like [MinIO](#). Check the list of [supported storages](#). Find the [storage configuration guides](#) for each backend.

- The Operator uses the [pgBackRest](#) tool to make backups. `pgBackRest` stores the backups and archives WAL segments in repositories. The Operator has up to four `pgBackRest` repositories named `repo1`, `repo2`, `repo3` and `repo4`. In this tutorial we use `repo2` for backups.
- Also, we will use some files from the Operator repository for setting up backups. So, clone the `percona-postgresql-operator` repository:

```
git clone -b v3.0.0 https://github.com/percona/percona-postgresql-operator
cd percona-postgresql-operator
```

Note

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

Configure backup storage

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



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



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

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

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

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

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

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

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

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

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

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

Make a backup

For manual backups, you need a backup configuration file.

- 1 Edit the example backup configuration file [deploy/backup.yaml [↗](https://raw.githubusercontent.com/percona/percona-postgresql-operator/v3.0.0/deploy/backup.yaml)]
(<https://raw.githubusercontent.com/percona/percona-postgresql-operator/v3.0.0/deploy/backup.yaml>).
Specify your cluster name and the `repo` name.

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

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

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

- 3 To make a backup takes a while. Track the backup progress:

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



Expected output



Congratulations! You have made the first backup manually. Want to learn more about backups? See the [Backup and restore section](#) for details like types, retention and how to [automatically make backups according to the schedule](#). When you need to restore, see [Restore options](#) for in-place restore and cluster cloning.

Next steps



Monitor the database →

5 Monitor the database

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

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

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

PMM2 has reached the end-of-life stage and is deprecated. We recommend to use the latest PMM 3.

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

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

Considerations

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

Install PMM Server

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

Install PMM Client

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

Configure authentication

PMM3

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

Warning

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

PMM2 (deprecated)

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

From PMM UI

[Generate the PMM API key](#)

From command line

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

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

Warning

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

Create a secret

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

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

PMM 3

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

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



PMM 2 (deprecated)

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

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



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

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



Expected output



Deploy a PMM Client

1. Update the `pmm` section in the [deploy/cr.yaml](#)  file.

- Set `pmm.enabled = true`.
- Specify your PMM Server hostname / an IP address for the `pmm.serverHost` option. The PMM Server IP address should be resolvable and reachable from within your cluster.
- Specify the name of the Secret object that you created earlier

```
pmm:
  enabled: true
  image: percona/pmm-client:3.7.1
#   imagePullPolicy: IfNotPresent
  secret: cluster1-pmm-secret
  serverHost: monitoring-service
```

2. Update the cluster

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

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

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

Update the secrets file

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

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



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



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

For example, to set the new service account token in the `my-cluster-name-secrets` object, do the following:



```
kubectl patch secret/cluster1-pmm-secret -p '{"data":{"PMM_SERVER_TOKEN": '$(echo -n <new-token> | base64 --wrap=0)'}}'
```



```
kubectl patch secret/cluster1-pmm-secret -p '{"data":{"PMM_SERVER_TOKEN": '$(echo -n <new-token> | base64)'}}'
```

Check the metrics

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

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

Next steps

[What's next →](#)

What's next?

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

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

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

Installation

Install Percona Distribution for PostgreSQL on Minikube

Installing the Percona Operator for PostgreSQL on [Minikube](#)  is the easiest way to try it locally without a cloud provider.

Minikube runs Kubernetes on GNU/Linux, Windows, or macOS system using a system-wide hypervisor, such as VirtualBox, KVM/QEMU, VMware Fusion or Hyper-V. Using it is a popular way to test Kubernetes application locally prior to deploying it on a cloud.

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

Set up Minikube

- 1 [Install Minikube](#) , using a way recommended for your system. This includes the installation of the following three components:
 - 1 kubectl tool,
 - 2 a hypervisor, if it is not already installed,
 - 3 actual minikube package
- 2 After the installation, initialize and start the Kubernetes cluster. The parameters we pass for the following command increase the virtual machine limits for the CPU cores, memory, and disk, to ensure stable work of the Operator:

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


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

Deploy the Percona Operator for PostgreSQL

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

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

 Expected output

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

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

```
kubectl apply --server-side -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/v3.0.0/deploy/bundle.yaml -n postgres-operator
```

 Expected output

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

- 3 Deploy Percona Distribution for PostgreSQL:

```
kubectl apply -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/v3.0.0/deploy/cr.yaml -n postgres-operator
```

 Expected output

Note

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

```
git clone -b v3.0.0 https://github.com/percona/percona-postgresql-operator
```

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

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

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

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

 Expected output

Verify the Percona Distribution for PostgreSQL cluster operation

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

When the Operator deploys a database cluster, it generates several [Secrets](#). Among them there is the Secret with the credentials of the default PostgreSQL user. This default user has the same username as the cluster name.

- 1 Use `kubectl get secrets -n <namespace>` command to see the list of Secrets objects. The Secrets object you are interested in is named in the format `<cluster_name>-pguser-<cluster_name>` (where the `<cluster_name>` is the [name of your Percona Distribution for PostgreSQL Cluster](#)). For example, if your cluster name is `cluster1`, the Secret name will be `cluster1-pguser-cluster1`.
- 2 Use the following command to get the password of this user. Replace the `<cluster_name>` and `<namespace>` placeholders with your values:

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

- 3 To connect to PostgreSQL, you will use the `pgbouncer` service as the entry point to your cluster. To find this service, use the following command:

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

Look for the service named `<cluster-name>-pgbouncer`.

Sample output



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

```
kubectl run -n <namespace> -i --rm --tty pg-client --image=percona/percona-  
distribution-postgresql:18.4-1 --restart=Never -- bash -il
```

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

- 5 Run a container with `psql` tool and connect its console output to your terminal. Substitute the `<namespace>` placeholder with your value in the following command to connect as a `cluster1` user to the `cluster1` database via the PostgreSQL interactive terminal.

```
[postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer.  
<namespace>.svc.cluster.local -p 5432 -U cluster1 cluster1
```

 Sample output



Delete the cluster

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

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

- 1 Stop the Minikube cluster:

```
minikube stop
```



- 2 Delete the cluster

```
minikube delete
```



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

Install Percona Distribution for PostgreSQL cluster using OpenEverest

[OpenEverest](#)  is an open source cloud-native database platform and a [CNCFSandbox project](#)  that helps developers deploy code faster, scale deployments rapidly, and reduce database administration overhead while regaining control over their data, database configuration, and DBaaS costs.

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

You can try it in action by [Installing OpenEverest](#)  and [managing your first cluster](#) .

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

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

Prerequisites

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

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

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

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

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



Create and configure the GKE cluster

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

```
gcloud container clusters create cluster-1 --project <project ID> --zone us-central1
a --cluster-version 1.35 --machine-type n1-standard-4 --num-nodes=3
```



Note

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

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



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



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

☐	☒	cluster1	eu-west-1	3	12	45 GB	—	⋮
								Edit Connect Delete

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



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 v3.0.0 https://github.com/percona/percona-postgresql-operator  
cd percona-postgresql-operator
```

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

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



Expected output





Note

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

- 3 Deploy the Operator [using](#) [the following command](#):

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



Expected output



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

- 4 Deploy Percona Distribution for PostgreSQL:

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



Expected output



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

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



Expected output



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



Verifying the cluster operation

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

When the Operator deploys a database cluster, it generates several [Secrets](#) [the following](#). Among them there is the Secret with the credentials of the default PostgreSQL user. This default user has the same username as the cluster name.

- 1 Use `kubectl get secrets -n <namespace>` command to see the list of Secrets objects. The Secrets object you are interested in is named in the format `<cluster_name>-pguser-<cluster_name>` (where the `<cluster_name>` is the [name of your Percona Distribution for PostgreSQL Cluster](#)). For example, if your cluster name is `cluster1`, the Secret name will be `cluster1-pguser-cluster1`.

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

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

- 3 To connect to PostgreSQL, you will use the `pgbouncer` service as the entry point to your cluster. To find this service, use the following command:

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

Look for the service named `<cluster-name>-pgbouncer`.

 Sample output



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

```
kubectl run -n <namespace> -i --rm --tty pg-client --image=percona/percona-  
distribution-postgresql:18.4-1 --restart=Never -- bash -il
```

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

- 5 Run a container with `psql` tool and connect its console output to your terminal. Substitute the `<namespace>` placeholder with your value in the following command to connect as a `cluster1` user to the `cluster1` database via the PostgreSQL interactive terminal.

```
[postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer.  
<namespace>.svc.cluster.local -p 5432 -U cluster1 cluster1
```

 Sample output



Removing the cluster

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

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

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

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



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



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



The cluster deletion may take time.



Warning

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

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

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

Prerequisites

Software installation

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

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

Also, you need to configure AWS CLI with your credentials according to the [official guide](#) .

Creating the EKS cluster

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

- name of your EKS cluster,
- AWS region in which you wish to deploy your cluster,
- the amount of nodes you would like to have,
- the desired ratio between [on-demand](#)  and [spot](#)  instances in the total number of nodes.

Note

[spot](#)  instances are not recommended for production environment, but may be useful e.g. for testing purposes.

After you have settled all the needed details, create your EKS cluster [following the official cluster creation instructions](#) .

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

 Note

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

Install the Operator and Percona Distribution for PostgreSQL

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

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

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

 Expected output

 Note

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

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

```
kubectl apply --server-side -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/v3.0.0/deploy/bundle.yaml -n postgres-operator
```

 Expected output

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

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

```
kubectl apply -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/v3.0.0/deploy/cr.yaml -n postgres-operator
```



Expected output



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 v3.0.0 https://github.com/percona/percona-postgresql-operator
```



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

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



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

```
kubectl get pg
```



Expected output



Verifying the cluster operation

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

When the Operator deploys a database cluster, it generates several [Secrets](#). Among them there is the Secret with the credentials of the default PostgreSQL user. This default user has the same username as the cluster name.

- 1 Use `kubectl get secrets -n <namespace>` command to see the list of Secrets objects. The Secrets object you are interested in is named in the format `<cluster_name>-pguser-<cluster_name>` (where the `<cluster_name>` is the [name of your Percona Distribution for PostgreSQL Cluster](#)). For example, if your cluster name is `cluster1`, the Secret name will be `cluster1-pguser-cluster1`.
- 2 Use the following command to get the password of this user. Replace the `<cluster_name>` and `<namespace>` placeholders with your values:

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



- 3 To connect to PostgreSQL, you will use the `pgbouncer` service as the entry point to your cluster. To find this service, use the following command:

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

Look for the service named `<cluster-name>-pgbouncer`.

 Sample output



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

```
kubectl run -n <namespace> -i --rm --tty pg-client --image=percona/percona-distribution-postgresql:18.4-1 --restart=Never -- bash -il
```

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

- 5 Run a container with `psql` tool and connect its console output to your terminal. Substitute the `<namespace>` placeholder with your value in the following command to connect as a `cluster1` user to the `cluster1` database via the PostgreSQL interactive terminal.

```
[postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer.<namespace>.svc.cluster.local -p 5432 -U cluster1 cluster1
```

 Sample output



Removing the cluster

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

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

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

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

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

The cluster deletion may take time.



Warning

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

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

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

Prerequisites

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

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

Also, you need to sign in with Azure CLI using your credentials according to the [official guide](#) .

Create and configure the AKS cluster

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

- name of your AKS cluster,
- an [Azure resource group](#) , in which resources of your cluster will be deployed and managed.
- the amount of nodes you would like to have.

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

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

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

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

Now you should configure the command-line access to your newly created cluster to make `kubect1` be able to use it.

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



Install the Operator and deploy your PostgreSQL cluster

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

```
kubect1 create namespace postgres-operator
```



Expected output



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

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

```
kubect1 apply --server-side -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/v3.0.0/deploy/bundle.yaml -n postgres-operator
```



Expected output



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

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

```
kubect1 apply -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/v3.0.0/deploy/cr.yaml -n postgres-operator
```



Expected output



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 v3.0.0 https://github.com/percona/percona-postgresql-operator
```

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

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

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

```
kubectl get pg
```

Expected output

Verifying the cluster operation

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

When the Operator deploys a database cluster, it generates several [Secrets](#). Among them there is the Secret with the credentials of the default PostgreSQL user. This default user has the same username as the cluster name.

- 1 Use `kubectl get secrets -n <namespace>` command to see the list of Secrets objects. The Secrets object you are interested in is named in the format `<cluster_name>-pguser-<cluster_name>` (where the `<cluster_name>` is the [name of your Percona Distribution for PostgreSQL Cluster](#)). For example, if your cluster name is `cluster1`, the Secret name will be `cluster1-pguser-cluster1`.
- 2 Use the following command to get the password of this user. Replace the `<cluster_name>` and `<namespace>` placeholders with your values:

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

- 3 To connect to PostgreSQL, you will use the `pgbouncer` service as the entry point to your cluster. To find this service, use the following command:

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



Look for the service named `<cluster-name>-pgbouncer`.

Sample output



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

```
kubectl run -n <namespace> -i --rm --tty pg-client --image=percona/percona-distribution-postgresql:18.4-1 --restart=Never -- bash -il
```



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

- 5 Run a container with `psql` tool and connect its console output to your terminal. Substitute the `<namespace>` placeholder with your value in the following command to connect as a `cluster1` user to the `cluster1` database via the PostgreSQL interactive terminal.

```
[postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer.<namespace>.svc.cluster.local -p 5432 -U cluster1 cluster1
```



Sample output



Removing the AKS cluster

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

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

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

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



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

Warning

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

Install Percona Distribution for PostgreSQL on OpenShift

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

To install Percona Distribution for PostgreSQL on OpenShift, you need to do the following:

1. First, install the Percona Operator for PostgreSQL Deployment.
2. Next, use the Operator to create Percona Distribution for PostgreSQL cluster.

Installation options

You can install Percona Operator for PostgreSQL on OpenShift in two ways:

- Using the [Operator Lifecycle Manager \(OLM\)](#)  via the OpenShift web console
- Using the command-line interface with `oc` commands

Choose the method that best fits your workflow.

Install via the Operator Lifecycle Manager (OLM)

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

Prerequisites

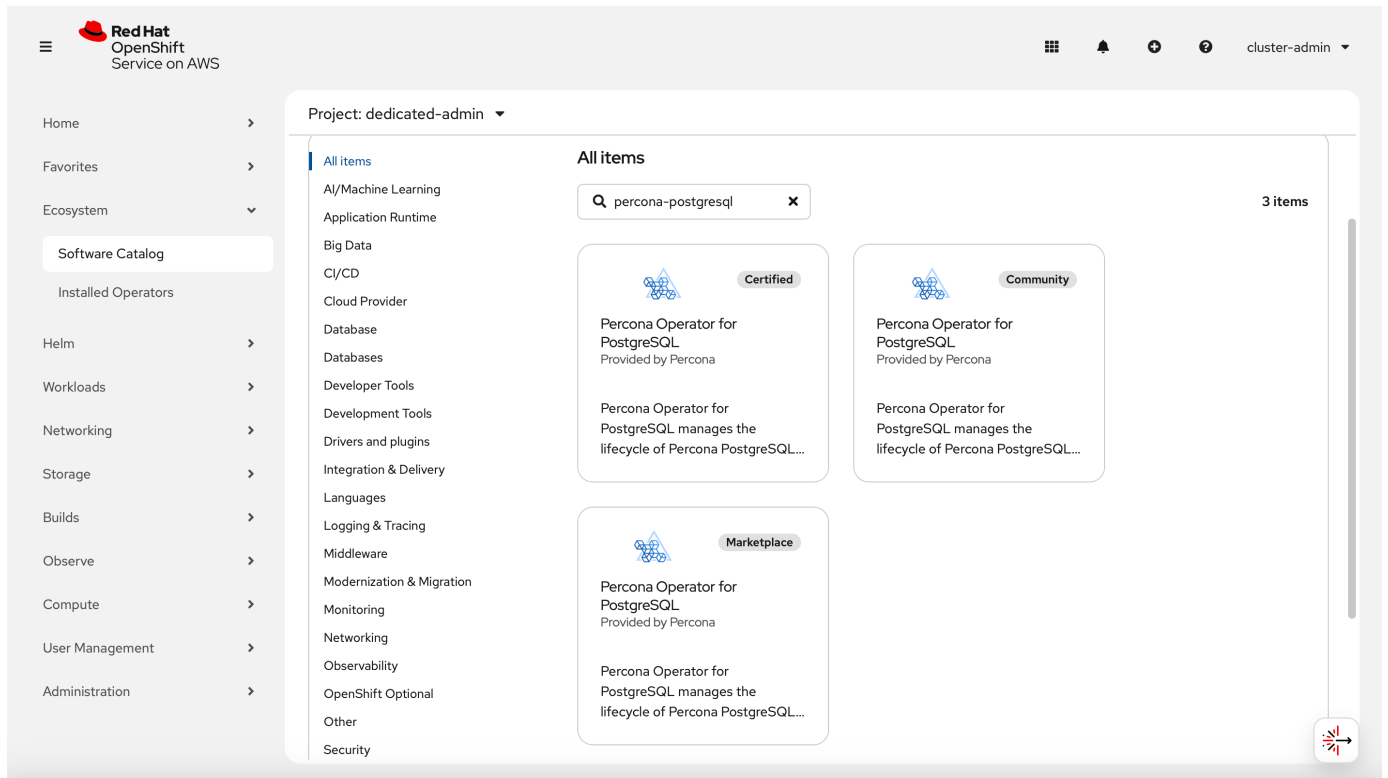
Before you start, ensure you have the following:

1. You can log in to the OLM console
2. You have the ARN role assigned to your OLM user.

Install the Operator Deployment

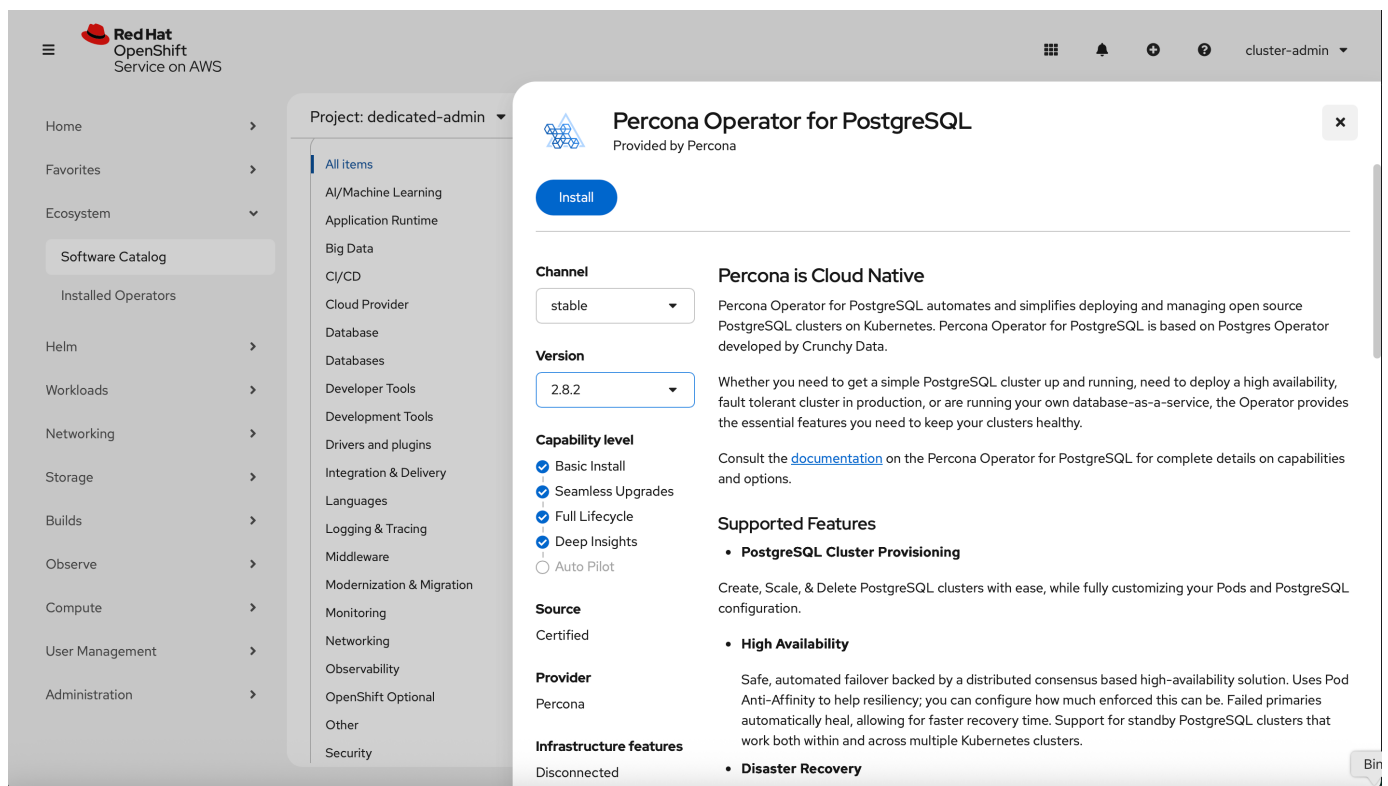
Follow these steps to deploy the Operator and Percona Distribution for PostgreSQL cluster:

1. Login to the OLM and navigate to the Software Catalog.
2. Search for the needed Operator and select it. You may need to change the project for your user:



Then click “Continue”, and “Install”.

3. A new page opens where you specify the ARN role assigned to your user. You also choose the Operator version and the Namespace / OpenShift project you would like to install the Operator into.



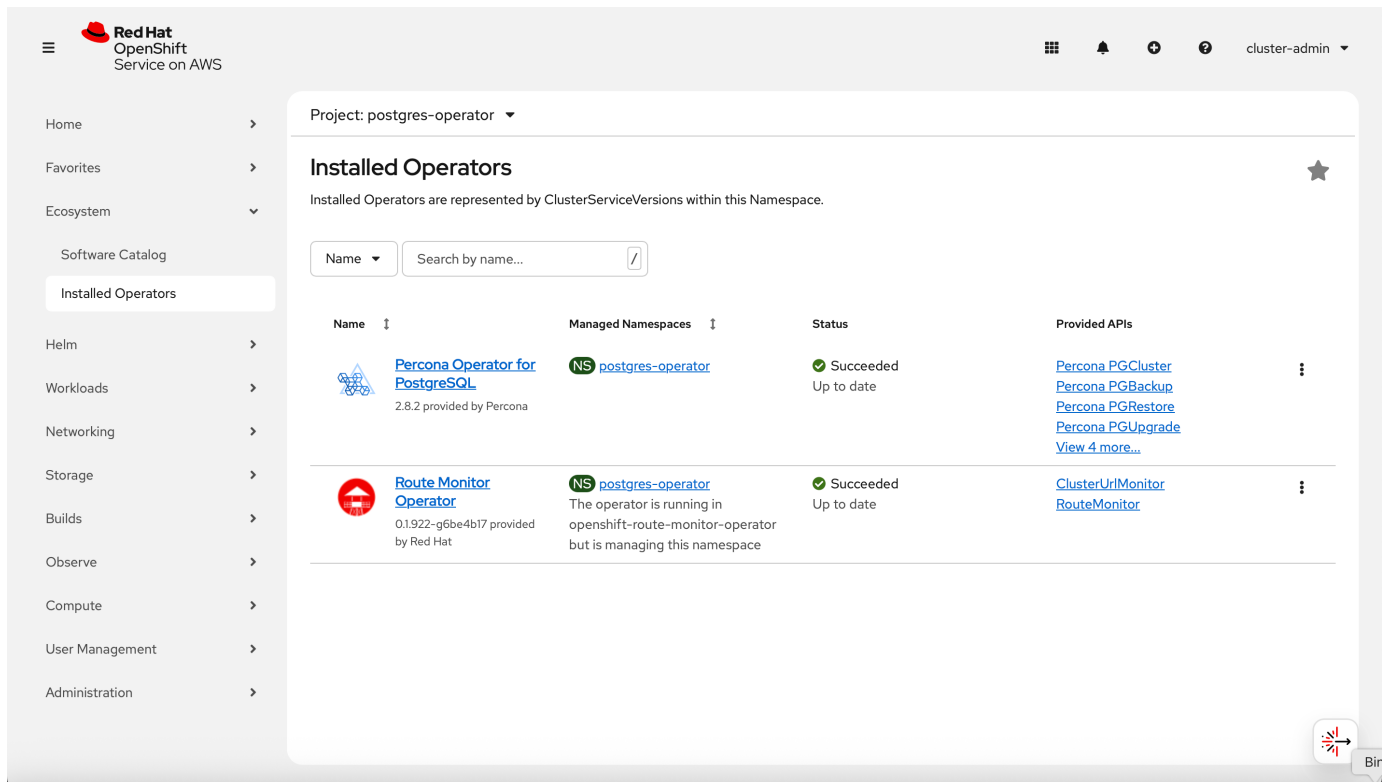
Note

To install the Operator in [multi-namespace \(cluster-wide\) mode](#), choose the value with the `-cw` suffix for the version, and select the “All namespaces on the cluster” radio button for the installation mode instead of choosing a specific Namespace:

The screenshot shows the Red Hat OpenShift Service on AWS console. The left sidebar contains navigation links: Home, Favorites, Ecosystem, Software Catalog (selected), Installed Operators, Helm, Workloads, Networking, Storage, Builds, Observe, Compute, User Management, and Administration. The main content area is titled 'Install Operator' and includes instructions: 'Install your Operator by subscribing to one of the update channels to keep the Operator up to date. The strate...'. A warning box states: 'Cluster in ST... This cluster i... will need to p... upgrade to e...'. The configuration fields are: 'role ARN *' with value 'arn:aws:iam::1191...', 'Update channel *' with value 'stable-cw', 'Version *' with value '2.8.2-cw', 'Installation mode' with 'All namespaces' selected (radio button), and 'Installed Namespace *' with a dropdown showing 'PR openshift-operators'. A 'Create Project' modal is open in the foreground, containing the text 'An OpenShift project is an alternative representation of a Kubernetes namespace.' and fields for 'Name *' (containing 'postgres-operator'), 'Display name', and 'Description'. The modal has 'Create' and 'Cancel' buttons at the bottom.

Click “Install” button to install the Operator.

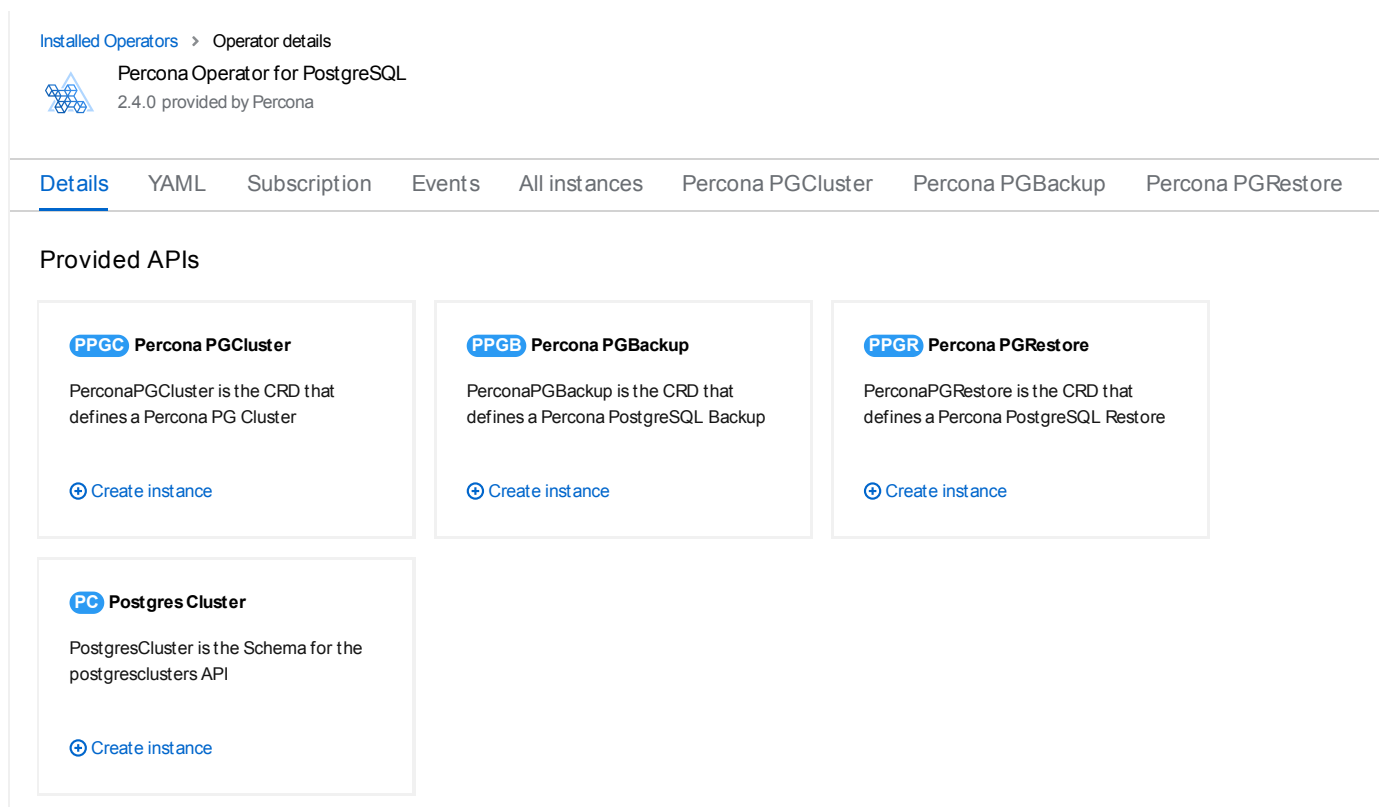
4. You can track the installation flow by clicking the link on the updated page. You will be redirected to the Installed Operators tab. Your installed Operator will appear there.



Deploy Percona Distribution for PostgreSQL

When the installation finishes, you can deploy PostgreSQL cluster.

1. In the “Operator Details” you will see Provided APIs (Custom Resources, available for installation). Click “Create instance” for the **PerconaPGCluster** Custom Resource.



2. You can either go with default settings or edit them as needed. You can use the form or edit the YAML manifest to set needed Custom Resource options.
3. Click the “Create” button to deploy your database cluster.

Install via the command-line interface

The following steps install the latest version of the Operator with default parameters. To install a specific version, replace the `v3.0.0` tag with your value. See the full list of tags [in the Operator repository](#)  on GitHub.

To install the Operator with customized parameters, see [Install Percona Operator for PostgreSQL with customized parameters](#).

Choose the approach that fits your needs:

- **Quick install** — Apply a single bundle file. Use this when you want to get started quickly with default settings.
- **Step-by-step install** — Run each installation step separately. Use this when you need to customize the installation (for example, apply the [anyuid](#)  security context constraint).

Quick install

The bundle file creates the Custom Resource Definition, sets up RBAC, and installs the Operator Deployment in one go.

1. Create the namespace for your cluster:

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



2. Export the namespace as an environment variable:

```
export NAMESPACE=postgres-operator
```



3. Apply the bundle to install the Operator:

```
oc apply --server-side -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/v3.0.0/deploy/bundle.yaml -n $NAMESPACE
```



Step-by-step install

Install the Operator step by step if you wish to have more control over the installation process and modify the manifests before you apply them.

1. Clone the percona-postgresql-operator repository:

```
git clone -b v3.0.0 https://github.com/percona/percona-postgresql-operator
cd percona-postgresql-operator
```

2. Create the Custom Resource Definition (CRD). CRDs are cluster-scoped and apply to all namespaces. You don't need to repeat this step for additional Operator deployments:

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

3. Create the namespace for your cluster (for example, `postgres-operator`):

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

4. Export the namespace as an environment variable:

```
export NAMESPACE=postgres-operator
```

5. Apply RBAC configuration. Your user must have cluster-admin privileges:

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

Note

For example, if you use Google OpenShift Engine, grant cluster-admin privileges with:

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

6. If you use the [anyuid](#)  security context constraint, modify the Operator manifest before applying:

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

7. Create the Operator Deployment:

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

Install Percona Distribution for PostgreSQL cluster

1. Create the Percona Distribution for PostgreSQL cluster:

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



2. Check the cluster status. Creation may take a few minutes:

```
oc get pg -n $NAMESPACE
```



Expected output



Optionally, you can add PostgreSQL Users secrets and TLS certificates before creating the cluster. If you don't, the Operator creates them automatically. See [Users](#) and [TLS certificates](#) for details.

Verifying the cluster operation

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

When the Operator deploys a database cluster, it generates several [Secrets](#). Among them there is the Secret with the credentials of the default PostgreSQL user. This default user has the same username as the cluster name.

- 1 Use `oc get secrets -n <namespace>` command to see the list of Secrets objects. The Secrets object you are interested in is named in the format `<cluster_name>-pguser-<cluster_name>` (where the `<cluster_name>` is the [name of your Percona Distribution for PostgreSQL Cluster](#)). For example, if your cluster name is `cluster1`, the Secret name will be `cluster1-pguser-cluster1`.
- 2 Use the following command to get the password of this user. Replace the `<cluster_name>` and `<namespace>` placeholders with your values:

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



- 3 To connect to PostgreSQL, you will use the `pgbouncer` service as the entry point to your cluster. To find this service, use the following command:

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



Look for the service named `<cluster-name>-pgbouncer`.



Sample output



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

```
oc run -n <namespace> -i --rm --tty pg-client --image=percona/percona-distribution-postgresql:18.4-1 --restart=Never -- bash -il
```



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

- 5 Run a container with `psql` tool and connect its console output to your terminal. Substitute the `<namespace>` placeholder with your value in the following command to connect as a `cluster1` user to the `cluster1` database via the PostgreSQL interactive terminal.

```
[postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer.<namespace>.svc.cluster.local -p 5432 -U cluster1 cluster1
```



Sample output



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 v3.0.0 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 Custom Resource Definition for Percona Distribution for PostgreSQL should be created from the `deploy/crd.yaml` file. Custom Resource Definition extends the standard set of resources which Kubernetes “knows” about with the new items (in our case ones which are the core of the Operator). [Apply it](#) as follows:

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

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

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

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

Note

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

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

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

Note

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

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

5 Start the Operator within Kubernetes:

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

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

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

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

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

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

Expected output

Verifying the cluster operation

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

When the Operator deploys a database cluster, it generates several [Secrets](#). Among them there is the Secret with the credentials of the default PostgreSQL user. This default user has the same username as the cluster name.

1 Use `kubectl get secrets -n <namespace>` command to see the list of Secrets objects. The Secrets object you are interested in is named in the format `<cluster_name>-pguser-<cluster_name>` (where the `<cluster_name>` is the [name of your Percona Distribution for PostgreSQL Cluster](#)). For example, if your cluster name is `cluster1`, the Secret name will be `cluster1-pguser-cluster1`.

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

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



- 3 To connect to PostgreSQL, you will use the `pgbouncer` service as the entry point to your cluster. To find this service, use the following command:

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



Look for the service named `<cluster-name>-pgbouncer`.



Sample output



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

```
kubectl run -n <namespace> -i --rm --tty pg-client --image=percona/percona-  
distribution-postgresql:18.4-1 --restart=Never -- bash -il
```



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

- 5 Run a container with `psql` tool and connect its console output to your terminal. Substitute the `<namespace>` placeholder with your value in the following command to connect as a `cluster1` user to the `cluster1` database via the PostgreSQL interactive terminal.

```
[postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer.  
<namespace>.svc.cluster.local -p 5432 -U cluster1 cluster1
```



Sample output



Deleting the cluster

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

Configuration

Users

The Percona Operator for PostgreSQL includes built-in functionality to simplify management of users and databases within your PostgreSQL cluster. By default, the Operator creates a single unprivileged user and the database that matches the cluster name.

However, many production workloads require more granular user access, separate databases for different applications, or restricted privileges for security and compliance. With the Operator, you can define custom users and manage their access to your database cluster resources:

This document explains how you can customize user and database management for your specific use case.

Understanding default user management

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

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

As an example, with the default PostgreSQL cluster name `cluster1`, the Operator creates the following:

- A database named `cluster1`.
- A schema named `cluster1` for the database `cluster1`

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

Custom users and databases

You can add and manage custom users and databases using the `spec.users` section in the Custom Resource. You can do this:

- at the cluster creation time
- at runtime.

Considerations

Here's what you need to know:

Adding custom users and databases:

- If you define custom users in `spec.users` during cluster creation, the Operator does **not** create any default users or databases (except for the `postgres` database). If you want additional databases, you must specify them explicitly.
- For each user added in `spec.users`, the Operator creates a Secret named `<clusterName>-pguser-<userName>` with that user's credentials. You can override this Secret name using the `spec.users.secretName` option.
 - If you do **not** specify any databases for a custom user, the resulting Secret will **not** include `dbname` or `uri` fields. This means the user will not have access to any database until one is assigned later.
 - If you include at least one database in `spec.users.databases` for the user, the Secret will include connection credentials for the **first** database in the list (`dbname` and `uri`).
- You can add a special `postgres` user as one of the custom users. This user is granted access to the `postgres` database, but its privileges cannot be changed.
- By default, the top-level `autoCreateUserSchema` option is set to `true`. This means each user will have automatically-created schemas in all databases listed for this user under `users.databases`.
- By default, users without superuser privileges do not have access to the `public` schema. To allow a non-superuser to create and update tables in the `public` schema, set the `grantPublicSchemaAccess` option to `true`. This gives the user permission to create and update tables in the `public` schema of every database they own.
- Your custom superusers automatically have access to the `public` schema for their assigned databases.
- If multiple users are granted access to the `public` schema in the same database, each can only access tables they themselves have created. If you want one user to access tables created by another user, the

table owner must explicitly grant privileges via PostgreSQL.

Behavior when removing or modifying users and databases:

- The Operator does **not** automatically drop users if you remove them from the Custom Resource, to prevent accidental data loss.
- Similarly, the Operator does **not** automatically drop databases when you remove them from the Custom Resource. (See how to actually drop a database [here](#).)
- Role attributes (such as SUPERUSER) are not automatically removed if you delete them from the Custom Resource. You must specify the opposite attribute (e.g., NOSUPERUSER) to explicitly revoke privileges.

Creating a new user

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

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



After you apply such changes with the usual `kubectl apply -f deploy/cr.yaml` command, the Operator will create the new user as follows:

- The credentials of this user are populated in the `<clusterName>-pguser-perconapg` secret. There are no connection credentials.
- The user is unprivileged.

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

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



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

Managing user passwords

Operator-generated passwords

The Operator generates a random password for each PostgreSQL user it creates. PostgreSQL allows almost any character in its passwords and the Operator generates passwords in [ASCII](#) format by default.

Your application may have stricter requirements to password creation. For example, if you need passwords without special characters, set the `spec.users.password.type` field for that user to `AlphaNumeric`.

To have the Operator generate a new password, remove the existing `password` field from the user Secret.

For example, to generate a new password for the user `cluster1` in the PostgreSQL cluster `cluster1` running in the `postgres-operator` namespace, use the following `kubectl patch` command:

```
kubectl patch secret -n postgres-operator cluster1-pguser-cluster1 -p '{"data":  
{"password":""}}'
```



Replace the namespace and the secret name with your values to reuse this command.

Custom passwords

You may want a complete control over user passwords by setting a specific password for a PostgreSQL user instead of letting Percona Operator for PostgreSQL generate one for you. To do that, create a user Secret and specify the password within.

When you create a user Secret, the way you name it is important:

- If you specify a Secret name using the default naming convention that the Operator expects (`<clusterName>-pguser-<userName>`), the Operator will detect and use it automatically.
- If you use a custom name for your Secret, you must explicitly reference that Secret in the Custom Resource to let the Operator know about it.

The Operator looks for two fields in the Secret:

- `password`: the plaintext password.
- `verifier`: a hashed representation of the password using `SCRAM-SHA-256`.

When the `verifier` changes, the Operator updates the password inside the PostgreSQL cluster. This approach ensures the password is securely passed into the database.

You can set a custom password in these ways:

- You can provide a plaintext password in the `password` field and omit the verifier. The Operator will detect this and automatically generate a SCRAM verifier for your password.
- You can supply both the `password` and the `verifier` yourself. If both are present, the Operator will use them as-is and skip the generation step. Once the Secret contains both values, the Operator will make sure the credentials are correctly applied to PostgreSQL.

Here's how to set a custom password within a Secret with a custom name:

1. Export your namespace as an environment variable

```
export NAMESPACE=postgres-operator
```



2. Create a Secrets object. For example, `cat-credentials`:

```
kubectl apply -n $NAMESPACE -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: cat-credentials
type: Opaque
data:
  password: $(echo -n 'mySuperStr0ngp@ssword' | base64)
EOF
```



Sample output

```
secret/cat-credentials created
```

3. Add a user and reference the Secret for them in the Custom Resource:

via cr.yaml

```
users:
- name: cat
  databases:
    - zoo
  secretName: "cat-credentials"
  grantPublicSchemaAccess: true
```



Apply the configuration:

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



via kubectl patch

To update a running cluster, use the `kubectl patch` command:

```
kubectl patch pg cluster1 -n $NAMESPACE --type=merge --patch '{
"spec": {
  "users": [
    {
      "name": "cat",
      "databases": ["zoo"],
      "secretName": "cat-credentials",
      "grantPublicSchemaAccess": true
    }
  ]
}
}'
```



4. After you update the cluster, the Operator updates the Secret with the login credentials and connection information. View the Secret object to verify this with this command:

```
kubectl get secret cat-credentials -o yaml -n $NAMESPACE
```



5. Verify that the user is created by [connecting to the database](#) as your custom user.

Password rotation

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

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



In the same way you can update a password with your custom one for the user. Do it as follows:

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

Adjusting privileges

You can set role privileges by using the standard [role attributes](#) that PostgreSQL provides and adding them to the `spec.users.options` subsection in the Custom Resource.

Grant privileges

The following example will make the `perconapg` a superuser. You can add the following to the spec in your `deploy/cr.yaml`:

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

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

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

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

Revoke privileges

To revoke the superuser privilege afterwards, apply the following configuration:

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

postgres User

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

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



This will create a Secret named `<clusterName>-pguser-postgres` that contains the credentials of the `postgres` user. The Operator creates a user `postgres` who can access the `postgres` database.

Deleting users and databases

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

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



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

```
DROP DATABASE pgtest;
```



Superuser and pgBouncer

For security reasons we do not allow superusers to connect to cluster through pgBouncer by default. As a superuser, you can connect through the `primary` service. Read more about this service in [exposure documentation](#).

Otherwise you can use the [proxy.pgBouncer.exposeSuperusers](#) Custom Resource option to enable superusers connection via pgBouncer.

LDAP authentication

LDAP authentication



Version added: [2.9.0](#)

Authentication is the process of verifying a client identity. When a user connects to the database they must authenticate themselves against it before doing any actions or accessing any resources.

By default, Percona Operator for PostgreSQL authenticates users with a database password stored in Secrets. You can simplify and centralize user management by enabling **LDAP authentication** with Percona Operator for PostgreSQL. Instead of verifying the passwords locally, your PostgreSQL cluster delegates password verification to an LDAP server. The authorization is done based on the [roles and privileges that you define for users](#) in the Custom Resource.

Using LDAP authentication means you can:

- Streamline account management: add, remove, or update users in one place
- Let users log in to the database with their existing organizational credentials and don't have to remember multiple logins
- Easily enforce centralized password policies and improve security

How LDAP authentication works

1. You configure usernames and passwords in the LDAP server of your choice as Distinguished Names (DNs). You also specify these users in the Custom Resource.
2. You configure the authentication parameters in the Custom Resource. This instructs the Operator to generate authentication rules in `pg_hba.conf`.
3. A client connects to PostgreSQL with a username and a password.
4. The rules in `pg_hba.conf` determine which connections use LDAP.
5. PostgreSQL performs an LDAP bind to verify the user (using either [simple bind or search+bind](#)).
6. If the bind succeeds, PostgreSQL grants access according to the privileges you defined for this user.

Connection and bind modes

Connection: plain LDAP vs LDAPS

You can connect to the LDAP server over an unencrypted or encrypted channel.

Plain LDAP sends credentials over the network without encryption. This connection is available at port `389`. Use it only in trusted networks, for example in development or when the LDAP server runs in the same cluster.

LDAPS uses TLS to encrypt traffic between PostgreSQL and the LDAP server. This connection is available at port `636`. Use LDAPS in production. You must provide the CA certificate that signed the LDAP server's certificate so PostgreSQL can verify the connection. You can generate these certificates via the `cert-manager` or use your custom ones.



Encryption scope

LDAPS encrypts only the link between PostgreSQL and the LDAP server. To encrypt the connection between the client and PostgreSQL as well, configure [TLS for the database](#).

Bind modes

PostgreSQL supports two ways to look up and verify users in LDAP.

Simple bind builds the user's Distinguished Name (DN) from a prefix and suffix you configure. The DN format is `<ldapprefix><username><ldapsuffix>`. For example, with `ldapprefix="uid="` and `ldapsuffix=",ou=users,dc=example,dc=com"`, user `alice` becomes `uid=alice,ou=users,dc=example,dc=com`. PostgreSQL then binds to LDAP as that DN with the password the client supplied. This is the simplest mode and works well when your LDAP structure is predictable.

Search+bind is more flexible. PostgreSQL first binds to LDAP (with a fixed service account or anonymously), searches for the user under a base DN using an attribute or filter, then re-binds as the found user with the client's password. Use search+bind when users live in different subtrees or when you need custom search logic (for example, matching by `uid` or `mail`). It requires `ldapbasedn` and either `ldapsearchattribute` or `ldapsearchfilter`; optionally `ldapbinddn` and `ldapbindpasswd` for the initial bind.

You cannot mix simple-bind options (`ldapprefix`, `ldapsuffix`) with search+bind options (`ldapbasedn`, `ldapsearchattribute`, etc.) in the same rule.

To learn more, see [PostgreSQL documentation](#) 

Username mapping

- PostgreSQL expects the LDAP username to match the PostgreSQL username.
- If a username contains special characters (such as `@`), escape them appropriately in both LDAP and PostgreSQL. For example, you can use `user@domain` directly if both LDAP and PostgreSQL support that character in usernames. For accurate rules on escaping special characters, refer to [RFC4514](#) , [RFC4515](#) , [RFC4516](#) , or consult your LDAP server documentation.

Next steps

[Configure LDAP authentication](#)

Configure LDAP authentication

This document walks you through setting up LDAP authentication in your PostgreSQL cluster managed by the Operator. For more information about LDAP authentication, how it works and why you need it, see [LDAP authentication overview](#).

Assumptions

1. To enable LDAP authentication, you must configure two components: an LDAP server and the Percona Operator for PostgreSQL.

The installation and initial setup of the LDAP server are outside the scope of this document. We assume your LDAP server is already running and accessible to the Operator over the network.

1. This guide assumes you are familiar with your LDAP server's schema.
2. The examples in this document use [OpenLDAP :octicons-external-link-16:](#) as the LDAP server. If you are using Active Directory, please refer to their [official documentation](#) for user and group configuration.
3. This example setup uses [Simple bind](#) mode and the bind user DN `cn=admin,dc=ldap,dc=local`. Make sure to add your own bind user and use it in the commands shown later in this document.

To use search+bind mode, specify the required options in your configuration. See the [PostgreSQL documentation](#) [↗](#) for usage examples, and refer to the [Custom Resource reference](#) for a description of available options.

Prerequisites

Before configuring LDAP authentication, ensure you have:

- A running LDAP server (e.g., OpenLDAP, Active Directory) reachable from the PostgreSQL Pods
- LDAP server hostname or IP and port: `389` for plain LDAP, `636` for LDAPS
- For LDAPS: the CA certificate that signed the LDAP server's TLS certificate



LDAPS certificate hostname

For LDAPS, the LDAP server certificate must match the hostname that you will specify in the `ldapservers` option in the Custom Resource (either `commonName` or a Subject Alternative Name). If it does not match, PostgreSQL will reject the connection.

Clone the repository, because you will need to modify the default configuration in the cluster Custom Resource:

```
git clone -b v3.0.0 https://github.com/percona/percona-postgresql-operator
```



Configure the OpenLDAP server

Add users to your LDAP directory. To do that, use LDIF (LDAP Data Interchange Format) files. LDIF is a standard plain-text format for representing LDAP entries and operations. You create LDIF files to define new users, groups, or other LDAP directory entries, and then apply them to your directory using LDAP utilities like `ldapadd`. LDIF can describe any LDAP entry or modification, making it essential for managing your LDAP Data Information Tree (DIT).

Add the following LDIF portions to your OpenLDAP server. This creates:

- `ou=perconadba,dc=ldap,dc=local` — This is an **organizational unit (OU)** entry in your LDAP directory. An OU acts like a folder or grouping mechanism to organize related LDAP objects, such as users. In this example, “perconadba” is the group under which users will be organized.
- `uid=percona,ou=perconadba,dc=ldap,dc=local` — This is a **user entry** in LDAP, representing the “percona” user. It belongs to the organizational unit “perconadba”. This entry will store information like username, password, and other user attributes. This is the account PostgreSQL will try to authenticate against when the client connects using the “percona” username.
- `cn=admin,ou=perconadba,dc=ldap,dc=local` — This is a **group entry** in LDAP, representing a group named “admin” within the organizational unit “perconadba”. This entry groups together related users, in this case adding the user `uid=percona,ou=perconadba,dc=ldap,dc=local` as a member. Groups like this can be used to assign roles or manage access collectively within LDAP.

Use the `ldapadd` command to add this structure. Use your LDAP server hostname and your bind user DN and password in the following command:

```
ldapadd -x -H ldap://<ldap-server-hostname>:389 -D "cn=admin,dc=ldap,dc=local" -w
adminpassword <<EOF
dn: ou=perconadba,dc=ldap,dc=local
objectClass: organizationalUnit
ou: perconadba

dn: uid=percona,ou=perconadba,dc=ldap,dc=local
objectClass: inetOrgPerson
uid: percona
cn: percona
sn: percona
userPassword: mysecretpassw

dn: cn=admin,ou=perconadba,dc=ldap,dc=local
objectClass: groupOfUniqueNames
cn: admin
uniqueMember: uid=percona,ou=perconadba,dc=ldap,dc=local
EOF
```

Verify that the user was added successfully:

```
ldapsrch -x \
-H ldap://<ldap-server-hostname>:389 \
-D "cn=admin,dc=ldap,dc=local" \
-w adminpassword \
-b "ou=perconadba,dc=ldap,dc=local" \
"(uid=percona)"
```

 Expected output



Configure the Operator and PostgreSQL

Plain LDAP

Use plain LDAP only in trusted networks. In this connection mode, PostgreSQL connects to the LDAP server on port 389.

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

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

2. Install the Operator deployment, following the steps from the [Quick install guide](#)
3. Edit the Custom Resource manifest and specify the following configuration:
 - Add the same user to the Custom Resource manifest as you defined in your LDAP directory. Specify the privileges you want this user to have in the `spec.users` section of the manifest. In this example setup

this is the user `percona`.

- Add authentication rules in the `spec.authentication.rules` section. The Operator will create a rule in the `pg_hba.conf` file based on this data:
- `connection` - set to `host`.
- `method` - set to `ldap`
- `users` - add users that will authenticate using LDAP
- `options.ldapserver` - the hostname of your LDAP server
- `options.ldapport` - the port to connect to. Set it to `389` for plain LDAP.
- `ldapprefix` - The string to prepend the username when forming a DN during simple bind. In our example this is `"uid="`.
- `ldapsuffix` - The string to append to the username when forming a DN to bind as. In our example this is `",ou=perconadba,dc=ldap,dc=local"`

The example configuration looks like this:

```
spec:
  users:
    - name: percona
      databases:
        - percona
  # The rest of your configuration
  authentication:
    rules:
      - connection: host
        method: ldap
        users:
          - percona
        options:
          ldapport: 389
          ldapprefix: uid=
          ldapscheme: ldap
          ldapserver: openldap
          ldapsuffix: ",ou=perconadba,dc=ldap,dc=local"
```

4. Apply the configuration to create the cluster:

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

LDAPS (LDAP over TLS)

LDAPS encrypts traffic between PostgreSQL and the LDAP server. Use it in production.

For LDAPS connection, you must provide the LDAP server's certificate to PostgreSQL. To do that, create a Secret with this certificate.

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

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



2. Create a Secret with the LDAP server's CA certificate. Specify the path to the CA certificate:

```
kubectl -n $NAMESPACE create secret generic ldap-ca \
--from-file=ca.crt=/path/to/ca.crt
```



3. Install the Operator deployment, following the steps from the [Quick install guide](#)

4. Edit the Custom Resource manifest and specify the following configuration:

- Add the same user to the Custom Resource manifest as you defined in your LDAP directory. Specify the privileges you want this user to have in the `spec.users` section of the manifest. In this example setup this is the user `percona`.
- Add authentication rules in the `spec.authentication.rules` section. The Operator will create a rule in the `pg_hba.conf` file based on this data:
- `connection` - set to `host`.
- `method` - set to `ldap`
- `users` - add users that will authenticate using LDAP
- `options.ldapserver` - the hostname of your LDAP server
- `options.ldapport` - the port to connect to. Set it to `636` for LDAP over TLS connection type.
- `ldapprefix` - The string to prepend the username when forming a DN during simple bind. In our example this is `"uid="`.
- `ldapsuffix` - The string to append to the username when forming a DN to bind as. In our example this is `",ou=perconadba,dc=ldap,dc=local"`
- Reference the Secret with the LDAP Server CA certificate in the `config.files.secret` option. The Operator mounts the CA certificate at the path `/etc/postgres/ldap/ca.crt` and automatically sets the `LDAPTLS_CACERT` environment value to this path.

This is the example configuration:

```
spec:
  users:
    - name: percona
      databases:
        - percona

  authentication:
    rules:
      - connection: host
        method: ldap
        users:
          - percona
        options:
          ldapscheme: ldaps
          ldapserver: openldap-tls
          ldapport: 636
          ldapprefix: "uid="
          ldapsuffix: ",ou=perconadba,dc=ldap,dc=local"

  config:
    files:
      - secret:
          name: ldap-ca
          items:
            - key: ca.crt
              path: ldap/ca.crt
```

5. Apply the configuration:

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

Verify LDAP authentication

To verify the LDAP authentication, let's do the following:

- Create a `pg-client` Pod
- Connect to PostgreSQL via `pgBouncer` as the user that has LDAP authentication configured (`percona` in our example)

Here are the steps:

1. List the services:

```
kubectl get svc -n $NAMESPACE
```

Look for the `<cluster-name>-pgbouncer` service.

2. Create a Pod where you start a container with Percona Distribution for PostgreSQL and establish a shell session inside:

```
kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-postgresql:18 --restart=Never -- bash
```



3. Inside the Pod, connect to PostgreSQL:

```
psql "postgres://percona@<cluster-name>-pgbouncer.<namespace>.svc:5432/percona"
```



Expected output



4. Specify the LDAP password for the user.



Expected output



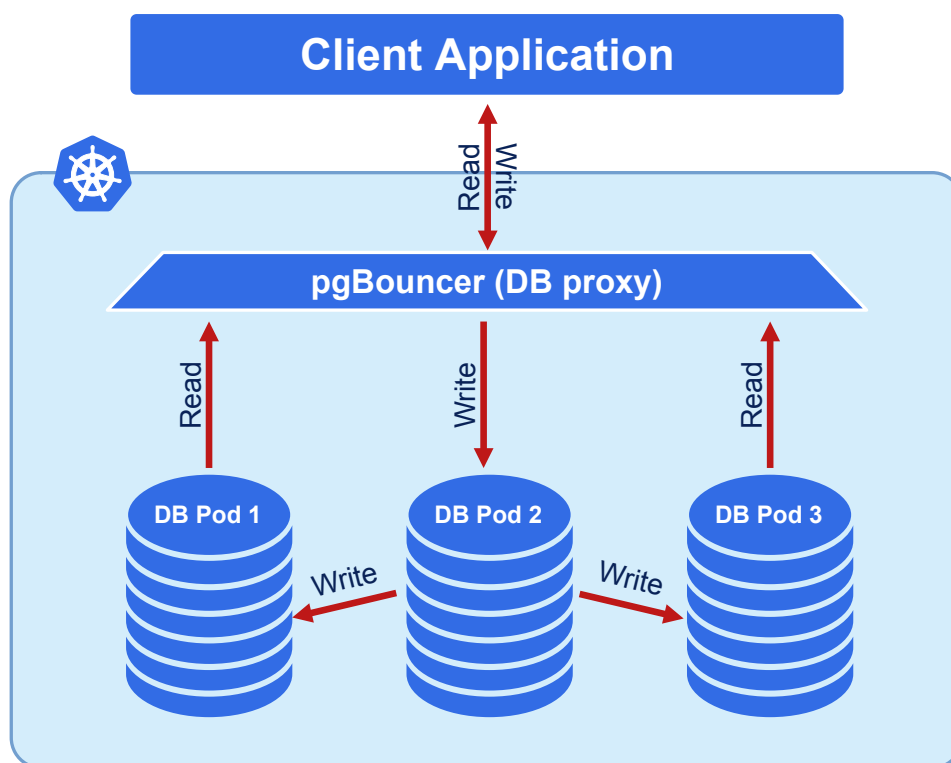
Exposing cluster

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

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

PgBouncer

We recommend exposing the cluster through PgBouncer, which is enabled by default.



You can disable pgBouncer by setting `proxy.pgBouncer.replicas` to 0.

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

```
proxy:
  pgBouncer:
    replicas: 2
    image: docker.io/percona/percona-pgbouncer:1.25.2-1
    expose:
      type: LoadBalancer
```



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

```
kubectl get service
```



Expected output



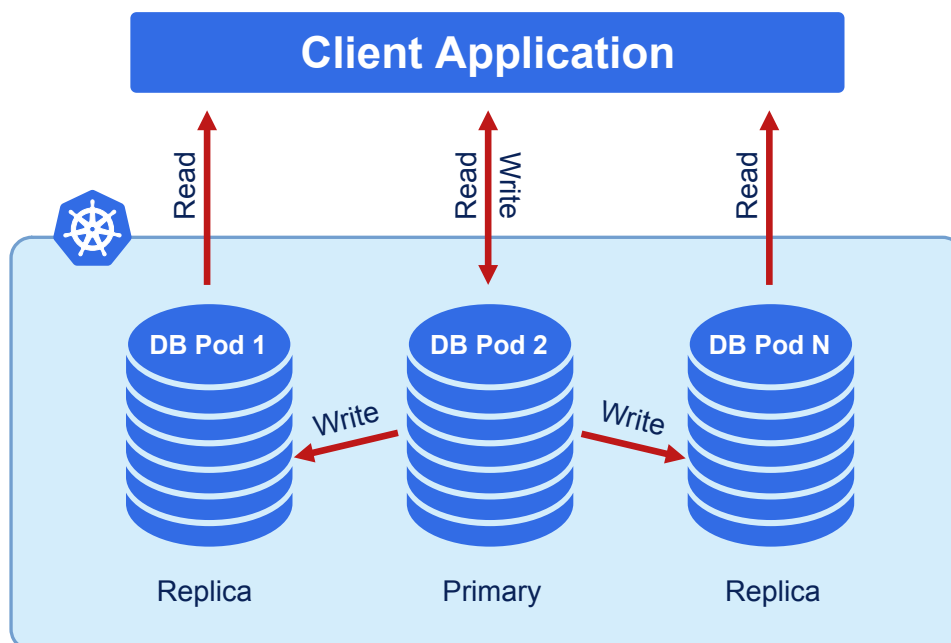
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
...					
cluster1-pgbouncer	LoadBalancer	10.88.8.48	34.133.38.186	5432:30601/TCP	20m
...					

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

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

Exposing the cluster without pgBouncer

You can connect to the cluster without a proxy.



For that use `<clusterName>-ha` Service object:

```
kubectl get service
```





Expected output



NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
...					
cluster1-ha	ClusterIP	10.88.8.121	<none>	5432/TCP	115s
...					
cluster1-replicas	ClusterIP	10.88.8.115	<none>	5432/TCP	2m16s

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

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

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



Changing PostgreSQL options

You may need to pass specific options to PostgreSQL instances directly, beyond the default configuration and options, available in the Custom Resource. For this purpose, use the [PostgreSQL dynamic configuration method](#) provided by Patroni. You can pass PostgreSQL options to Patroni through the Custom Resource. The Operator uses [Patroni dynamic configuration](#) to apply your changes.

Add your PostgreSQL options to the `patroni.dynamicConfiguration.postgresql.parameters` section in your `deploy/cr.yaml` Custom Resource:

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

Apply the updated Custom Resource:

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

This dynamically applies the changes to PostgreSQL instances both for new clusters during cluster creation and existing clusters at runtime.

Most options take effect without a PostgreSQL server restart. Some options, such as `wal_level` and `shared_buffers`, have the [postmaster context](#) and require a PostgreSQL restart. For these options, Patroni performs a rolling restart of all instances after you apply the change. To check whether an option requires a restart, run in PostgreSQL: `SELECT name, context FROM pg_settings;`

Note

The Operator does not validate the options it passes to Patroni. Invalid values can make the cluster unavailable. Also, only PostgreSQL parameters in the `patroni.dynamicConfiguration.postgresql.parameters` subsection are applied. Other Patroni options in `patroni.dynamicConfiguration` subsection are ignored.

Configuring wal_level

The `wal_level` option controls how much information is written to PostgreSQL WAL files. You can set it in `patroni.dynamicConfiguration.postgresql.parameters`:

- `replica` (PostgreSQL default) – sufficient for physical replication and most workloads

- `logical` (Operator default) — required for logical replication; increases WAL volume and I/O.

Read more about `wal_level` values in [PostgreSQL documentation](#) .

Note

Though the `wal_level` option can also have the value `minimal`, it will be rejected by the validation rules, since other parameters, such as `hot_standby`, require more WAL data.

```
...
patroni:
  dynamicConfiguration:
    postgresql:
      parameters:
        wal_level: replica
```



Use `replica` when you need physical replication or no replication. Use `logical` when you need logical replication. Both values allow for point-in-time recovery.

The `wal_level` parameter has the postmaster context. After you change it, Patroni restarts all PostgreSQL instances.

Note

The Operator manages certain PostgreSQL parameters (such as `archive_mode`, `archive_command`, `restore_command`, and TLS settings) internally and reverts any user changes. Other parameters (such as `wal_level`, `archive_timeout`, `jit`) can be overridden. See [Immutable options](#) for the full list.

Using host-based authentication (`pg_hba`)

PostgreSQL Host-Based Authentication (`pg_hba`) controls database access based on the client IP or hostname. Configure it in the `patroni.dynamicConfiguration.postgresql.pg_hba` section of the Custom Resource:

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



This example allows all hosts to connect to any database using MD5 password authentication.

You can use both `parameters` and `pg_hba` in the same configuration:

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

Apply the changes with `kubectl apply -f deploy/cr.yaml`.

Binding Percona Distribution for PostgreSQL components to specific Kubernetes/OpenShift Nodes

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

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

Affinity and anti-affinity

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

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

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

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



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

Topology Spread Constraints

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

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

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

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

Tolerations

Tolerations allow Pods having them to be able to land onto nodes with matching *taints*. 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, tolerations 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.

Labels and annotations

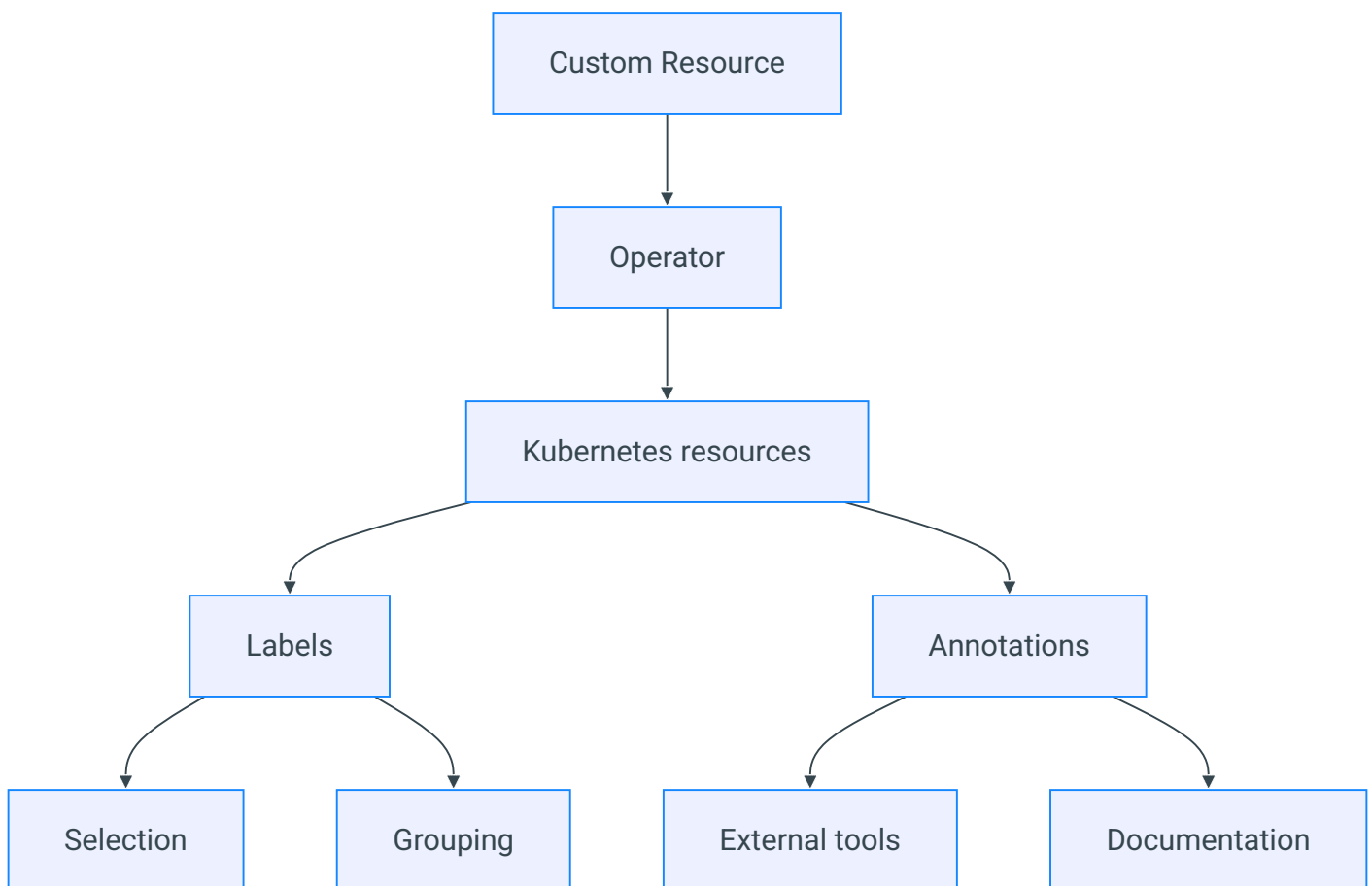
[Labels](#)  and [annotations](#)  are used to attach additional metadata information to Kubernetes resources.

Labels and annotations are rather similar but differ in purpose.

Labels are used by Kubernetes to identify and select objects. They enable filtering and grouping, allowing users to apply selectors for operations like deployments or scaling.

Annotations are assigning additional *non-identifying* information that doesn't affect how Kubernetes processes resources. They store descriptive information like deployment history, monitoring configurations or external integrations.

The following diagram illustrates this difference:



Both Labels and Annotations are assigned to the following objects managed by Percona Operator for PostgreSQL:

- Custom Resource Definitions
- Custom Resources
- Deployments
- Services

- StatefulSets
- PVCs
- Pods
- ConfigMaps and Secrets

When to use labels and annotations

Use **Labels** when:

- The information is used for object selection
- The data is used for grouping or filtering
- The information is used by Kubernetes controllers
- The data is used for operational purposes

Use **Annotations** when:

- The information is for external tools
- The information is used for debugging
- The data is used for monitoring configuration

Labels and annotations used by Percona Operator for PostgreSQL

Labels

Name	Objects	Description	Example values
pgv2.percona.com/version	CustomResourceDefinition	Specifies the version of the Percona Operator for PostgreSQL.	3.0.0
app.kubernetes.io/instance	Services, StatefulSets, Deployments	Identifies a specific instance of the application	cluster1
app.kubernetes.io/managed-by	Services, StatefulSets	Indicates the controller managing the object	percona-postgresql-operator
app.kubernetes.io/component	Services, StatefulSets	Specifies the component within the application	postgres, pgbouncer, pgbackrest

<code>app.kubernetes.io/part-of</code>	Services, StatefulSets	Indicates the higher-level application the object belongs to	percona-postgresql
<code>app.kubernetes.io/name</code>	Services, StatefulSets, Deployments, etc.	Specifies the name of the application	percona-postgresql
<code>postgres-operator.crunchydata.com/cluster</code>	StatefulSets, Deployments, Services, PVCs	Specifies the name of the application	cluster1
<code>postgres-operator.crunchydata.com/instance</code>	Services, StatefulSets, Deployments	Identifies a specific instance of the application	cluster1
<code>postgres-operator.crunchydata.com/instance-set</code>	Pods, StatefulSets	Describes the set of instances (such as a group of pods) within the PostgreSQL cluster.	
<code>postgres-operator.crunchydata.com/name</code>	pgBackRest resources (Jobs, CronJobs, Deployments, PVCs, etc.)	Used to specify the name of a pgBackRest repository.	
<code>postgres-operator.crunchydata.com/patroni</code>	Pods, StatefulSets	Indicates Patroni-related resources.	
<code>postgres-operator.crunchydata.com/role</code>	Pods, PVCs, Services	The role that Patroni sets on the Pod that is currently the leader	
<code>postgres-operator.crunchydata.com/cluster-certificate</code>	Secrets	Identifies a secret containing a cluster certificate	postgres-tls
<code>postgres-operator.crunchydata.com/data</code>	Pods, PVCs	Identifies Pods and Volumes that store Postgres data	
<code>postgres-operator.crunchydata.com/move-job</code>	Jobs	Identifies a directory move Job.	
<code>postgres-operator.crunchydata.com/move-pgbackrest-repo-dir</code>	Jobs	Identifies a Job moving a pgBackRest repo directory.	

<code>postgres-operator.crunchydata.com/move-pgdata-dir</code>	Jobs	Identifies a Job moving a pgData directory.	
<code>postgres-operator.crunchydata.com/move-pgwal-dir</code>	Jobs	Identifies a Job moving a pg_wal directory.	
<code>postgres-operator.crunchydata.com/pgbackrest</code>	pgBackRest resources	Indicates a resource that is for pgBackRest.	
<code>postgres-operator.crunchydata.com/pgbackrest-backup</code>	Backup Jobs	Indicates a resource that is for a pgBackRest backup.	
<code>postgres-operator.crunchydata.com/pgbackrest-config</code>	ConfigMaps, Secrets	Indicates a ConfigMap/Secret for pgBackRest.	
<code>postgres-operator.crunchydata.com/pgbackrest-dedicated</code>	ConfigMaps	Indicates a ConfigMap that is for a dedicated pgBackRest repo host.	
<code>postgres-operator.crunchydata.com/pgbackrest-repo</code>	Deployments, Pods	Indicates a Deployment or a Pod for a pgBackRest repo.	The name of the repository you define in CR
<code>postgres-operator.crunchydata.com/pgbackrest-volume</code>	PVCs	Indicates a PVC for a pgBackRest repo volume.	
<code>postgres-operator.crunchydata.com/pgbackrest-cronjob</code>	CronJobs	Indicates a resource is a pgBackRest CronJob.	
<code>postgres-operator.crunchydata.com/pgbackrest-restore</code>	Jobs, Pods	Indicates a Job/Pod for a pgBackRest restore.	
<code>postgres-operator.crunchydata.com/pgbackrest-restore-config</code>	ConfigMaps, Secrets	Indicates a configuration resource (e.g. a ConfigMap or Secret) for pgBackRest restore.	
<code>postgres-operator.crunchydata.com</code>	Pods	Added to Pods running the exporter container for	

<code>m/crunchy-postgres-exporter</code>		Prometheus discovery.	
<code>postgres-operator.crunchydata.com/pguser</code>	Secrets, Users	Identifies the PostgreSQL user an object is for/about.	Username
<code>postgres-operator.crunchydata.com/startup-instance</code>	Pods, Jobs	Indicates the startup instance associated with a resource.	
<code>postgres-operator.crunchydata.com/cbc-pgrole</code>	Secrets	Identifies a CBC PostgreSQL role secret.	
<code>postgres-operator.crunchydata.com/pgadmin</code>	pgAdmin resources	Indicates a resource for a standalone pgAdmin instance.	

Annotations

Name	Objects	Description	Example Values
<code>postgres-operator.crunchydata.com/trigger-switchover</code>	Custom Resource	Initiates a failover, switchover	
<code>postgres-operator.crunchydata.com/pg-backrest-backup-job-completion</code>	Restore, PVC	Added to restore jobs, pvcs, and VolumeSnapshots that are involved in the volume snapshot creation process. The annotation holds a RFC3339 formatted timestamp that corresponds to the completion time of the associated backup job.	timestamp
<code>postgres-operator.crunchydata.com/pg</code>	Custom Resource	Specifies the hash value associated with a repo configuration as needed to detect configuration changes that invalidate running Jobs (and therefore must be recreated)	

backrest-hash			
postgres-operator.crunchydata.com/pgbackrest-ip-version	Custom Resource	Indicates whether to use an IPv6 wildcard address for the pgBackRest "tls-server-address". Set the value "IPv6" to use an IPv6 addresses. If the annotation is not present or has a value other than IPv6, it defaults to IPv4 (0.0.0.0).	0.0.0.0
postgres-operator.crunchydata.com/postgres-exporter-collectors	Pods	Specifies which collectors to enable for the exporter. The value "None" disables all postgres_exporter defaults. Disabling the defaults may cause errors in dashboards.	database, table
postgres-operator.crunchydata.com/adopt-bridge-cluster	CrunchyBridgeCluster Custom Resource	Allows users to "adopt" or take control over an existing Bridge Cluster with a CrunchyBridgeCluster Custom Resource. Essentially, if a CrunchyBridgeCluster Custom Resource does not have a status.ID, but the name matches the name of an existing bridge cluster, the user must add this annotation to the Custom Resource to allow it to take control of the Bridge Cluster. The Value assigned to the annotation must be the ID of existing cluster.	existing cluster ID
postgres-operator.crunchydata.com/automateCreateUserSchema	Custom Resource	Controls if the Operator should create schemas for the users defined in spec.users for all of the databases listed for that user	true
postgres-operator.crunchydata.com/authorizeBackupRemoval	Custom Resource	Allows removal of PVC-based backups when changing from a cluster with backups to a cluster without backups. Backups stored on the cloud storage are intact	true
postgres-operator.crunchydata.com/override-	ConfigMaps	Used to override default configuration from a ConfigMap.	custom-config

config			
pgv2.perc ona.com/monitor- user- secret- hash	Custom Resource	Hash of the monitor user secret, used to detect changes and trigger updates.	b6e1a2c3. ..
pgv2.perc ona.com/backup-in- progress	Custom Resource	Indicates a backup that is currently running for the cluster.	true
pgv2.perc ona.com/cluster- bootstrap- -restore	Custom Resource	Marks that the cluster was bootstrapped from a restore.	2024-07- 01T12:34: 56Z
pgv2.perc ona.com/patroni- version	Pods, StatefulSets	The Patroni version running in the Pod or StatefulSet.	4.1.0
pgv2.perc ona.com/custom- patroni- version	Pods, StatefulSets	Custom Patroni version specified by the user. Deprecated and ignored starting with version 2.8.0	3.3.0- percona
kubectl.k ubernetes .io/default- container	Pods	Defines a default container used when the -c flag is not passed when executing to a Pod.	

Setting labels and annotations in the Custom Resource

You can define both Labels and Annotations as `key-value` pairs in the metadata section of a YAML manifest for a specific resource.

Set labels and annotations for Pods

For PostgreSQL, pgBouncer and pgBackRest Pods, use

`instances.metadata.annotations/instances.metadata.labels`,
`proxy.pgouncer.metadata.annotations/proxy.pgouncer.metadata.labels`, or
`backups.pgbackrest.metadata.annotations/backups.pgbackrest.metadata.labels` keys as follows:

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



Set labels and annotations for Services

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

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



Set global labels and annotations

You can also use the top-level spec `metadata.annotations` and `metadata.labels` options to set annotations and labels at a global level, for all resources created by the Operator:

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



Settings labels and annotations for the Operator Pod

You can assign labels and/or annotations to the Operator itself by editing the [deploy/operator.yaml configuration file](#)  before [applying it during the installation](#). This way you add labels and annotations to the Pod where the Operator is running

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



Querying labels and annotations

To check which **labels** are attached to a specific object, use the additional `--show-labels` option of the `kubectl get` command.

For example, to see the Operator version associated with a Custom Resource Definition, use the following command:

```
kubectl get crd perconapgclusters.pgv2.percona.com --show-labels
```



 Sample output



perconapgclusters.pgv2.percona.com 2025-07-01T13:13:36Z pgv2.percona.com/version=v3.0.0 ``

To check **annotations** associated with an object, use the following command:

```
kubectl get <resource> <resource-name> -o jsonpath='{.metadata.annotations}'
```



For example, this command lists annotations assigned to a `cluster1-pgbouncer` Service:

```
kubectl get service cluster1-instance1-xvbt-0 -o jsonpath='{.metadata.annotations}'
```



Sample output



Special annotations

Metadata can be used as an additional way to influence the Operator behavior by setting special annotations.

Customizing Patroni version (for the Operator version 2.6.0 – 2.7.0)

Note

This behavior is deprecated and the annotation is ignored starting with version 2.8.0.

Starting from the Operator 2.6.0, Percona distribution for PostgreSQL comes with Patroni 4.x, which introduces breaking changes compared to previously used 3.x versions. To maintain backward compatibility, the Operator needs to detect the Patroni version used in the image. For this, it runs a temporary Pod named `cluster_name-patroni-version-check` with the following default resources:

```
Resources:
  Requests:
    memory: 32Mi
    cpu: 50m
  Limits:
    memory: 64Mi
    cpu: 100m
```



You can disable this auto-detection feature by manually setting the Patroni version via the following annotation in the metadata part of the Custom Resource (it should contain “4” for Patroni 4.x or “3” for Patroni 3.x):

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGCluster
metadata:
  name: cluster1
  annotations:
    pgv2.percona.com/custom-patroni-version: "4"
  ...
```



Transport encryption (TLS/SSL)

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.

You can configure TLS in these ways:

- Have the Operator generate certificates automatically during cluster creation,
- Use the [cert-manager](#) to manage certificates and their lifecycle,
- [Generate certificates manually](#).

You can [migrate your running cluster to cert-manager](#) to benefit from automatic renewal and centralized management.



Known limitation for Operator 2.9.0

If you had **cert-manager** installed and used the Operator-generated TLS certificates, upgrading the Operator to version **2.9.0** could put the cluster in an inconsistent state: the root CA stayed on internal PKI, but leaf certificates were handed to cert-manager. See [TLS and cert-manager \(Operator 2.9.0\)](#) for more information.

To fix this, either [move fully to cert-manager](#) or [switch to custom TLS Secrets](#). This issue is fixed in Operator **3.0.0** and later.

Additionally, you can *force* your database cluster to use only encrypted channels for both internal and external communications. To do this, set the `tlsOnly` Custom Resource option to `true`.

Automatic certificate generation by the Operator

The Operator can generate long-term certificates automatically and enable encryption automatically during cluster creation.

Upon cluster creation, the Operator reviews the Custom Resource configuration to determine the TLS approach:

- If you created custom certificate Secrets and referenced them in the cluster spec, the Operator uses them for TLS.
- If custom Secrets are not specified but [cert-manager](#) is installed, the Operator generates certificates and issuer and delegates certificate lifecycle management to cert-manager.
- If neither condition is met, the Operator generates the necessary certificates and Secrets itself.



Note

Beginning with version 2.5.0, the Operator creates a dedicated root CA for each cluster. Earlier versions used a single generated root CA for all database clusters.

TLS configuration

The following sections provide guidelines how to:

- [Configure TLS security with the Operator using cert-manager](#)
- [Generate certificates manually](#)
- [Migrate from Operator-generated certificates to cert-manager](#)
- [Migrate from cert-manager to custom TLS certificates](#)
- [Update certificates](#)
- [Check TLS communication to a cluster](#)

Keep certificates after deleting the cluster

When you [delete the cluster](#), the Operator handles SSL objects (Secrets, certificates, and issuer) as follows:

- The Operator doesn't delete TLS Secrets, certificates, and issuer it generated by default.
- The Operator removes the cert-manager Issuers and Certificates it created but keeps the Secrets.

If you want to clean up SSL objects, set the `finalizers.percona.com/delete-ssl` finalizer in the Custom Resource. The Operator deletes the all SSL objects.

Configure TLS security with the Operator using cert-manager

Starting with version 2.9.0, the Percona Operator for PostgreSQL integrates with [cert-manager](#)  for TLS certificate management.

When the Operator creates a database cluster, it checks if the cert-manager is installed and if you haven't provided custom TLS secrets. If these conditions are met, the Operator creates the self-signed Issuer resource within the cert-manager and requests a certificate from it. The cert-manager generates certificates and stores them in Kubernetes Secrets. The Operator uses these Secrets for TLS in the cluster. The cert-manager manages the certificate lifecycle.

The Percona Operator self-signed issuer is local to the Operator namespace. This self-signed issuer is created because Percona Distribution for PostgreSQL requires all certificates issued by the same CA (Certificate authority).

If cert-manager is not installed or not ready, the Operator falls back to its built-in certificate generation.

This approach gives you:

- **Automatic renewal** – cert-manager renews certificates before they expire (by default, 30 days before expiry)
- **Configurable validity** – you can set certificate and CA validity durations via Custom Resource options
- **Centralized management** – use cert-manager's tooling and policies for all TLS certificates in the cluster

Certificate lifecycle management

The cert-manager handles:

- **Issuance** – creates certificates when the cluster is created
- **Renewal** – renews certificates before expiry (default: 30 days before). You can configure the certificate duration in the Custom Resource
- **Rotation** – updates Secrets when certificates are renewed

The Operator does not renew certificates when using cert-manager; cert-manager does. You do not need to restart the cluster when certificates are renewed.

Prerequisites

To use cert-manager with the Operator, ensure the following:

1. You have deployed the Operator. Check if it runs with `kubectl get deploy -n <namespace>` command.
2. Your Custom Resource does **not** include these options:
 - `secrets.customTLSSecret`
 - `secrets.customReplicationTLSSecret`
 - `secrets.customRootCATLSSecret`

If you provide any of these, the Operator uses your custom certificates and does not create cert-manager resources.

Install cert-manager

Install cert-manager before deploying the Operator and cluster. You can use either kubectl or Helm.

By default the cert-manager is installed in the `cert-manager` namespace.

with kubectl

```
kubectl apply -f https://github.com/cert-manager/cert-manager/releases/download/v1.20.2/cert-manager.yaml
```



Expected output



with Helm

1. Add the Helm chart and update the repositories

```
helm repo add jetstack https://charts.jetstack.io --force-update  
helm repo update
```



2. Install cert-manager with default parameters:

```
helm install cert-manager jetstack/cert-manager \n  --namespace cert-manager \n  --create-namespace \n  --version v1.20.2 \n  --set crds.enabled=true
```



Expected output



You can customize the Helm installation by passing the TLS options via `values.yaml` or `--set`. See the [percona-helm-charts](#) repository for the available parameters.

Verify that cert-manager is running:

```
kubectl get pods -n cert-manager
```



 Expected output



Configure the certificate validity

1. Add the `tls` section to your Custom Resource to set certificate validity durations. These options apply only when cert-manager is used.

```
spec:
  tls:
    certValidityDuration: 2160h    # 90 days for TLS certificates (default: 8760h /
1 year)
    caValidityDuration: 26280h     # 3 years for the CA certificate (default: 8760h
/ 1 year)
    pgBackRestCertValidityDuration: 2160H # 90 days for TLS certificates
```



Use [Go duration format](#) (e.g. `2160h`, `8760h`).

2. Deploy the cluster:

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



Once you create the database with the Operator, it will automatically trigger the cert-manager to create certificates. Whenever you check certificates for expiration, you will find that they are valid and short-term.

Verify cert-manager resources

After the cluster is created, you can inspect the cert-manager resources:

```
# List Issuers
kubectl get issuers -n <namespace>

# List Certificates
kubectl get certificates -n <namespace>

# Check certificate status
kubectl get certificate <cluster-name>-cluster-ca-cert -n <namespace> -o yaml
```



The Operator creates Issuers and Certificates in the same namespace as the cluster. Secrets created by cert-manager follow the same naming as with built-in certificate generation (e.g. `<cluster-name>-cluster-ca-cert`, `<cluster-name>-cluster-cert`, `<cluster-name>-replication-cert`).

Related

If you want to stop using cert-manager for an existing cluster and use your own TLS Secrets instead, see [Migrate from cert-manager to custom TLS certificates](#).

Migrate from Operator-generated or custom certificates to cert-manager

You can start using cert-manager for TLS certificates lifecycle management if you have previously deployed your cluster with auto-generated or custom certificates. The use of cert-manager provides automatic certificate renewal, configurable validity periods, and centralized certificate management across your Kubernetes cluster.

Read more about cert-manager in [Configure TLS security with the Operator using cert-manager](#).

Migration steps

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

```
export CLUSTER=<cluster1>
export NAMESPACE=<namespace>
```



2. [Pause the cluster](#) to stop reconciliation before you change TLS resources. Run the following command to patch your running cluster:

```
kubectl patch -n $NAMESPACE pg $CLUSTER --type merge --patch '{
  "spec": {
    "pause": true}
}'
```



3. Verify the cluster status:

```
kubectl get pg $CLUSTER -n $NAMESPACE
```



Wait until the cluster status is `Paused`. You can verify with `kubectl get pods -n $NAMESPACE`.

4. Deploy cert-manager. In this guide we install cert-manager with `kubectl`. Refer to [Configure TLS security with the Operator using cert-manager](#) for Helm installation instructions.

```
kubectl apply -f https://github.com/cert-manager/cert-
manager/releases/download/v1.20.2/cert-manager.yaml
```



This command installs cert-manager in the default `cert-manager` namespace.

5. Verify that cert-manager is running:

```
kubectl get pods -n cert-manager
```



6. Delete the TLS Secrets.

Operator-generated certificates

a. Delete the Secrets:

```
kubectl delete secret $CLUSTER-cluster-ca-cert $CLUSTER-cluster-cert $CLUSTER-replication-cert -n $NAMESPACE
```

Example for a cluster named `cluster1`:

```
kubectl delete secret cluster1-cluster-ca-cert cluster1-cluster-cert cluster1-replication-cert -n $NAMESPACE
```

Custom certificates

a. Update the Custom Resource and remove all references to Secrets that contain your custom certificates:

```
kubectl -n $NAMESPACE patch pg $CLUSTER --type merge --patch '{
  "spec": {
    "secrets": {
      "customRootCATLSecret": null,
      "customTLSecret": null,
      "customReplicationTLSecret": null
    }
  }
}'
```

If `pgBouncer` also uses a custom certificate, clear the `spec.proxy.pgBouncer.customTLSecret` option too.

b. Delete the Secrets. This includes your custom Secrets and the one generated by the Operator:

```
kubectl delete secret <customRootCATLSecret-value> <customTLSecret-value> <customReplicationTLSecret-value> <pgbouncer.customTLSecret> -n $NAMESPACE
kubectl delete secret $CLUSTER-cluster-ca-cert $CLUSTER-cluster-cert $CLUSTER-replication-cert $CLUSTER-pgbouncer-frontend-tls -n $NAMESPACE
```

7. Resume the cluster. Set `spec.pause` back to `false`:

```
kubect1 patch -n $NAMESPACE pg cluster1 --type merge --patch '{
"spec": {
  "pause": false}
}'
```



The Operator detects the missing Secrets, sees cert-manager installed, requests new certificates from cert-manager, creates the Secrets, and resumes the cluster.

Verify the migration

After the cluster is running, verify that cert-manager resources were created:

```
kubect1 get issuers -n $NAMESPACE
kubect1 get certificates -n $NAMESPACE
kubect1 get secret <cluster-name>-cluster-ca-cert <cluster-name>-cluster-cert
<cluster-name>-replication-cert -n $NAMESPACE
```



You can configure certificate validity in the Custom Resource. See [Configure the certificate validity](#) for details.

Migrate from cert-manager to custom TLS certificates

When [cert-manager](#) is installed in the cluster and your PostgreSQL Custom Resource does **not** reference custom TLS Secrets, the Percona Operator for PostgreSQL creates cert-manager `Issuer` and `Certificate` resources and delegates certificate issuance and renewal to cert-manager when it creates a **new** cluster. The Operator does not perform its own built-in certificate rotation in that mode.

If you prefer **not** to use cert-manager for this cluster, you can instead supply Kubernetes Secrets with your own TLS material and point the Custom Resource at them under `spec.secrets`. This guide describes how to move an **existing** cluster from cert-manager-managed certificates to **custom** certificates **without rotating the cluster CA**, so clients and replicas that already trust the current CA keep working.

For background on cert-manager integration, see [Configure TLS security with the Operator using cert-manager](#). For certificate layout, Secret keys, and Common Name (CN) rules, see [Generate certificates manually](#).



Do not rotate the cluster CA in this migration

Re-sign **only** the PostgreSQL server and replication certificates with the **same** CA that already backs your cluster. Generating a new CA or changing the root CA requires updating trust everywhere and is out of scope for this procedure.

Prerequisites

- A running cluster that currently uses cert-manager (Secrets such as `<cluster-name>-cluster-ca-cert`, `<cluster-name>-cluster-cert`, and `<cluster-name>-replication-cert` exist in the cluster namespace).
- `kubectl` and `openssl` (or another PKI toolchain you use to issue TLS certificates).
- Permission to create Secrets and patch the `pg` Custom Resource in the namespace.

1. Identify the current CA

1. Set the cluster name and namespace (replace placeholders):

```
export CLUSTER=<cluster-name>
export NAMESPACE=<namespace>
```



2. Record the CA certificate fingerprint so you can confirm it is unchanged after migration. Use `root.crt` for the Operator 2.9.0 split state, or `tls.crt` if cert-manager already manages the CA:

```
kubectl get secret "${CLUSTER}-cluster-ca-cert" -n "${NAMESPACE}" \
  -o jsonpath='{.data.tls\.crt}' | base64 -d \
  | openssl x509 -fingerprint -sha256 -noout
```



3. Decode the CA certificate and private key to local files. You will use them only to sign new leaf certificates; you are **not** replacing the CA in the cluster trust store.

cert-manager or post-recovery Secret (`tls.crt` / `tls.key`)

```
kubectl get secret "${CLUSTER}-cluster-ca-cert" -n "${NAMESPACE}" \
  -o jsonpath='{.data.tls\.crt}' | base64 -d > cluster-ca.crt
kubectl get secret "${CLUSTER}-cluster-ca-cert" -n "${NAMESPACE}" \
  -o jsonpath='{.data.tls\.key}' | base64 -d > cluster-ca.key
```



Operator 2.9.0 split state (`root.crt` / `root.key`)

If the root CA Secret still uses internal-PKI key names after the 2.9.0 upgrade, extract the CA with:

```
kubectl get secret "${CLUSTER}-cluster-ca-cert" -n "${NAMESPACE}" \
  -o jsonpath='{.data.root\.crt}' | base64 -d > cluster-ca.crt
kubectl get secret "${CLUSTER}-cluster-ca-cert" -n "${NAMESPACE}" \
  -o jsonpath='{.data.root\.key}' | base64 -d > cluster-ca.key
```



Inspect the Secret with `kubectl get secret ... -o yaml` if you are unsure which key names it uses.

2. Generate new certificates with the same CA certificate

Issue **new** TLS certificates for:

- **External (client) traffic** — signed with the existing CA, with a CN (and Subject Alternative Names, if you use them) that match your primary Service. The usual form is `<cluster-name>-primary.<namespace>.svc.cluster.local`. Inspect the current certificate for reference:

```
kubectl get secret "${CLUSTER}-cluster-cert" -n "${NAMESPACE}" \
  -o jsonpath='{.data.tls\.crt}' | base64 -d | openssl x509 -text -noout
```



- **Replication** — signed with the **same** CA, with CN matching the replication user (by default `_crunchyrep1`).

Each resulting certificate must be paired with its private key. Follow the steps in [Generate certificates](#) for concrete `openssl` or `CFSSL` examples.

3. Create Kubernetes Secrets

Create generic Secrets that include `tls.crt`, `tls.key`, and `ca.crt` for each certificate. The `ca.crt` content in both Secrets must be the same. Use **new** Secret names so you do not overwrite active TLS Secrets before the Custom Resource is updated.

Example (adjust paths and names):

```
kubectl create secret generic "${CLUSTER}-custom-tls" -n "${NAMESPACE}" \
  --from-file=ca.crt=cluster-ca.crt \
  --from-file=tls.crt=server.crt \
  --from-file=tls.key=server.key

kubectl create secret generic "${CLUSTER}-custom-replication-tls" -n "${NAMESPACE}" \
  --from-file=ca.crt=cluster-ca.crt \
  --from-file=tls.crt=replication.crt \
  --from-file=tls.key=replication.key
```

Optionally, if you use a dedicated root CA Secret in the Custom Resource, create it as described in [Provide a pre-existing custom root CA certificate to the Operator](#). For many migrations it is enough that each leaf Secret includes the same `ca.crt`.

4. Update the PostgreSQL Custom Resource

At this stage, you need to update the cluster configuration to reference the new certificates. To do so, you must pause the cluster to prevent the Operator from automatically reconciling (making changes or reverting your edits) while you update the certificate references. Note that pausing the cluster will make it temporarily unavailable to users and applications, resulting in some planned downtime.

1. [Pause the cluster](#) so the Operator does not reconcile conflicting TLS state while you change references:

```
kubectl patch -n "${NAMESPACE}" "pg/${CLUSTER}" --type merge --patch '{
  "spec": { "pause": true }
}'
```

2. Point `spec.secrets` at your new Secrets (names must match what you created):

```
kubectl patch -n "${NAMESPACE}" "pg/${CLUSTER}" --type merge --patch "{
  \"spec\": {
    \"secrets\": {
      \"customTLSSecret\": { \"name\": \"${CLUSTER}-custom-tls\" },
      \"customReplicationTLSSecret\": { \"name\": \"${CLUSTER}-custom-replication-
tls\" }
    }
  }
}"
```

If you also set `customRootCATLSSecret`, use the manifest shape from [tls-manual.md](#).

3. Resume reconciliation:

```
kubectl patch -n "${NAMESPACE}" "pg/${CLUSTER}" --type merge --patch '{
  "spec": { "pause": false }
}'
```



The Operator reconciles the cluster and may restart or reload database Pods so they load the new Secrets.

5. Verify the migration

- **CA unchanged** — Compare the SHA-256 fingerprint of `ca.crt` inside your new Secrets (or the CA Secret you kept) with the fingerprint you recorded in step 1.
- **Server certificate** — Decode `tls.crt` from `${CLUSTER}-custom-tls` and confirm dates, SANs/CN, and issuer match expectations.
- **Replication certificate** — Confirm `${CLUSTER}-custom-replication-tls` is present, signed by the same CA, and uses the correct replication CN (`_crunchyrepl` by default).
- **Cluster status** — `kubectl get pg "${CLUSTER}" -n "${NAMESPACE}"` should return to **Ready** (and Pods healthy) after reconciliation.

For connection checks, see [Check TLS communication to a cluster](#).

Optional: remove cert-manager objects for this cluster

After the cluster runs on custom Secrets, you can delete cert-manager **namespaced** resources the Operator had created for it (for example `Certificate`, `CertificateRequest`, and `Issuer` in `${NAMESPACE}`). List them first:

```
kubectl get certificates,certificaterequests,issuers -n "${NAMESPACE}"
```



Delete only resources that belong to this cluster and are no longer required.

When **all** PostgreSQL clusters in the environment use custom TLS (or no longer need cert-manager), you may uninstall or scale down the cert-manager deployment. Coordinate with your platform team; other workloads may still depend on cert-manager.

Known limitation after removing cert-manager CRDs

If you delete cert-manager CustomResourceDefinitions from the cluster while the Percona Operator is still running, the Operator process may continue to register watches on cert-manager API types and log errors about missing resources.

Restart the Operator (for example, restart its Deployment in the Operator namespace) **after** the cert-manager CRDs are removed so a fresh process starts **without** those watches.

Ongoing lifecycle with custom certificates

With custom Secrets, **you** control rotation and expiry. The Operator does not renew those certificates for you. Plan updates using [Update TLS/SSL certificates](#).

Options under `spec.tls` that tune cert-manager validity apply only when cert-manager manages the cluster; they do not apply once you fully rely on custom Secrets for TLS material.

Generate certificates manually

You can customize TLS for the Operator by providing your own TLS certificates. To do this, you must create two Kubernetes Secret objects *before* deploying your cluster:

- One for external communication, later referenced by the `spec.secrets.customTLSSecret` field in the `deploy/cr.yaml`
- One for internal communication (used for replication authentication), referenced by the `spec.secrets.customReplicationTLSSecret` field in the `deploy/cr.yaml`.

Each Secret must contain the following fields:

- `tls.crt` (the TLS certificate)
- `tls.key` (the TLS private key)
- `ca.crt` (the Certificate Authority certificate)

Note that you cannot use only one custom set of certificates. If you provide a custom TLS Secret, you **must** also provide a custom replication TLS Secret, and both must contain the same `ca.crt`.

Provide existing custom certificates

For example, you have files named `ca.crt`, `my_tls.key`, and `my_tls.crt`. Run the following command to create a custom TLS Secret named `cluster1-tls`:

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



In the same way, create the custom TLS replication Secret, for example `replication1-tls`.

Next, reference your Secrets in the `deploy/cr.yaml` Custom Resource manifest as follows:

- add a Secret created for the external use to the `secrets.customTLSSecret.name` field
- add a Secret created for internal communications to the `secrets.customReplicationTLSSecret.name` field

Here's the sample configuration:

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



Now you can create a cluster with your custom certificates:

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



Provide a pre-existing custom root CA certificate to the Operator

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

To make the Operator use a custom root certificate, create a separate secret with this certificate and specify this secret in the Custom Resource options **before** you deploy a cluster.

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

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



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

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



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

Warning

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

Generate custom certificates for the Operator yourself

Understand certificate requirements

To find out the certificates specifics needed for the Operator, view the certificates generated by the Operator automatically. For example, if you have a cluster deployed in some staging environment.

Here's how to do it:

1. Check the secrets created by the Operator:

```
kubectl get secrets
```



Expected output



The Secrets of interest are `cluster1-cluster-cert` for external communication and `cluster1-replication-cert` for internal communication.

2. You can examine the auto-generated CA certificate (`ca.crt`) as follows:

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



Expected output



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

External communication

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

 Expected output



Internal communication

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

 Expected output



Both secrets share the same `ca.crt` certificate but have different `tls.crt` certificates. The `tls.crt` in the Secret for external communications should have a Common Name (CN) setting that matches the primary Service name (`CN = cluster1-primary.default.svc.cluster.local` in the above example). Similarly, the `tls.crt` in the Secret for internal communications should have a Common Name (CN) setting that matches the preset replication user: `CN=_crunchyrep1`.

Generate certificates

One of the options to create certificates yourself is to use [CloudFlare PKI and TLS toolkit](#).

You must generate certificates twice: one set is for external communications, and another set is for internal ones!

Let's say that your cluster name is `cluster1` and the desired namespace is `postgres-operator`. The commands to generate certificates may look as follows:

1. Set cluster context

```
export CLUSTER_NAME=cluster1
export NAMESPACE=postgres-operator
```

2. Generate the root CA certificate:

```
cat <<EOF | cfssl gencert -initca - | cfssljson -bare ca
{
  "CN": "*",
  "key": {
    "algo": "ecdsa",
    "size": 384
  }
}
EOF
```



Expected output



You should have the following files:

- ca-key.pem – CA private key
- ca.pem – CA certificate

3. Define the CA signing policy for certificates signed by the CA.

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



Explanation of the values:

- `expiry` - sets the lifetime for the certificates
- `usages` specifies what the certificate is valid for:
 - `digital signature`: for signing data
 - `key encipherment`: for secure key exchange
 - `content commitment`: ensures data integrity
- Generate the custom TLS certificates for external communication and sign them using the previously created CA certificate. These certificates have the Common Name (CN) `cluster1-primary.postgres-operator.svc.cluster.local`

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

You should have the following files as defined by the `-bare server` part of the command:

- `server.pem` - the signed certificate
- `server-key.pem` - the private key
- Generate the custom TLS certificates for internal communication and sign them using the previously created CA certificate. These certificates have the Common Name (CN) `_crunchyrepl`.

```
cat <<EOF | cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=./ca-config.json
| cfssljson -bare replication
{
  "CN": "_crunchyrepl",
  "key": {
    "algo": "ecdsa",
    "size": 384
  }
}
EOF
```

You should have the following files as defined by the `-bare` part of the command:

- `replication.pem` - the signed certificate
- `replication-key.pem` - the private key

You can find more on generating certificates this way in [official Kubernetes documentation](#).

Refer to the [Provide pre-existing custom certificates](#) section for the steps to create Secrets and configure the Operator. Replace the values with your files.

Update certificates

How your TLS certificates are updated depends on how they were created:

- Certificates generated by the Operator are long-term. The Operator automatically updates the automatically-generated certificates to ensure your applications continue operation without communication issues.
- Certificates issued by the cert-manager are short-term. You can configure their validity in the Custom Resource when you deploy a cluster. The cert-manager automatically reissues the certificates on schedule and without downtime.
- Certificates you generated manually or provided to the Operator are out of its control. The Operator doesn't update custom certificates. It is your responsibility to timely update them.

This document focuses on how to update manually generated certificates.

Check your certificates for expiration

1. List the Secrets

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



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

You will have the following response:

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

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

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



The resulting output will be self-explanatory:

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

Update custom certificates

You can update only custom certificates for external and / or internal communication and keep the same root CA certificate.

You can update the contents of your existing Secrets referenced in the `spec.secrets.customTLSSecret` and/or `spec.secrets.customReplicationTLSSecret` fields in `deploy/cr.yaml` without changing their names. In this case, the Operator detects the updated certificate data and applies the changes to the running cluster without restarting it. Such update is called hot reload.

This example shows how you can do it. Let's say you have the following certificates and Secrets:

- `server.pem` / `server-key.pem` and the `cluster1-cert` Secret for external communication,
- `replica.pem` / `replica-key.pem` and `cluster1-replication-cert` Secret for internal communication
- `ca.pem` / `ca-key.pem` is the existing CA root certificate that you keep

Your cluster is deployed in the `postgres-operator` namespace.

1. Set the context for the cluster:

```
export NAMESPACE=postgres-operator
```



2. Create a YAML manifest for the `cluster1-cert` Secret. Run the following command to generate a YAML manifest (adjust file paths if needed):

```
kubectl create secret generic cluster1-cert \
  --from-file=tls.crt=server.pem \
  --from-file=tls.key=server-key.pem \
  --from-file=ca.crt=ca.pem \
  -n "$NAMESPACE" \
  --dry-run=client -o yaml > cluster1-cert.yaml
```



3. Create a YAML manifest for the `cluster1-replication-cert` Secret. Run the following command to generate a YAML manifest (adjust file paths if needed):

```
kubectl create secret generic cluster1-replication-cert \
  --from-file=tls.crt=replica.pem \
  --from-file=tls.key=replica-key.pem \
  --from-file=ca.crt=ca.pem \
  -n "$NAMESPACE" \
  --dry-run=client -o yaml > cluster1-replication-cert.yaml
```

4. Apply the manifests to update the Secrets:

```
kubectl apply -f cluster1-cert.yaml -f cluster1-replication-cert.yaml -n
"$NAMESPACE"
```

If you create new Secrets with new names and values, update the `spec.customTLSSecret` and `spec.customReplicationTLSSecret` fields in the `deploy/cr.yaml`. When you apply the new configuration, this causes the Operator to restart the cluster.

Update a custom root CA certificate

Here's what you need to know if you wish to update a custom root CA certificate:

- If you change a root CA certificate, you must also change your custom TLS certificates for external and internal communications as these must be signed with the same root CA.
- The new root CA and associated certs must be stored in new Secrets (not overwriting existing ones). This ensures rollback capability in case of misconfiguration or validation issues.
- You must [pause the cluster](#) before applying changes. This prevents the Operator from restarting or reconfiguring Pods mid-update.

To update a custom root CA certificate, do the following:

1. Generate a new root CA certificate and key. For example, you have them in files named `new-ca.pem` and `new-ca-key.pem`.
2. Generate all dependent certificates for external and internal communication and sign them using the new root CA certificate. Check the [Generate certificates manually](#) section for the steps. For example, you end up with the following certificates:
 - `server.pem` and `server-key.pem` for external communication
 - `replication.pem` and `replication-key.pem` for internal communication
3. Create a new Secret object for the new root CA certificate and define the new CA certificate and key within. Let's name it `cluster1-ca-cert-new`.

```
kubectl create secret generic -n postgres-operator cluster1-ca-cert-new \
  --from-file=ca.crt=new-ca.pem \
  --from-file=ca.key=new-ca-key.pem
```



4. Create new Secrets for external and internal communications, named `cluster1-tls` and `cluster1-replication-tls` respectively

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



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



5. Pause the cluster to prevent the Operator to restart the Pods mid-update.

```
kubectl patch pg cluster1 \
  --type merge \
  --patch '{"spec": {"pause": true}}' \
  --namespace postgres-operator
```



6. Specify details about new custom certificates in the `deploy/cr.yaml`. Since this is a provisioned cluster, apply the patch as follows:



```
kubectl patch pg cluster1 \
  --type merge \
  --patch '{
    "spec": {
      "secrets": {
        "customRootCATLSSecret": {
          "name": "cluster1-ca-cert-new",
          "items": [
            {
              "key": "ca.crt",
              "path": "root.crt"
            },
            {
              "key": "ca.key",
              "path": "root.key"
            }
          ]
        },
        "customTLSSecret": {
          "name": "cluster1-tls"
        },
        "customReplicationTLSSecret": {
          "name": "cluster1-replication-tls"
        }
      }
    }
  }' \
  --namespace postgres-operator
```

7. Unpause the cluster to resume the Operator control:



```
kubectl patch pg cluster1 \
  --type merge \
  --patch '{"spec": {"pause": false}}' \
  --namespace postgres-operator
```

Check TLS communication to the cluster

You can check TLS communication using `psql`, the standard interactive terminal-based frontend to PostgreSQL.

For this purpose, we will create a `pg-client` Deployment, which includes the necessary tools. We will use the existing Secret object with the TLS certificates, generated by the Operator.

Follow these steps:

1. Export the namespace as the environment variable to simplify further configuration:

```
export NAMESPACE=<postgres-operator>
```



2. List the Secret objects:

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



Expected output



The secret with TLS certificates is `<cluster-name>-cluster-ca-cert` (`cluster1-cluster-ca-cert` by default).

3. Create a deployment. Replace the placeholders in the following command with your values:

- Replace the `<cluster-name>` placeholder with your actual cluster name
- Specify the Secret object with the TLS certificate from step 2
- Specify the CA certificate file for the `volumeMounts.name` and `volumes.name` options. This file is used to mount the root CA certificate into the container so that client tools like `psql` can verify the server's certificate. To view the file name, run `kubectl get secret <cluster-name>-cluster-ca-cert -o yaml -n $NAMESPACE` command.



```
cat <<EOF | kubectl apply -n $NAMESPACE -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: percona/percona-distribution-postgresql:18.4-1
          imagePullPolicy: Always
          command:
            - sleep
          args:
            - "100500"
          volumeMounts:
            - name: root
              mountPath: "/tmp/tls"
      volumes:
        - name: root
          secret:
            secretName: <cluster-name>-cluster-ca-cert
            items:
              - key: root.crt
                path: root.crt
                mode: 0777
EOF
```

4. Retrieve the `pgBouncer` URI to connect to PostgreSQL. It is stored in the Secret object with user credentials: `<cluster-name>-pguser-<db-name>`. The default value is `cluster1-pguser-cluster1`. Run the following command and replace the `cluster1-pguser-cluster1` Secret name with your value:



```
kubectl get secret cluster1-pguser-cluster1 -o jsonpath='{.data.pgouncer-uri}' -n $NAMESPACE | base64 --decode
```

 **Sample output**



5. Now get shell access to the newly created container:



```
kubectl exec -it deployment/pg-client -- bash -il
```



Expected output



6. Launch the `psql` interactive terminal to check connectivity over the encrypted channel. Make sure to use your cluster name, the pgBouncer URL you retrieved at the previous step, and the CA certificate you provided when you created the `pg-client` deployment:

```
PGSSLMODE=verify-ca PGSSLROOTCERT=/tmp/tls/root.crt psql 'postgresql://cluster1:  
<password>@cluster1-pgbouncer.default.svc:5432/cluster1'
```

Now you should see the prompt of PostgreSQL interactive terminal:

```
psql (18.4-1)  
Type "help" for help.  
cluster1=>
```



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 3.0.0 --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 `kubect1 apply -f deploy/operator.yaml` command. Open the `operator.yaml` file with your text editor, find the `DISABLE_TELEMETRY` environment variable and set it to `"true"`

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



Configure concurrency for a cluster reconciliation

Reconciliation is the process by which the Operator continuously compares the desired state with the actual state of the cluster. The desired state is defined in a Kubernetes custom resource, like `PostgresCluster`.

If the actual state does not match the desired state, the Operator takes actions to bring the system into alignment—such as creating, updating, or deleting Kubernetes resources (Pods, Services, ConfigMaps, etc.) or performing database-specific operations like scaling, backups, or failover.

Reconciliation is triggered by a variety of events, including:

- Changes to the cluster configuration
- Changes to the cluster state
- Changes to the cluster resources

By default, the Operator has one reconciliation worker. This means that if you deploy or update 2 clusters at the same time, the Operator will reconcile them sequentially.

The `PGO_WORKERS` environment variable in the `percona-postgresql-operator` deployment controls the number of concurrent workers that can reconcile resources in PostgreSQL clusters in parallel.

Thus, to extend the previous example, if you set the number of reconciliation workers to `2`, the Operator will reconcile both clusters in parallel. This also helps you with benchmarking the Operator performance.

The general recommendation is to set the number of concurrent workers equal to the number of PostgreSQL clusters. When the number of workers is greater, the excessive workers will remain idle.

Set the number of reconciliation workers

1. Check the index of the `PGO_WORKERS` environment variable using the following command:

```
kubectl get deployment percona-postgresql-operator -o  
jsonpath='{.spec.template.spec.containers[0].env[?(@.name=="PGO_WORKERS")].value}'
```

 Sample output



2. List deployments to find the right one:

```
kubectl get deployment
```





Sample output



3. To set a new value, run the following command to patch the deployment:

```
kubectl patch deployment percona-postgresql-operator \
  --type='json' \
  -p='[{"op": "replace", "path": "/spec/template/spec/containers/0/env/5",
"value": {"name": "PGO_WORKERS", "value": "2"}}]'
```



The command does the following:

- Patches the deployment to update the `PGO_WORKERS` environment variable
- Sets the value to `2`

The value can be set to any number greater than 0.

Verify the change

To verify that the change has been applied, run the following command:

```
kubectl get deployment percona-postgresql-operator -o
jsonpath='{.spec.template.spec.containers[0].env[?(@.name=="PGO_WORKERS")].value}'
```



The output should be `2`.

Environment variables

Define environment variables

You can configure environment variables in Percona Operator for PostgreSQL for the following purposes:

- 1. **Operator environment variables** – To control the Operator’s behavior, such as logging, telemetry, and which namespaces it watches. You set these in the Operator Deployment (for example in `deploy/bundle.yaml` or via the [Helm chart](#) values).
- 2. **Cluster component environment variables** – To customize the behavior of cluster components (PostgreSQL, pgBackRest, pgBouncer). You define these in your cluster Custom Resource and, when needed, store sensitive values in [Kubernetes Secrets](#).
- 3. **Backup and restore environment variables** – To customize backup and restore flows. You can specify the environment variables either directly in the backup / restore manifest, or reference a [Kubernetes Secret](#) that stores sensitive values.

When to use environment variables

Type	Use cases
Operator environment variables	- Control logging for debugging and log aggregation - Manage telemetry - Configure which namespaces the Operator watches (single-namespace vs cluster-wide) - Set the number of concurrent reconciliation workers for multi-cluster environments - Enable feature gates such as auto-growable volumes
Cluster component environment variables	- Customize PostgreSQL, pgBackRest, or pgBouncer behavior - Pass configuration or secrets into Pods - Integrate with external systems or monitoring.
Backup and restore environment variables	- Customize backup and restore operations - Pass configuration or secrets into backup / restore Jobs

Next steps

Configure Operator environment variables

Environment variables for cluster components

Configure the Operator environment variables

You can configure the Percona Operator for PostgreSQL behavior by setting environment variables in the Operator Deployment. You can set them when you install the Operator in the following ways:

- For installations via `kubectl`, edit the Operator Deployment manifest `deploy/operator.yaml` or `deploy/cw-operator.yaml`. Alternatively you can modify the Deployment resource in `deploy/bundle.yaml`, or `deploy/cw-bundle.yaml` files.
- For Helm installations you can set environment variables through Helm values when you install the `percona/pg-operator` chart.
- For installations on OpenShift, you can edit the manifests and apply them with the `oc apply` command. If you installed via the [Operator Lifecycle Manager \(OLM\)](#), you can configure environment variables through the OLM subscription.

Available environment variables

LOG_LEVEL

Controls the verbosity of the Operator’s logging output. This helps with debugging and monitoring the Operator behavior.

Value type	Default	Example
string	"INFO"	"DEBUG"

Accepted values are:

- "DEBUG" – Most verbose, includes detailed debugging information
- "INFO" – Standard informational messages (default)
- "WARN" – Warning messages only
- "ERROR" – Error messages only

Example configuration:

```
spec:
  containers:
    - name: percona-postgresql-operator
      env:
        - name: LOG_LEVEL
          value: "DEBUG"
```



DISABLE_TELEMETRY

Controls whether the Operator sends anonymous telemetry data to Percona. Telemetry helps Percona understand usage patterns and improve the Operator.

Value type	Default	Example
string	"false"	"true"

When set to `true`, the Operator does not send anonymous telemetry data to Percona.

Learn more about [Telemetry](#).

Example configuration:

```
spec:
  containers:
    - name: percona-postgresql-operator
      env:
        - name: DISABLE_TELEMETRY
          value: "true"
```

PGO_FEATURE_GATES

Enables experimental or advanced features in the Operator. Feature gates allow you to opt into specific functionality that may not be enabled by default.

The value is a comma-separated list of feature gate key-value pairs. By default this variable is not set in the Operator.

Value type	Default	Example
string	" " (empty)	"AutoGrowVolumes=true"

Following feature gates are present:

- `AutoGrowVolumes=true` - Enables automatic PVC resize when the storage usage reaches a threshold. The Operator can trigger volume expansion for database data volumes. To learn more, refer to the [Scale your cluster](#) chapter.
- `BackupSnapshots=true` - Enables [PVC snapshot support](#) for backups and restores. When enabled and configured in the cluster Custom Resource, the Operator creates volume snapshots in coordination with pgBackRest backups, enabling much faster backups and restores for large datasets. Available as of Operator version 2.9.0.

Example configuration:

```
spec:
  containers:
    - name: percona-postgresql-operator
      env:
        - name: PGO_FEATURE_GATES
          value: "AutoGrowVolumes=true"
```

To enable multiple features, list them separated by a comma:

```
- name: PGO_FEATURE_GATES
  value: "AutoGrowVolumes=true,BackupSnapshots=true"
```

LOG_STRUCTURED

Controls whether the Operator outputs logs in a structured JSON format instead of the plain text. Structured logging is useful for log aggregation tools.

Value type	Default	Example
string	"false"	"true"

When set to "true", the logs are produced in JSON format. When set to "false" or not set at all, the Operator outputs plain text logs. This is the default behavior.

Example configuration:

```
spec:
  containers:
    - name: percona-postgresql-operator
      env:
        - name: LOG_STRUCTURED
          value: "true"
```

PGO_WORKERS

Specifies the number of worker threads the Operator uses to process events and reconcile resources. This controls the Operator’s concurrency. Default value is 1.

Value type	Default	Example
string	"1"	"2"

Keep in mind that concurrent reconciliations are done only on different objects. For the same object reconciliation is always done serially, regardless of the value set in `PGO_WORKERS` . This is defined by how the controller runtime works with the queue to avoid any race conditions or incorrect modification of objects.

To illustrate how it works:

- If you have two `PerconaPGCluster` objects (A and B) and set `PGO_WORKERS=1` , a single worker thread will reconcile the clusters serially, one after another.
- If you set `PGO_WORKERS=4` but only have one `PerconaPGCluster` object, the Operator still reconciles this object serially.
- If you set `PGO_WORKERS=4` and have two `PerconaPGCluster` objects (A and B), the Operator uses two separate threads to reconcile each object in parallel; however, it always processes events for each individual object sequentially.

Example configuration:

```
spec:
  containers:
    - name: percona-postgresql-operator
      env:
        - name: PGO_WORKERS
          value: "4"
```

WATCH_NAMESPACE

Specifies which namespaces the Operator watches for Custom Resources (`PerconaPGCluster` and related resources). This is a critical configuration for determining the Operator’s scope of operation.

By default, the value is set to the Operator’s own namespace from the `metadata.namespace` option via a downward API `fieldRef`:

```
- name: WATCH_NAMESPACE
  valueFrom:
    fieldRef:
      apiVersion: v1
      fieldPath: metadata.namespace
```

Value type	Default	Example
string	Operator’s namespace (from <code>fieldRef</code>)	<code>"pg-operator, percona-db-1"</code> or <code>" "</code>

Accepted values:

- If set to a comma-separated list, the Operator watches those specific namespaces. The namespace list

must include the namespace where the Operator itself is deployed. Use this approach for the [cluster-wide mode](#).

- If set to an empty string (""), the Operator watches all namespaces in the Kubernetes cluster (cluster-wide mode).

When you deploy the Operator in cluster-wide mode, it should be associated with the appropriate ClusterRole.

Example configuration:

```
spec:
  containers:
    - name: percona-postgresql-operator
      env:
        - name: WATCH_NAMESPACE
          value: "pg-operator,percona-db-1,percona-db-2"
```

PGO_NAMESPACE

Specifies the Kubernetes namespace where the Operator itself is deployed and runs. The value is set automatically by Kubernetes from `metadata.namespace` via a downward API `fieldRef`.

This variable is used by the Operator to refer objects like Secrets for the normal functioning of the Operator.

This is particularly important in [cluster-wide](#) deployment scenarios where the Operator manages resources across multiple namespaces.

Example configuration:

```
spec:
  containers:
    - name: percona-postgresql-operator
      env:
        - name: PGO_NAMESPACE
          value: "pg-operator"
```

PGO_CONTROLLER_LEADER_ELECTION_ENABLED

Controls whether the Operator uses leader election. Leader election ensures only one Operator instance manages resources when multiple replicas run. Set to `"false"` to disable leader election (single-replica deployments only).

Value type	Default	Example
string	"true"	"false"

Example configuration:

```
spec:
  containers:
    - name: percona-postgresql-operator
      env:
        - name: PGO_CONTROLLER_LEADER_ELECTION_ENABLED
          value: "false"
```

PGO_CONTROLLER_LEASE_NAME

Specifies the name of the Lease resource used for the leader lock. Leave empty to use the default Lease `08db3feb.percona.com` .

Value type	Default	Example
string	<code>" "</code> (empty)	<code>"my-lease"</code>

Example configuration:

```
spec:
  containers:
    - name: percona-postgresql-operator
      env:
        - name: PGO_CONTROLLER_LEASE_NAME
          value: "my-lease"
```

PGO_CONTROLLER_LEASE_DURATION

Duration that non-leader candidates wait before forcing leader acquisition. This is measured against the time of last observed acknowledgment. Uses Go duration format (for example, `60s`). You can increase this value if the Operator experiences leader election failures in high-latency or resource-constrained environments.

Value type	Default	Example
string	<code>"60s"</code>	<code>"90s"</code>

Example configuration:

```
spec:
  containers:
  - name: percona-postgresql-operator
    env:
    - name: PGO_CONTROLLER_LEASE_DURATION
      value: "90s"
```



PGO_CONTROLLER_RENEW_DEADLINE

Duration that the acting leader retries refreshing the lease before giving up. Uses Go duration format (for example, 60s).

Value type	Default	Example
string	"40s"	"60s"

Example configuration:

```
spec:
  containers:
  - name: percona-postgresql-operator
    env:
    - name: PGO_CONTROLLER_RENEW_DEADLINE
      value: "60s"
```



PGO_CONTROLLER_RETRY_PERIOD

Duration between leader election retry attempts. Uses Go duration format (for example, 60s).

Value type	Default	Example
string	"10s"	"15s"

Example configuration:

```
spec:
  containers:
  - name: percona-postgresql-operator
    env:
    - name: PGO_CONTROLLER_RETRY_PERIOD
      value: "15s"
```



PPROF_BIND_ADDRESS

Specifies the TCP address that the controller binds to for serving pprof profiling endpoints. Use this when you need to collect CPU or memory profiles or investigate Operator performance issues.

Value type	Default	Example
string	"0" (disabled)	"127.0.0.1:6060"

When set to "" or "0", pprof serving is disabled. Set it to an address such as 127.0.0.1:6060 to enable profiling. You can then use `kubectl port-forward` to access the pprof endpoints from your local machine.

See [Profiling the Operator with pprof](#) for usage instructions.

Example configuration:

```
spec:
  containers:
    - name: percona-postgresql-operator
      env:
        - name: PPROF_BIND_ADDRESS
          value: "127.0.0.1:6060"
```

Update environment variables

Using kubectl patch

You can update environment variables in an existing Operator Deployment by applying a patch. To keep existing variables, include the full list in your patch.

1. Get the current environment variables:

```
kubectl get deployment percona-postgresql-operator -o
jsonpath='{.spec.template.spec.containers[0].env}'
```

2. Edit the output to add or change a variable (for example PGO_WORKERS), then apply a patch with the full env list. Alternatively, patch a single entry by index (see [Configure concurrency for a cluster reconciliation](#)).

Using Helm

For Helm installations, set or change environment variables through Helm values (for example logLevel, logStructured, disableTelemetry, watchNamespace, watchAllNamespaces). Refer to the [pg-operator chart](#) documentation for the exact value names and syntax. To add variables not exposed by the chart, use a chart value that merges extra env entries if supported, or switch to patching the Deployment after install.

After the update

After you change environment variables, the Operator Pod is restarted so the new configuration takes effect.

Define environment variables for cluster components

To pass environment variables into PostgreSQL, pgBackRest, or pgBouncer Pods, use the Custom Resource options described in [Custom Resource options](#). For example:

- **PostgreSQL instances:** use `spec.instances[].env`, `spec.instances[].envFrom`, or reference a Secret via `spec.instances[].envVarsSecret`.
- **pgBackRest:** use the relevant `spec.backups.pgbackrest.*` or restore job options and any supported env or Secret references.
- **pgBouncer:** use `spec.proxy.pgouncer.env`, `spec.proxy.pgouncer.envFrom`, or `spec.proxy.pgouncer.envVarsSecret`.

Sensitive values (passwords, API keys) should be stored in [Kubernetes Secrets](#) and referenced by name in the cluster spec. For more options, see [Secrets options](#) and the [Custom Resource options](#) reference.

Management

Back up and restore

About backups

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

You can make backups in two ways:

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

What you need to know

Backup methods

By default, the Operator uses the open source [pgBackRest](#)  backup and restore utility to make backups.

Starting with version 2.9.0, the Operator also supports [PVC snapshots](#) - storage-level copies of your data volumes. PVC snapshots offer much faster backups and restores. When used with `pgBackRest` WAL archiving, they maintain data consistency and enable you to make point-in-time recovery of your database.

This feature is in the tech preview stage. You must explicitly enable it for the Operator deployment.

Backup repositories

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

A `pgBackRest` repository consists of the following Kubernetes objects:

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

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

Backup types

You can make the following types of backups:

- `full`: A full backup of all the contents of the PostgreSQL cluster,

- `differential`: A backup of only the files that have changed since the last full backup,
- `incremental`: Default. A backup of only the files that have changed since the last full or differential backup.

Backup storage

You have the following options to store PostgreSQL backups:

- Cloud storage:
 - [Amazon S3](#) or any S3-compatible storage
 - [Google Cloud Storage](#)
 - [Azure Blob Storage](#)
- A [Persistent Volume](#) attached to the pgBackRest Pod.

Next steps

Ready to move forward? [Configure backup storage](#)

Configure storage for backups

Configure storage for backups

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

You can use the following storage types with the Operator:

- **Amazon S3 or S3-compatible storage** — Use native AWS S3 buckets on Amazon EKS or any other Kubernetes platform. Or, use compatible services like MinIO and others that expose an S3-compatible API. On EKS, you can authenticate with [S3 access keys](#) or [IAM roles for Service Accounts](#).
- **Google Cloud Storage** — Use Google Cloud object storage.
- **Azure Blob Storage** (tech preview) — Use Microsoft Azure Blob Storage.
- **Persistent Volume** — Use a [persistent volume attached to the pgBackRest Pod](#) that is created together with the PostgreSQL cluster.

For cloud object storage, you typically create a [Kubernetes Secret](#)  with credentials and reference it in `backups.pgbackrest.configuration`. If you use IAM roles on Amazon EKS, you don't need to create a Secret.

Storage setup guides

Select the appropriate guide below based on your chosen storage type:

- [Amazon S3 storage](#)
- [Google Cloud Storage](#)
- [Microsoft Azure Blob Storage](#)
- [Persistent Volume](#)

Next steps

[Make an on-demand backup](#)

[Make a scheduled backup](#)

Amazon S3 storage

To use [Amazon S3](#) for backups, you need to have the following information:

- S3 bucket name
- AWS region where the bucket is located. Check the [Region considerations for pgBackRest](#)
- Authentication to the bucket. You can use one of the following options:
 - **S3 access keys.** Store your AWS S3 access key and secret key in a Kubernetes Secret and reference it in your cluster configuration. This method works on any Kubernetes platform and requires that you manage the credentials yourself.
 - **IAM role for Service Accounts (IRSA).** Assign an IAM role to the Kubernetes Service Account used by pgBackRest. With IRSA, you don't need to manage static secrets since AWS automatically provides temporary credentials to the Pod, following best practices for security and rotation.
- For S3-compatible storage other than native Amazon S3, also specify:
 - The endpoint - the actual URI to access the bucket
 - The URI style — determines how the Operator accesses the bucket. For most S3-compatible providers, you must set the URI style to `path`. This is required for correct operation with most S3-compatible storage endpoints.

Region considerations for pgBackRest

pgBackRest archives write-ahead logs (WAL) to backup storage. If the bucket is in a region far from your PostgreSQL cluster, archiving can lag: a new WAL file must be archived within [60 seconds by default](#), and long delays can cause backup and replica issues when the primary restarts. Use the same AWS region for the cluster and the bucket when possible, or replicate the bucket with [Amazon S3 Cross-Region Replication](#).

Configuration steps

Select your authentication method below. If you are using S3-compatible storage, follow the “S3 access keys” instructions.

The examples in the following sections use the `repo2` repository name. Use the same name consistently for Secrets, `global` settings, and `repos` in your configuration.

S3 access keys

Follow these steps to authenticate using an AWS S3 access key and secret key. This method works on any Kubernetes environment.

- 1 Encode the S3 credentials and the pgBackRest repository name that you will use for backups. The following example uses `repo2`:



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



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

- 2 Create the Secret manifest and set `data.s3.conf` to the encoded string from the previous step. The following is the example of the `cluster1-pgbackrest-secrets.yaml` configuration file:

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

You can store credentials for several repositories in one Secret by adding separate data keys.

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

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

- 4 Update `deploy/cr.yaml` Custom Resource manifest:

- Reference the Secret in `backups.pgbackrest.configuration`
- Provide the backup path in [backups.pgbackrest.global](#) with the pgBackRest `path` option. Specify `repo2-path`.
- For S3-compatible storage:
 - Set `global.repo2-storage-verify-tls` to `y` to enable TLS verification

- Set the [global.repo2-s3-uri-style option](#) to `path`. Without it, backup jobs may log a "host not found" error.
- Specify additional [repository options](#) in `global`, if your S3-compatible storage requires them.
- Define the S3 bucket, region and endpoint under `repos`

Amazon S3 storage

```
...
backups:
  pgbackrest:
    ...
    configuration:
      - secret:
          name: cluster1-pgbackrest-secrets
    ...
    global:
      repo2-path: /pgbackrest/postgres-operator/cluster1/repo2

    ...
    repos:
      - name: repo2
        s3:
          bucket: "<YOUR_AWS_S3_BUCKET_NAME>"
          region: "<YOUR_AWS_S3_REGION>"
```

S3-compatible storage

```
...
backups:
  pgbackrest:
    ...
    configuration:
      - secret:
          name: cluster1-pgbackrest-secrets
    ...
    global:
      repo2-path: /pgbackrest/postgres-operator/cluster1/repo2
      repo2-storage-verify-tls: "y"
      repo2-s3-uri-style: path
    ...
    repos:
      - name: repo2
        s3:
          bucket: "<YOUR_S3_BUCKET_NAME>"
          endpoint: "<YOUR_S3_ENDPOINT>"
          region: "<YOUR_S3_REGION>"
```

5 Apply the cluster Custom Resource:

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



IAM role for Service Account

On [Amazon EKS](#), you can grant the Operator access to S3 with [IAM roles for Service Accounts \(IRSA\)](#). pgBackRest authenticates through the pod identity; therefore, you don't need to create a Secret with s3 credentials and reference it in the Custom Resource.

- 1 Create an IAM role for your EKS OIDC provider and attach a policy that allows the required S3 bucket access. See the [official AWS documentation](#) for details.
- 2 Annotate the Custom Resource with the IAM role Amazon Resource Name (ARN) and set `repo2-s3-key-type` to `web-id` in `backups.pgbackrest.global`:

```
spec:
  crVersion: 3.0.0
  metadata:
    annotations:
      eks.amazonaws.com/role-arn: arn:aws:iam::1191:role/role-pgbackrest-access-s3-bucket
    ...
  backups:
    pgbackrest:
      image: percona/pgbackrest:2.58.0-2
      global:
        repo2-path: /pgbackrest/postgres-operator/cluster1/repo2
        repo2-s3-key-type: web-id
```



3 Apply the cluster Custom Resource:

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



If you configure both IRSA and S3 keys in a Secret, the Secret credentials take precedence for authentication.

Next steps

Make an on-demand backup

Make a scheduled backup

Google Cloud Storage

To use [Google Cloud Storage \(GCS\)](#) for backups, configure a pgBackRest repository in `backups.pgbackrest.repos` and provide:

- A GCS bucket name which you will reference in the `gcs.bucket` option under the repository entry
- A Google service account key, which the Operator will use to access the storage.

Configuration steps

The following example configuration uses the `repo3` repository name.

1. Create a service account key following the [Google Cloud instructions](#).
2. Download the key from the Google Cloud console (*IAM & Admin* → *Service Accounts* → your account → *KEYS* → *ADD KEY* → *Create new key* → *JSON*). The downloaded file is typically named `gcs-key.json`.

← 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 ▾

3. Prepare the Secret data:

- Create `gcs.conf` for repository `repo3`:

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

- Encode both `gcs-key.json` and `gcs.conf` files using base64:



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



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

- Create the Secret manifest. For example, `cluster1-pgbackrest-secrets.yaml`:

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

You can store credentials for several repositories in one Secret by adding separate data keys.

4. Apply the Secret. Replace `<namespace>` with your namespace:

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

5. Update `deploy/cr.yaml` Custom Resource manifest:

- Reference the Secret in `backups.pgbackrest.configuration`
- Provide the backup path in [backups.pgbackrest.global](#) with the `pgBackRest path` option. The repository name must match the name in `gcs.conf` (`repo3-path` in this example)
- Define the GCS bucket under `repos`.

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

6. Apply the cluster Custom Resource. Replace the `<namespace>` placeholder with your value:

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

Next steps

Make an on-demand backup

Make a scheduled backup

Microsoft Azure Blob Storage

Azure Blob Storage support is currently in tech preview.

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

- Azure container name
- Azure Storage account name and key, stored in a [Kubernetes Secret](#)

Configuration steps

The examples below use the `repo4` repository name.

1. Encode the Azure Storage credentials and the pgBackRest repo name that you will use for backups with base64:



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



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



2. Create the Secret manifest and set `data.azure.conf` to the encoded string. The following is the example of the `cluster1-pgbackrest-secrets.yaml` Secret file:

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



You can store credentials for several repositories in one Secret by adding separate data keys.

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

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

4. Update the `deploy/cr.yaml` Custom Resource:

- Reference the Secret in `backups.pgbackrest.configuration`
- Provide the backup directory path in [backups.pgbackrest.global](#) with the `pgBackRest path` option (for example `repo4-path`). The repository name must match the name used when encoding credentials
- Specify the Azure container name under `repos`

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

5. Apply the cluster Custom Resource:

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

Next steps

[Make an on-demand backup](#)[Make a scheduled backup](#)

Persistent Volume

Percona Operator for PostgreSQL uses [Kubernetes Persistent Volumes](#)  for PostgreSQL data. You can use the same mechanism for backups on a volume attached to the pgBackRest Pod.

The Operator creates a Persistent Volume when it provisions the cluster. The default backup repository on a volume is `repo1` in `backups.pgbackrest.repos`:

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



This configuration is enough to run backups. To use a specific [Storage Class](#) , set it in `volumeClaimSpec` as described in the [Operator Custom Resource options](#).

Next steps

[Make an on-demand backup](#)[Make a scheduled backup](#)

Make a backup

Make scheduled backups

Backups schedule is defined on the per-repository basis in the `backups.pgbackrest.repos` subsection of the `deploy/cr.yaml` file.

You can supply each repository with a `schedules.<backup type>` key equal to an actual schedule that you specify in crontab format.

- 1 Before you start, make sure you have [configured a backup storage](#).
- 2 Configure backup schedule in the `deploy/cr.yaml` file. The schedule is specified in crontab format as explained in [Custom Resource options](#). The repository name must be the same as the one you defined in the [backup storage configuration](#). The following example shows the schedule for `repo1` repository:

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



1. Update the cluster:

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



Next steps

[Restore from a backup](#)

Useful links

- [Backup retention](#)
- [Restore options](#)

Making on-demand backups

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

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



Here's a sequence of steps to follow:

- 1 Before you start, make sure you have [configured a backup storage](#).
- 2 In the `deploy/backup.yaml` configuration file, specify the cluster name and the repository name to be used for backups. The repository name must be the same as the one you defined in the [backup storage configuration](#). It must also match the repository name specified in the `backups.pgbackrest.manual` subsection of the `deploy/cr.yaml` file.
- 3 If needed, you can add any [pgBackRest command line options](#) .
- 4 Make a backup with the following command (modify the `-n postgres-operator` parameter if your database cluster resides in a different namespace):

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



 Expected output



- 5 Making a backup takes time. Use the `kubectl get pg-backup` command to track the backup progress. When finished, backup should obtain the `Succeeded` status:

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



 Expected output



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



Tip

To list available backups, run:

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



Next steps

[Restore from a backup](#)

Useful links

- [Backup retention](#)
- [Restore options](#)

Restore from a backup

Restore options: in-place restore vs cluster clone

You can restore your PostgreSQL data from a backup in these ways:

- [In-place restore](#) – restore data into the same cluster using the [PerconaPGRestore](#) custom resource. By default, the Operator restores the most recent backup. You can specify what backup to restore from using the `--set` option.
- [Cluster clone](#) – create a new cluster from a backup using the `dataSource` option in the Custom Resource of a **new** cluster. Use this approach to deploy a copy of your cluster in a new namespace or a different Kubernetes cluster to:
 - Test your disaster recovery strategy
 - Analyze the data without affecting the production cluster
 - Restore the database from a cloud storage when the source cluster no longer exists.

Both approaches support full restore and point-in-time recovery. Choose the method that fits your scenario.

Next steps

- [In-place restore](#) – restore to the same cluster
- [Cluster clone](#) – restore to a new cluster
- [Restore options reference](#) – PerconaPGRestore spec

Restore to an existing PostgreSQL cluster (in-place restore)

To restore data into an existing cluster, use the `PerconaPGRestore` custom resource.

You can make a full restore or restore the database to a specific point in time.

Important

This operation overwrites the current data and is destructive.

Configure `PerconaPGRestore` custom resource using a *backup restore* configuration file. The example of the backup configuration file is [deploy/restore.yaml](#) .

Prepare your environment

Export the namespace where your cluster is running as an environment variable. Replace the `<namespace>` placeholder with your value:

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



Make a full restore from the latest backup (default)

Specify the following options in the `deploy/restore.yaml` configuration file for the `PerconaPGRestore` object:

- `pgCluster` - the name of your cluster,
- `repoName` - the name of the `pgBackRest` repository, where the backup is located. The repo with the same name must already be configured in the `backups.pgbackrest.repos` subsection of the cluster Custom Resource,
- `options` (optional) - additional [pgBackRest command line options](#) .

Here is the example configuration:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGRestore
metadata:
  name: restore1
spec:
  pgCluster: cluster1
  repoName: repo1
```



Start the restore process:

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



Restore from a specific backup

When you have multiple backups, the Operator restores the [latest full backup](#) by default.

If you want to restore from a specific previous backup, use the `--set` option with the backup label.

Here's the sequence of steps to follow:

- 1 List available backups:

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



- 2 Get detailed information about the backup you wish to restore from:

```
kubectl describe pg-backup <BACKUP NAME> -n $NAMESPACE
```



Sample output



Look for the “Backup Name” in the Status section (for example, `20240628-074416F`). This is the label that you will use with the `--set` option.

- 3 Modify the `deploy/restore.yaml` configuration file. Specify this information:

- `pgCluster` - the name of your cluster
- `repoName` - the name of the `pgBackRest` repository, where the backup is located. The repo with the same name must already be configured in the `backups.pgbackrest.repos` subsection of the cluster Custom Resource
- Configure the `options` section:
 - `--type=<type>` - Specify how you wish to restore the data. The `default` type restores the backup and replays WAL up to the end of available WAL. The `immediate` type restores the backup exactly as it was at the backup time, without replaying WAL.
 - `--set=<backup_label>` - Specify the backup label.

Here's the example configuration to restore from a backup `20240628-074416F`:

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



4 Start the restore

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



Restore the cluster with point-in-time recovery

Point-in-time recovery lets you restore your database to the state it was before a change occurred (for example, before accidental data deletion or corruption).



Note

To support this, the Operator automatically creates an initial full backup when your cluster is first created. This initial backup is used as the starting point for point-in-time recovery. This backup is required internally and does not appear when running `kubectl get pg-backup`.

By default, the Operator uses the latest successful full backup as the base for point-in-time restore. You can specify another backup to use as the base by referencing its ID. Refer to the [Specify a base backup for point-in-time restore](#) section to learn more.

To make a point-in-time restore, you need the following:

- A backup that finished before your target time. You cannot restore to the time where there was no backup
- All relevant WAL files must be successfully archived
- Use the `--type=time` and `--target` options in the `options` subsection of the `deploy/restore.yaml` configuration file.

Here's the sequence of steps to follow:

1 List available backups:

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



- 2 Determine the target restore time. The Operator tracks the latest restorable time for each backup by default. To view this value, run:

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



- 3 Edit the `deploy/restore.yaml` configuration file and specify this information:

- `pgCluster` - the name of your cluster
- `repoName` - the name of the `pgBackRest` repository, where the backup is located. The repo with the same name must already be configured in the `backups.pgbackrest.repos` subsection of the cluster Custom Resource
- Configure the `options` section:
 - `--type` - set to `time`,
 - `--target` set the target time that you retrieved at the previous step. The format is `<YYYY-MM-DD HH:MM:DD>`, optionally followed by a timezone offset: `"2021-04-16 15:13:32+00"` (`+00` here means UTC).

Here's the example configuration:

```
apiVersion: pgv2.percona.com/v2  
kind: PerconaPGRestore  
metadata:  
  name: restore1  
spec:  
  pgCluster: cluster1  
  repoName: repo1  
  options:  
    - --type=time  
    - --target="2025-11-30 15:12:11+03"
```



- 4 Start the restore process:

```
bash
```

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

Specify a base backup for point-in-time restore

You can select a base backup for point-in-time restore. To do this you must get the backup ID and then specify it for the `--set` option in the restore configuration file.

To get the backup ID, do the following:

1. Get the Pod name:

```
kubectl get pods -n $NAMESPACE
```



2. Connect to the Pod and get the backupID with the `pgbackrest --stanza=db info` command :

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



Find ID of the needed backup in the output:

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

  db (prior)
    wal archive min/max (16):
0000000F0000000000000001C/00000020000000036000000C5

full backup: 20240401-173403F
  timestamp start/stop: 2024-04-01 17:34:03+00 / 2024-04-01 17:36:57+00
  wal start/stop: 0000001200000000000000022 / 0000001200000000000000024
  database size: 31MB, database backup size: 31MB
  repo1: backup set size: 4.1MB, backup size: 4.1MB

incr backup: 20240401-173403F_20240415-201250I
  timestamp start/stop: 2024-04-15 20:12:50+00 / 2024-04-15 20:14:19+00
  wal start/stop: 000000190000000000000005C / 000000190000000000000005D
  database size: 46.0MB, database backup size: 25.7MB
  repo1: backup set size: 6.1MB, backup size: 3.8MB
  backup reference list: 20240401-173403F

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

3. Reference this backup ID to the *backup restore* configuration file:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGRestore
metadata:
  name: restore1
spec:
  pgCluster: cluster1
  repoName: repo1
  options:
    - --type=time
    - --target="2024-04-01 17:36:57+00"
    - --set="20240401-173403F"
```

4. Start the restore:

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

Provide pgBackRest with a custom restore command

There may be cases where it is needed to control what files are restored from the backup and apply fine-grained filtering to them. For such scenarios there is a possibility to overwrite the [restore_command used in PostgreSQL archive recovery](#). You can do it in the `patroni.dynamicConfiguration` subsection of the Custom Resource as follows:

```
patroni:
  dynamicConfiguration:
    postgresql:
      parameters:
        restore_command: "pgbackrest --stanza=db archive-get %f \"%p\""
```

The `%f` template in the above example is replaced by the name of the file to retrieve from the archive, and `%p` is replaced by the copy destination path name on the server. See [PostgreSQL official documentation](#) for more low-level details about this feature.

Fix the cluster if the restore fails

The restore process overwrites database files. Wrong data or a misconfigured restore can leave your cluster in a non-operational state.

For example, incorrect `pgBackRest` arguments in the [PerconaPGRestore custom resource](#) can break the database while the restore hangs.

Here's what you can do:

- You can remove the restore annotation from your cluster Custom Resource to stop the restore:

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



- Alternatively, you can delete the cluster [by removing the Custom Resource](#) and recreate it. Before you delete the Custom Resource, ensure the [finalizers.percona.com/delete-pvc finalizer](#) is not set, or you will lose your data. Then run the same `kubectl apply` command you used to deploy the cluster originally.

Another example that can also cause restore failures is corrupted backup repository or missing files. In that case, [remove the Custom Resource](#), [locate and delete the startup PVC](#), then recreate the cluster.

Restore the backup to a new cluster (cluster clone)

Apart from [restoring the data on the same database cluster](#), you can restore a backup to a new cluster and run it alongside the existing one.

This is useful for:

- Cloning a cluster to a new namespace or Kubernetes environment
- Creating a copy for development, testing, or reporting
- Restoring from a cloud storage when the source cluster no longer exists

You can make a full restore or restore the database to a specific point in time. For each restore scenario, you must define the Custom Resource for a **new** cluster with these configuration options:

- `dataSource` section - where to take the data from
- `backups` section. The new cluster needs its own backup configuration.

Understand the `dataSource` options

The `dataSource` section in the Custom Resource includes two subsections: `dataSource.postgresCluster` and `dataSource.pgbackrest`.



Note

You cannot use both `dataSource.postgresCluster` and `dataSource.pgbackrest` at the same time. If both are present in the Custom Resource, the `dataSource.postgresCluster` option will take precedence and the Operator will use it to restore the data.

`dataSource.postgresCluster`

Configure this subsection **to clone an existing cluster**. The key options are:

- `dataSource.postgresCluster.clusterName` is the name of the cluster you restore from. This is the source cluster. The option value corresponds to the `metadata.name` of the source cluster Custom Resource.
- `dataSource.postgresCluster.clusterNamespace` is the namespace where the source cluster is deployed. Use it if namespaces of source and new clusters differ.
- `dataSource.postgresCluster.repoName` is the name of the `pgBackRest` repository on the source cluster where the backup you restore from is located. It must exist on the source.

- `dataSource.postgresCluster.options` are additional `pgBackRest` options that you pass for the restore. For example, you configure them for point-in-time recovery.

Read more about all available options in the [Custom Resource reference](#)

`dataSource.pgbackrest`

Configure this subsection **to restore from backup repository stored in a cloud storage**.

Its structure closely matches the source cluster's `backups.pgbackrest` section, with these main points:

- Define the backup source using a single `repo` object (not an array as in `backups.pgbackrest`).
- Specify `stanza` (usually `db`), required to identify the backup.
- Reference the same Secret for cloud credentials in both the restore and backup configuration.

Key options are:

- `dataSource.pgbackrest.stanza` - the name of `pgBackRest` stanza - a unique identifier for a source PostgreSQL cluster's backup configuration
- `dataSource.pgbackrest.configuration.secret.name` - the name of the Secret object with the credentials to the cloud storage. It must be the same in both source and new clusters because the restore Pod requires the same credentials as the original backup Pod.
- `dataSource.pgbackrest.global` is the location of a backup.
- `dataSource.pgbackrest.repo` is the name of the `pgBackRest` repository. It is the same on both source and new clusters.

For all options, see the [Custom Resource reference](#).

Clone from an existing cluster

Make a full data clone

To create an independent copy of your cluster, add the `dataSource.postgresCluster` section to the Custom Resource of the *new* cluster.

Key fields:

- `clusterName` – name of the source cluster
- `clusterNamespace` – namespace of the source cluster (required when cloning to a different namespace; requires the Operator in [cluster-wide mode](#))
- `repoName` – name of the `pgBackRest` repository in the source cluster containing the backup to use for the

restore

You also need to configure the storage and backup settings for the new cluster:

- In the `instances` section, define the `dataVolumeClaimSpec` for your new cluster, which sets up the PVC. This determines the storage resources (size, access mode, etc.) allocated for your cloned database data.
- In the `backups.pgbackrest.repos` section, set up a backup repository for the new cluster. The repo name must match the one used in `repoName` above. Also configure the backup storage. This ensures the new cluster both restores data and is able to perform its own backups independently.

The following example creates a cluster named `cluster2` as a clone of `cluster1` in the `percona-db-1` namespace:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGCluster
metadata:
  name: cluster2
spec:
  crVersion: 3.0.0
  dataSource:
    postgresCluster:
      clusterName: cluster1
      clusterNamespace: percona-db-1
      repoName: repo1
  instances:
    - name: instance1
      replicas: 1
      dataVolumeClaimSpec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - ReadWriteOnce
              resources:
                requests:
                  storage: 1Gi
```



Deploy the new cluster:

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



This configuration allows your new cluster to both restore data from the source backup and operate as a fully functioning, independently backed-up PostgreSQL cluster.

Make a clone with point-in-time recovery

To restore from a backup up to a specific point in time, you need the following:

- A backup that finished before your target time. You cannot restore to the time where there was no backup
- All relevant WAL files must be successfully archived
- Use the `--type=time` and `--target` options in the `options` subsection of the `deploy/restore.yaml` configuration file.

Use the same settings as for [a full data clone](#). Also, add `pgBackRest` options for point-in-time recovery to `dataSource.postgresCluster.options`. These options are:

- `--type=time`: Instructs `pgBackRest` to initiate a point-in-time recovery.
- `--target`: The timestamp up to which to restore the data. To get the timestamp, run this command on the **source cluster**: `kubectl get pg-backup <backup_name> -n <namespace> -o jsonpath='{.status.latestRestorableTime}'`
- `--set` (optional): Allows you to specify a particular backup as the starting point for point-in-time recovery. For more information how to do it, refer to the [Specify a base backup for point-in-time restore](#) section.



```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGCluster
metadata:
  name: cluster2
spec:
  crVersion: 3.0.0
  dataSource:
    postgresCluster:
      clusterName: cluster1
      clusterNamespace: percona-db-1
      repoName: repo1
      options:
        - --type=time
        - --target="2025-11-30 15:12:11+03"
  instances:
    - name: instance1
      replicas: 1
      dataVolumeClaimSpec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - ReadWriteOnce
              resources:
                requests:
                  storage: 1Gi
```

The new cluster will be restored to the specified point in time and then promoted. You can start accessing it from that specific timestamp.

Restore specific databases

You might need to restore only specific databases on a new cluster. For example, for performance reasons or due to storage limits.

Important

Note that **only the specified databases** will be restored and available on a new cluster. All other databases from a backup will **not** be accessible. This means that if you have `db1`, `db2` and `db3` in a backup and you specify only `db1`, you will have access only to this `db1`. `db2` and `db3` will not be restored.

Also check [pgBackRest limitations for restoring specific databases](#)

To restore only specific databases to a new cluster, start with [the basic cluster clone configuration](#), and add the `--db-include` flag under `options` to list the databases you want to restore. For example, to restore just the `app1` database, use:

```
spec:
  dataSource:
    postgresCluster:
      clusterName: cluster1
      clusterNamespace: percona-db-1
      repoName: repo1
      options:
        - --db-include=app1
```

List additional databases with separate `--db-include` flags as needed.

Clone from cloud storage (S3, GCS, Azure Blob)

You can create a new cluster when the source cluster no longer exists but backups remain in a cloud storage (AWS S3, Google Cloud Storage, or Azure Blob Storage). This is useful for disaster recovery, for keeping data compressed on cheaper storage and restoring it when needed, or for creating a standalone copy from archived backups.

Before you start

You need the backup configuration from the original cluster: the path where backups were stored, the Secret with cloud credentials, and the storage settings (bucket, endpoint, region).

If the source cluster is still running and you plan to delete it, take a full backup first for best results, then delete the cluster once the backup completes.

Clone from S3 storage

1. Configure the `dataSource.pgbackrest` subsection in the new cluster Custom Resource.

Configure these fields correctly:

Section	Field	Purpose
<code>dataSource.e.pgbackrest</code>	<code>stanza</code>	pgBackRest stanza name (usually <code>db</code>). Required for cloud restore.
<code>dataSource.e.pgbackrest</code>	<code>configuration.secret.name</code>	Secret with cloud credentials. Must match the Secret used by the source cluster.
<code>dataSource.e.pgbackrest</code>	<code>global.restore.pg1-path</code>	Path where the source cluster stored its backups. Use the same path as in the original cluster's <code>backups.pgbackrest.global</code> .
<code>dataSource.e.pgbackrest</code>	<code>repo</code>	Storage config (bucket, endpoint, region) matching the source. Single object, not an array.
<code>backups.pgbackrest</code>	<code>global.restore.pg1-path</code>	Path for the new cluster's backups. Use a different path (e.g., with the new cluster name) so the clone backs up to its own location and does not overwrite the original backups.

The following example creates `cluster2` from backups that `cluster1` stored in the S3 storage. The source cluster may already be deleted.



```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGCluster
metadata:
  name: cluster2
spec:
  crVersion: 3.0.0
  dataSource:
    pgbackrest:
      stanza: db
      configuration:
        - secret:
            name: cluster1-pgbackrest-secrets
    global:
      repo1-path: /pgbackrest/postgres-operator/cluster1/repo1
  repo:
    name: repo1
    s3:
      bucket: my-bucket
      endpoint: s3.ca-central-1.amazonaws.com
      region: ca-central-1
  instances:
    - name: instance1
      replicas: 1
      dataVolumeClaimSpec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      configuration:
        - secret:
            name: cluster1-pgbackrest-secrets
    global:
      repo1-path: /pgbackrest/postgres-operator/cluster2/repo1
  repos:
    - name: repo1
      s3:
        bucket: my-bucket
        endpoint: s3.ca-central-1.amazonaws.com
        region: ca-central-1
```

2. Deploy the cluster:



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

3. Check that the cluster is ready:



```
kubectl describe perconapgcluster cluster2 -n percona-db-2
```

When the number of ready instances matches the expected instances, the cloned cluster is up and running.

Clone from cloud, backup to local storage

You can restore from cloud storage but configure the new cluster to use a local Persistent Volume for its own backups. Replace the `backups.pgbackrest` section with a volume-based repo:

```
backups:
  pgbackrest:
    repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
              - ReadWriteOnce
            resources:
              requests:
                storage: 1Gi
```



The `dataSource.pgbackrest` section stays the same; only the new cluster's backup destination changes.

GCS and Azure Blob Storage

For Google Cloud Storage or Azure Blob Storage, use the same structure but replace the `repo.s3` block with `repo.gcs` or `repo.azure` and the matching configuration. See [Google Cloud Storage](#) and [Microsoft Azure Blob Storage](#) for examples.

PVC snapshots

PVC snapshot support

This feature is in the tech preview stage. The API and behavior may change in future releases.

This document provides an overview of PVC snapshots. If you are familiar with the concept and want to try it out, jump to the [Configure and use PVC snapshots](#) tutorial. If you run on Amazon EKS, start with [Set up PVC snapshots on EKS](#).

By reading this document you will learn the following:

- [PVC snapshot support](#)
 - [Overview](#)
 - [Workflow](#)
 - [Why to use PVC snapshots](#)
 - [Requirements](#)
 - [Limitations](#)

Overview

A PVC snapshot is a point-in-time copy of a [Persistent Volume Claim](#)  created by the storage provider. It captures the volume contents at a specific moment without copying data block by block.

PVC snapshots are much faster than streaming data to cloud storage or a backup volume. This is especially beneficial for large datasets. The Operator uses the [Kubernetes VolumeSnapshot API](#)  to create PVC snapshots at the storage level. When used with `pgBackRest` WAL archiving, PVC snapshots ensure data consistency and provide support for point-in-time recovery.

Workflow

The Operator currently supports only cold backups (the `offline` mode). A cold backup is also known as an offline backup. It is a storage-level backup taken from a PostgreSQL replica instance after it is temporarily suspended by the Operator. This ensures consistency by capturing the entire database exactly as it exists at the moment when the replica is temporarily suspended.

During cold backups, the Operator:

1. Selects a **replica** instance as the snapshot target.
2. Issues a PostgreSQL `CHECKPOINT` on that replica (if checkpoint is enabled).
3. **Suspends** the replica StatefulSet (scales it to zero).

4. Creates Kubernetes `VolumeSnapshot` objects for the data PVC, WAL PVC (if separate), and any tablespace PVCs.
5. Waits for all snapshots to become `ReadyToUse`.
6. **Resumes** the replica StatefulSet.

This approach ensures a crash-consistent snapshot while minimizing the impact on the primary. Only a replica is taken offline, so the cluster continues serving read/write traffic on the primary during the snapshot.

Why to use PVC snapshots

PVC snapshots speed up backups and restores, which is especially beneficial for large data sets. With this feature, you get:

- **Much faster backups** – Snapshot creation is typically seconds to minutes, regardless of database size. Time it takes to run traditional full backups increases as the size of your database grows.
- **Much faster restores** – Restoring from a snapshot is significantly faster than restoring from cloud storage. Both in-place restores and restores to a new cluster are supported.
- **Lower resource usage** – Snapshots avoid the CPU and network overhead of streaming data to a remote storage.

Requirements

Before enabling PVC snapshots, ensure the following:

1. Your Kubernetes cluster must have the CSI driver that supports VolumeSnapshot API. An example of such driver for GKE is `pd.csi.storage.gke.io`, for EKS - `ebs.csi.aws.com`.
2. Your Kubernetes cluster must have the VolumeSnapshot CRDs installed. Verify if they are installed with this command:

```
kubectl get crd volumesnapshots.snapshot.storage.k8s.io
```



Expected output

```
volumesnapshotclasses.snapshot.storage.k8s.io volumesnapshotcontents.snapshot.storage.k8s.io  
volumesnapshots.snapshot.storage.k8s.io
```

3. At least one `VolumeSnapshotClass` must exist and be compatible with the storage class used by your PostgreSQL data volumes. Check it with:

```
kubectl get volumesnapshotclasses
```



See how to add it in the [Add a VolumeSnapshotClass](#) section.

1. You must explicitly enable the `BackupSnapshots` feature gate in the Operator Deployment. See [Enable the feature gate](#).
2. You must explicitly specify the storage class whose CSI driver supports VolumeSnapshot API in the `spec.instances.[ ]dataVolumeClaimSpec.storageClassName` option in the Custom Resource for your cluster.

Limitations

- **Currently only offline mode** – Only offline snapshots are supported; the Operator must stop a replica pod to take a consistent snapshot of the database.
- **At least one replica required** – Your cluster must have at least one replica pod besides the primary. The Operator takes the snapshot from a replica; clusters without replicas cannot use this feature.
- **CSI driver support required** – your Kubernetes cluster's storage provisioner must support the Volume Snapshot API.
- **One snapshot backup at a time** – You can only run one snapshot backup at a time on a given cluster; concurrent snapshot backups are not supported.
- **No automatic retention policy for scheduled backups yet** - You must manually delete outdated backup objects. Retention policy is planned to be added in future releases.

[Configure PVC snapshots](#)

Configure and use PVC snapshots

This document provides step-by-step instructions for configuring and using Persistent Volume Claim (PVC) Snapshots with Percona Operator for PostgreSQL on Kubernetes.

For a high-level explanation of PVC snapshots, please refer to the [PVC snapshot support](#) chapter.

Amazon EKS users

If you run your cluster on Amazon EKS, refer to the [Set up PVC snapshots on EKS](#) tutorial. EKS requires specific addons, a gp3 storage class, and a matching VolumeSnapshotClass before you can use PVC snapshots.

Prerequisites

To use PVC snapshots, ensure you have the following prerequisites met:

1. Your Kubernetes cluster must have a CSI driver that supports Volume Snapshots

For example, Google Kubernetes Engine (GKE) with `pd.csi.storage.gke.io`, or Amazon EKS with `ebs.csi.aws.com`. Check what driver you have with:

```
kubectl get csidriver
```



2. Your Kubernetes cluster must have VolumeSnapshot CRDs installed. Most managed Kubernetes providers include these by default. Verify by running:

```
kubectl get crd | grep volumesnapshots
```



Expected output



3. At least one `VolumeSnapshotClass` must exist and be compatible with the storage class used by your PostgreSQL data volumes. Check it with:

```
kubectl get volumesnapshotclasses
```



If you don't have one, you can add it yourself. Refer to the [Add a VolumeSnapshotClass](#) section.

4. You must enable the `BackupSnapshots` feature gate for the **Percona Operator for PostgreSQL** deployment. Refer to the [Enable the feature gate](#) section for details.

Before you start

1. Check the [prerequisites](#) and [limitations](#)
2. Clone the Operator repository to be able to edit manifests:

```
git clone -b v3.0.0 https://github.com/percona/percona-postgresql-operator
```



3. Export the namespace where you run your cluster as an environment variable:

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



Configuration

Enable the feature gate

If you have the Operator Deployment up and running, you can edit the `deploy/operator.yaml` manifest. If you deploy the Operator from scratch, edit the `deploy/bundle.yaml` manifest.

1. Edit the `deploy/operator.yaml` or `deploy/bundle.yaml` and set the `PGO_FEATURE_GATES` environment variable for the Operator Deployment to `"BackupSnapshots=true"`:

```
spec:
  containers:
  - name: percona-postgresql-operator
    env:
    - name: PGO_FEATURE_GATES
      value: "BackupSnapshots=true"
```



2. Apply the configuration:

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



or

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



Add a VolumeSnapshotClass

If your Kubernetes cluster doesn't have a `VolumeSnapshotClass` that matches your CSI driver, create one.

1. Create a `VolumeSnapshotClass` configuration file with the following configuration:

Google Kubernetes Engine (GKE)

volume-snapshot-class.yaml

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotClass
metadata:
  name: gke-snapshot-class
driver: pd.csi.storage.gke.io
deletionPolicy: Delete
```



Microsoft Azure Kubernetes Service (AKS)

volume-snapshot-class.yaml

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotClass
metadata:
  name: aks-snapshot-class
driver: disk.csi.azure.com
deletionPolicy: Delete
```



OpenShift

volume-snapshot-class.yaml

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotClass
metadata:
  name: openshift-snapshot-class
driver: ebs.csi.aws.com
deletionPolicy: Delete
```



2. Create the VolumeSnapshotClass resource:

```
kubectl apply -f volume-snapshot-class.yaml
```



Expected output



Configure PVC snapshots in your cluster

You must reference the `VolumeSnapshotClass` in your cluster Custom Resource.

1. Check the name of the `VolumeSnapshotClass` that works with your storage. You can list available classes with:

```
kubectl get volumesnapshotclasses
```



Sample output



2. Edit the `deploy/cr.yaml` Custom Resource. Reference the Storage Class that supports the VolumeSnapshot API in the `spec.instances.[ ]dataVolumeClaimSpec.storageClassName` option. The Operator then uses this storage class when it creates the cluster.

Here's the example configuration:

```
spec:
  instances:
  - name: instance1
    dataVolumeClaimSpec:
      storageClassName: standard
      accessModes:
      - ReadWriteOnce
      resources:
        requests:
          storage: 1Gi
```



3. Edit the `deploy/cr.yaml` Custom Resource and add the `volumeSnapshots` subsection under `backups`. Specify these keys:
 - `className` - the name of the `VolumeSnapshotClass`
 - `mode` - how to make backups. `offline` is currently the only supported mode.

```
spec:
  backups:
    volumeSnapshots:
      className: <name-of-your-volume-snapshot-class>
      mode: offline
```



4. Apply the configuration to update the cluster:

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



Once configured, snapshots are created automatically when you [make a manual on-demand backup](#) or when [a scheduled backup runs](#).

Configure PVC snapshots on Amazon EKS

This tutorial walks you through setting up PVC snapshots for Percona Operator for PostgreSQL on Amazon Elastic Kubernetes Service (EKS). EKS has specific requirements that differ from other Kubernetes providers such as GKE.

For a high-level explanation of PVC snapshots, see the [PVC snapshot support](#) chapter.

Prerequisites

To proceed with the setup, ensure you have:

- An EKS cluster (or plan to create one)
- `kubectl` configured to access your cluster
- The AWS CLI installed and configured

EKS requires specific addons, a `gp3` storage class, and a matching `VolumeSnapshotClass` before you can use PVC snapshots.

Before you start

1. Check the [prerequisites](#) and [limitations](#)
2. Clone the Operator repository to be able to edit manifests:

```
git clone -b v3.0.0 https://github.com/percona/percona-postgresql-operator
```



3. Export the namespace where you run your cluster as an environment variable:

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



Configuration

Enable required EKS addons

EKS requires two addons for volume snapshots: the Amazon EBS CSI driver and the snapshot controller. You can enable them when you create the cluster or add them to an existing cluster.

Option A: Add addons when creating the cluster

If you deploy the cluster with `eksctl` or a similar tool, include these addons in your cluster configuration:

```
addons:
  - name: aws-ebs-csi-driver
    wellKnownPolicies:
      ebsCSIDriver: true
  - name: snapshot-controller
nodeGroups:
  - name: ng-1
    desiredCapacity: 3
    minSize: 3
    maxSize: 3
```



Option B: Add addons to an existing cluster

If your cluster already exists, enable the addons with the AWS CLI:

```
aws eks create-addon \
  --cluster-name <CLUSTER_NAME> \
  --addon-name aws-ebs-csi-driver \
  --resolve-conflicts OVERWRITE

aws eks create-addon \
  --cluster-name <CLUSTER_NAME> \
  --addon-name snapshot-controller
```



Replace `<CLUSTER_NAME>` with the name of your EKS cluster.

Create a gp3 storage class

The default `gp2` storage class on EKS doesn't support volume snapshots. You must use `gp3` instead.

1. Create a storage class configuration file:

ebs-gp3-storage-class.yaml

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ebs-csi-gp3
provisioner: ebs.csi.aws.com
volumeBindingMode: WaitForFirstConsumer
allowVolumeExpansion: true
parameters:
  type: gp3
```



2. Apply the storage class:

```
kubectl apply -f ebs-gp3-storage-class.yaml
```



Create a VolumeSnapshotClass

1. Create a VolumeSnapshotClass configuration file:

ebs-gp3-snapshot-class.yaml

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotClass
metadata:
  name: ebs-csi-gp3
driver: ebs.csi.aws.com
deletionPolicy: Delete
```



2. Apply the VolumeSnapshotClass:

```
kubectl apply -f ebs-gp3-snapshot-class.yaml
```



Expected output



Enable the feature gate

If you have the Operator Deployment up and running, you can edit the `deploy/operator.yaml` manifest. If you deploy the Operator from scratch, edit the `deploy/bundle.yaml` manifest.

1. Edit the `deploy/operator.yaml` or `deploy/bundle.yaml` and set the `PGO_FEATURE_GATES` environment variable for the Operator Deployment to `"BackupSnapshots=true"`:

```
spec:
  containers:
  - name: percona-postgresql-operator
    env:
    - name: PGO_FEATURE_GATES
      value: "BackupSnapshots=true"
```



2. Apply the configuration:

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



or

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



Configure PVC snapshots in your cluster

You must reference the `VolumeSnapshotClass` in your cluster Custom Resource.

1. Check the name of the `VolumeSnapshotClass` that works with your storage. You can list available classes with:

```
kubectl get volumesnapshotclasses
```

2. Edit the `deploy/cr.yaml` Custom Resource. Reference the Storage Class you created in the `spec.instances.[ ]dataVolumeClaimSpec.storageClassName` option. The Operator then uses this storage class when it creates the cluster.

Here's the example configuration:

```
spec:
  instances:
  - name: instance1
    dataVolumeClaimSpec:
      storageClassName: ebs-csi-gp3
      accessModes:
      - ReadWriteOnce
      resources:
        requests:
          storage: 1Gi
```

3. In the Custom Resource, add the `volumeSnapshots` subsection under `backups`. Specify these keys:
 - `className` - the name of the `VolumeSnapshotClass`
 - `mode` - how to make backups. `offline` is currently the only supported mode.

```
spec:
  backups:
    volumeSnapshots:
      className: <name-of-your-volume-snapshot-class>
      mode: offline
```

4. Apply the configuration to update the cluster:

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

Once configured, snapshots are created automatically when you [make a manual on-demand backup](#) or when [a scheduled backup runs](#).

Use PVC snapshots

Once the PVC snapshots are configured, you can use them to make backups and restores.

Make an on-demand backup from a PVC snapshot

1. Configure the `PerconaPGBBackup` object. Edit the `deploy/backup.yaml` manifest and specify the following keys:
 - `pgCluster` - the name of your cluster. Check it with the `kubectl get pg -n $NAMESPACE` command
 - `method` - the backup method. Specify `volumeSnapshot`.

Here's the example configuration:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGBBackup
metadata:
  name: my-snapshot-backup
spec:
  pgCluster: cluster1
  method: volumeSnapshot
```



2. Apply the configuration to start a backup:

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



3. Check the backup status:

```
kubectl get pg-backup my-snapshot-backup -n $NAMESPACE
```



Sample output



Make a scheduled snapshot-based backup

1. Configure the backup schedule in your cluster Custom Resource. Edit the `deploy/cr.yaml` manifest. In the `schedule` key in the `volumeSnapshots` subsection under `backups`, specify the schedule in the Cron format for the snapshots to be made automatically. Your updated configuration should look like this:



```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGCluster
metadata:
  name: my-cluster
spec:
  backups:
    volumeSnapshots:
      className: my-snapshot-class
      mode: offline
      schedule: "0 3 * * *" # Every day at 3:00 AM
```

2. Apply the configuration to update the cluster:



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

In-place restore from a PVC snapshot

An in-place restore is a restore to the same cluster using the `PerconaPGRestore` custom resource. You can make a full in-place restore or a point-in-time restore.

When you create the `PerconaPGRestore` object, the Operator performs the following steps:

1. Suspends all instances in the cluster.
2. Deletes all existing PVCs in the cluster. This removes all existing data, WAL, and tablespaces.
3. Creates new PVCs with the snapshot serving as the data source. This restores the data, WAL, and tablespaces from that snapshot.
4. Spins up a job to configure the restored PVCs to be used by the cluster.
5. Resumes all instances in the cluster. The cluster starts with the data from the snapshot.



Important

An in-place restore overwrites the current data and is destructive. Any data that was written after the backup was made is lost. Therefore, consider restoring to a new cluster instead. This way you can evaluate the data before switching to the new cluster and don't risk losing data in the existing cluster.

Follow the steps below to make a full in-place restore from a PVC snapshot.

1. Configure the `PerconaPGRestore` object. Edit the `deploy/restore.yaml` manifest and specify the following keys:
 - `pgCluster` - the name of your cluster. Check it with the `kubectl get pg -n $NAMESPACE` command

- `volumeSnapshotBackupName` - the name of the PVC snapshot backup. Check it with the `kubectl get pg-backup -n $NAMESPACE` command.

Here's the example configuration:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGRestore
metadata:
  name: restore1
spec:
  pgCluster: cluster1
  volumeSnapshotBackupName: my-snapshot-backup
```

2. Apply the configuration to start a restore:

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

3. Check the restore status:

```
kubectl get pg-restore restore1 -n $NAMESPACE
```

 **Sample output**



In-place restore with point-in-time recovery

You can make a point-in-time restore from a PVC snapshot and replay WAL files from a WAL archive made with pgBackRest. For this scenario, your cluster must meet the following requirements:

1. Have a `pgBackRest` configuration, including the backup storage and at least one repository. See the [Configure backup storage](#) section for configuration steps.
2. The repository must have at least one WAL archive.

The workflow for point-in-time restore is similar to [a full in-place restore](#). After the Operator restores the data from the snapshot, it replays the WAL files from the WAL archive to bring the cluster to the target time.

Important

An in-place restore overwrites the current data and is destructive. Any data that was written after the backup was made is lost. Therefore, consider restoring to a new cluster instead. This way you can evaluate the data before switching to the new cluster and don't risk losing data in the existing cluster.

Follow the steps below to make a point-in-time restore from a PVC snapshot.

1. Check the repo name and the target time for the restore.

- List the backups:

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

- For a `pgBackRest` backup run the following command to get the target time:

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

2. Configure the `PerconaPGRestore` object. Edit the `deploy/restore.yaml` manifest and specify the following keys:

- `pgCluster` - the name of your cluster. Check it with the `kubectl get pg -n $NAMESPACE` command
- `volumeSnapshotBackupName` - the name of the PVC snapshot backup.
- `repoName` - the name of the `pgBackRest` repository that contains the WAL archives.
- `options` - the options for the restore. Specify the following options:
 - `--type=time` - set to `time` to make a point-in-time restore.
 - `--target` - set the target time for the restore.

Here's the example configuration:

```
apiVersion: pgv2.percona.com/v2  
kind: PerconaPGRestore  
metadata:  
  name: pitr-restore  
spec:  
  pgCluster: cluster1  
  volumeSnapshotBackupName: my-snapshot-backup  
  repoName: repo1  
  options:  
    - --type=time  
    - --target="2026-02-16T11:00:00Z"
```

3. Apply the configuration to start a restore:

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

4. Check the restore status:

```
kubectl get pg-restore pitr-restore -n $NAMESPACE
```

Create a new cluster from a PVC snapshot

You can create a new cluster from a PVC snapshot. This is useful when you want to restore the data to a new cluster and don't want to overwrite the existing data in the existing cluster.

To create a new cluster from a PVC snapshot, you need to configure the `PerconaPGCluster` object and specify an `volumesnapshot` object as the `dataSource`. You also need to configure the `instances` and `backups` sections to set up the new cluster.

For more information about the `dataSource` options, see the [Understand the `dataSource` options](#) section. Also check the [Custom Resource reference](#) for all available options.

Follow the steps below to create a new cluster from a PVC snapshot.

1. Create the namespace where a new cluster will be deployed and export it as the environment variable:

```
kubectl create namespace <new-namespace>
export NEW_NAMESPACE=<new-namespace>
```



2. Configure the `PerconaPGCluster` object. Edit the `deploy/cr.yaml` manifest and specify the following keys:
 - `dataSource` - the name of the PVC snapshot backup. Check it with the `kubectl get pg-backup my-snapshot-backup -o jsonpath='{.status.snapshot.dataVolumeSnapshotRef}'` command on the **source** cluster.
 - `instances` - the instances configuration for the new cluster.
 - `backups` - the backups configuration for the new cluster.

Here's the example configuration:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGCluster
metadata:
  name: new-cluster
spec:
  instances:
    - name: instance1
      replicas: 3
      dataVolumeClaimSpec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 10Gi
  dataSource:
    apiGroup: snapshot.storage.k8s.io
    kind: VolumeSnapshot
    name: <name-of-the-pvc-snapshot-backup>
```



3. Apply the configuration to create the new cluster:

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



The new cluster will be provisioned shortly using the volume of the source cluster.

Configure backup encryption

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

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

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

This document describes how to configure backup encryption.

Generate the encryption key

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

```
openssl rand -base64 48
```



Configure backup storage

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

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

1. Encode the storage configuration file.

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

macOS

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

2. Create the Secrets configuration file and the Secrets object following steps 2-3 of the [S3-compatible backup storage configuration](#). For other storage types, consult the [relevant configuration tutorial](#).
3. Update the `deploy/cr.yaml` configuration. Specify the following information:
 - The Secret name you created in the `backups.pgbackrest.configuration` subsection
 - All storage-related information in the `backups.pgbackrest.repos` subsection under the repository name that you intend to use for backups. This name must match the name you used when you encoded S3 credentials on step 1.
 - The cipher type in the `pgbackrest.global` subsection

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

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

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

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



Make a backup

[Make an on-demand backup](#)

[Make a scheduled backup](#)

Speed-up backups with pgBackRest asynchronous archiving

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

The pgBackRest tool used by the Operator solves this problem by using the [WAL asynchronous archiving](#) feature.

Asynchronous archiving is enabled by default in the `pgBackRest` configuration. It requires the spool path to store transient data. The spool path may differ depending on the image and version. An example spool path is `/pgdata/pgbackrest-spool`.

You can further fine-tune asynchronous archiving by setting the max number of parallel processes for `archive-push` and `archive-get` commands.

Be sure not to set a high `process-max` value because it may affect normal database operations.

Your storage configuration file may look as follows:

s3.conf

```
[global]
repo2-s3-key=REPLACE-WITH-AWS-ACCESS-KEY
repo2-s3-key-secret=REPLACE-WITH-AWS-SECRET-KEY
repo2-storage-verify-tls=n
repo2-s3-uri-style=path

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

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



No modifications are needed aside of setting these additional parameters.

Further reading

Find more information about WAL asynchronous archiving in:

- [pgBackRest official documentation](#)
- [How pgBackRest is Addressing Slow PostgreSQL WAL Archiving Using Asynchronous Feature](#) blog post.

Backup retention

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

You can control backup retention by the following `pgBackRest` options:

- `--<repo name>-retention-full` number of full backups to retain,
- `--<repo name>-retention-diff` number of differential backups to retain.

You can also specify retention type for full backups as `<repo name>-retention-full-type`, setting it to either `count` (the number of full backups to keep) or `time` (the number of days to keep a backup for).

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

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



Differential retention can be set in a similar way:

```
backups:
  pgbackrest:
  ...
    global:
      repo1-retention-diff: "3"
      ...
```



Delete the unneeded backup

The maximum amount of stored backups is controlled by the [retention policies](#). Older backups are automatically deleted.

Manual deleting of a previously saved backup requires not more than the backup name. This name can be taken from the list of available backups returned by the following command:

```
kubectl get pg-backup
```



When the name is known, backup can be deleted as follows:

```
kubectl delete pg-backup/<backup-name>
```



Delete backups on cluster deletion

You can enable [percona.com/delete-backups finalizer](#) in the Custom Resource (turned off by default) to ensure that all backups are removed when the cluster is deleted. If the finalizer is enabled, the Operator will delete all the backups from all the configured repos on cluster deletion. Besides removing all the physical backup files, finalizer will also delete all `pg-backup` objects.



Warning

This `percona.com/delete-backups` finalizer is in tech preview state, and it is not yet recommended for production environments.

Disable backups

The recommended approach to deploy and run the database is with the disaster recovery strategy in mind. Therefore, the Operator is designed and running with the backups enabled by default.

There are some specific use cases when you may wish to run a database without enabled backups. Disabling backups should be a conscious decision based on your data's value and recoverability. These are example use cases where it is considered acceptable are when the data is fully disposable:

- Ephemeral development/testing environments: For clusters that are frequently torn down and rebuilt from application code or test data scripts.
- CI/CD pipeline jobs: For automated pipeline runs where the cluster's entire lifecycle is temporary and tied to a single job.

Key considerations before disabling backups

Before you proceed with disabling backups, here's what you need to know and carefully assess:

1. Without backups you have no way to restore data. If by mistake you drop a table, that data is lost as you have no option to recover it.
2. You cannot clone a cluster when you [deploy a standby cluster for disaster recovery](#). This is because cloning is based on restoring a backup on a new cluster.
3. When you run a cluster without backups, `pgBackRest` metrics are unavailable.

Start a new cluster with disabled backups

To deploy a new cluster without backups, do the following:

1. Clone the Operator repository to be able to edit resource manifests.

```
git clone -b v3.0.0 https://github.com/percona/percona-postgresql-operator
```



2. Edit the `deploy/cr.yaml` Custom Resource and set the `backups.enabled` option to `false`

```
spec:
  backups:
    enabled: false
```



3. Apply the Custom Resource to start the cluster creation.

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



Disable backups for a running cluster

Before you start, read the [considerations](#) carefully.

To disable backups for a running cluster, update the `deploy/cr.yaml` Custom Resource manifest with the following configuration:

- Set the `backups.enabled` option to `false`
- Add the annotation `pgv2.percona.com/authorizeBackupRemoval:"true"`

Since it is a running cluster, we will use the `kubectl patch` command to update its configuration:

```
kubectl patch pg cluster1 --type merge \
  -p '{
    "metadata": {
      "annotations": {
        "pgv2.percona.com/authorizeBackupRemoval": "true"
      }
    },
    "spec": {
      "backups": {
        "enabled": false
      }
    }
  }' -n <namespace>
```



Warning

After you apply this configuration and disable backups, the Operator deletes the `repo-host` PVC. Thus, all data that was stored in that PVC will be deleted too. The backups stored on the cloud backup storage remain.

Re-enable backups

To re-enable backups for a running cluster, do the following:

1. Remove the annotation `pgv2.percona.com/authorizeBackupRemoval:"true"`

```
kubectl annotate pg cluster1 pgv2.percona.com/authorizeBackupRemoval-
```



2. Apply the patch to your running cluster and enable backups:



```
kubect1 patch pg cluster1 --type merge \  
-p '{  
  "spec": {  
    "backups": {  
      "enabled": true  
    }  
  }  
'
```

Deploy a standby cluster for Disaster Recovery

Deploy a standby cluster for Disaster Recovery

A standby cluster is a special type of PostgreSQL cluster designed to maintain a continually updated, read-only copy of your primary PostgreSQL cluster. A standby cluster applies changes made on the primary cluster either through backups or streaming replication. This ensures you always have an up-to-date replica ready to take over if the primary becomes unavailable.

In disaster recovery scenarios, standby clusters are invaluable. They let you minimize downtime and data loss by quickly promoting the standby to a writable primary when needed. This is especially important in multi-cloud or multi-regional environments, where your PostgreSQL workloads may span across various platforms and geographies.

The Percona Operator for PostgreSQL makes it straightforward to deploy and manage standby clusters on Kubernetes. By automating complex tasks and integrating tightly with Kubernetes, the operator helps you protect critical data and achieve high availability, while simplifying your operational experience.

Types of standby clusters

You can use the Percona Operator for PostgreSQL to set up several types of standby clusters, each suited to different recovery strategies:

1. A repo-based standby that recovers WAL files from a `pgBackRest` repo stored in external storage. For this setup, you reference the `pgBackRest` repo name and the cloud-based backup configuration that matches the one from the primary site. Refer to the [Standby cluster deployment based on pgBackRest](#) tutorial for the setup steps.
2. A streaming standby receives WAL files by connecting to the primary over the network. The primary site must be accessible over the network and allow secure authentication with TLS. The standby cluster must securely authenticate to the primary. For this reason, both sites must have the same custom TLS certificates. For the setup, you provide the host and port of the primary cluster and the certificates. Learn more about the setup in the [Standby cluster deployment based on streaming replication](#) tutorial.
3. Streaming standby with external repository is the combination of two previous types and is configured with the options from both types. In this setup, the standby cluster streams WAL records from the primary. If the streaming replication falls behind, the cluster recovers WAL from the backup repo.

Detect replication lag for standby cluster

If your primary cluster has a large volume of WAL files, the standby cluster may not be able to apply them quickly enough. This may cause the standby to fall behind. This lag can result in replication issues and temporarily leave some data unavailable on the standby cluster.

You can enable replication lag detection for any standby type by setting the `standby.maxAcceptableLag` option in the Custom Resource.

How replication lag is detected and handled

The Operator checks the WAL source based on the standby type: from the primary site or from an external repository.

For a standby cluster configured with both streaming and external repository, the Operator detects the replication lag as follows:

1. The Operator first attempts to check streaming replication lag.
2. If it fails to retrieve the streaming lag, the Operator falls back to checking the WAL replication lag from an external pgBackRest repository. Note the [known limitation for the repo-based standby](#).
3. If streaming lag is successfully detected, this value is used.

When the WAL lag exceeds the value you specified in the `standby.maxAcceptableLag` option, the following occurs:

- The primary pod in the standby cluster is marked as `Unready`
- The cluster goes into the `initializing` state
- The `StandbyLagging` condition is set in the cluster status. You can check the conditions with the `kubectl describe pg <cluster-name> -n <namespace>` command.

Monitor lag in the cluster status

When lag detection is enabled, the cluster status includes a `standby` section that shows the current lag and when it was last computed. Use `kubectl get pg <cluster-name> -n <namespace> -o yaml` and look at `status.standby`:

```
status:
  standby:
    lagBytes: 2343212
    lagLastComputedAt: "2026-02-24T12:07:05Z"
```



- `lagBytes` — the current WAL lag in bytes (if any)
- `lagLastComputedAt` — the timestamp of the last lag check

This helps you understand if replication is lagging or broken. By surfacing the standby lag condition, you get a clear signal when your standby is not ready to serve traffic, enabling faster troubleshooting and preventing application downtime during disaster recovery scenarios.

Known limitation for a repo-based standby cluster

For WAL lag detection to work in this standby type, the Operator must have access to the primary cluster. Therefore, WAL lag detection is available in these setups:

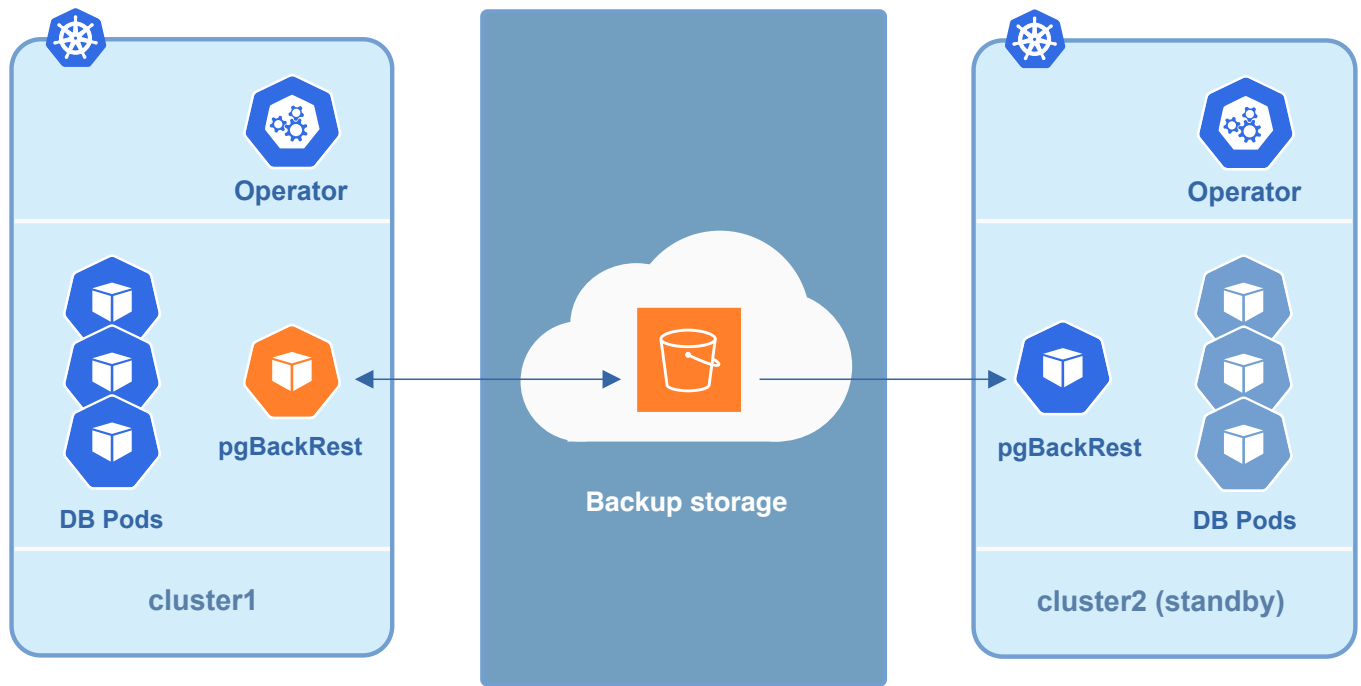
- Primary and standby clusters are deployed in the same namespace
- Primary and standby are deployed in different namespaces of the same cluster and the Operator is installed in cluster-wide mode.

Disable replication lag detection

To disable detecting the replication lag, remove the `standby.maxAcceptableLag` from the Custom Resource. The changes apply without restarting the database Pods.

Standby cluster deployment based on pgBackRest

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



pgBackrest repo based standby

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

Deploy disaster recovery for PostgreSQL on Kubernetes

Configure Main site

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

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

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

3. Once configured, apply the custom resource:

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

Expected output



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

Configure DR site

The configuration of the disaster recovery site is similar [to that of the Main site](#), with the difference in standby settings.

The following manifest has `standby.enabled` set to `true` and points to the `repoName` where backups are (GCS in our case). Optionally, add `standby.maxAcceptableLag` to enable [replication lag detection](#). Check the [known limitation for this standby type](#).

```
metadata:
  name: standby
spec:
  ...
  backups:
    configuration:
      - secret:
          name: standby-pgbackrest-secrets
    pgbackrest:
      repos:
        - name: repo1
      gcs:
        bucket: MY-BUCKET
  standby:
    enabled: true
    repoName: repo1
    # maxAcceptableLag: 10Mi # optional: enables replication lag detection
```

Deploy the standby cluster by applying the manifest:

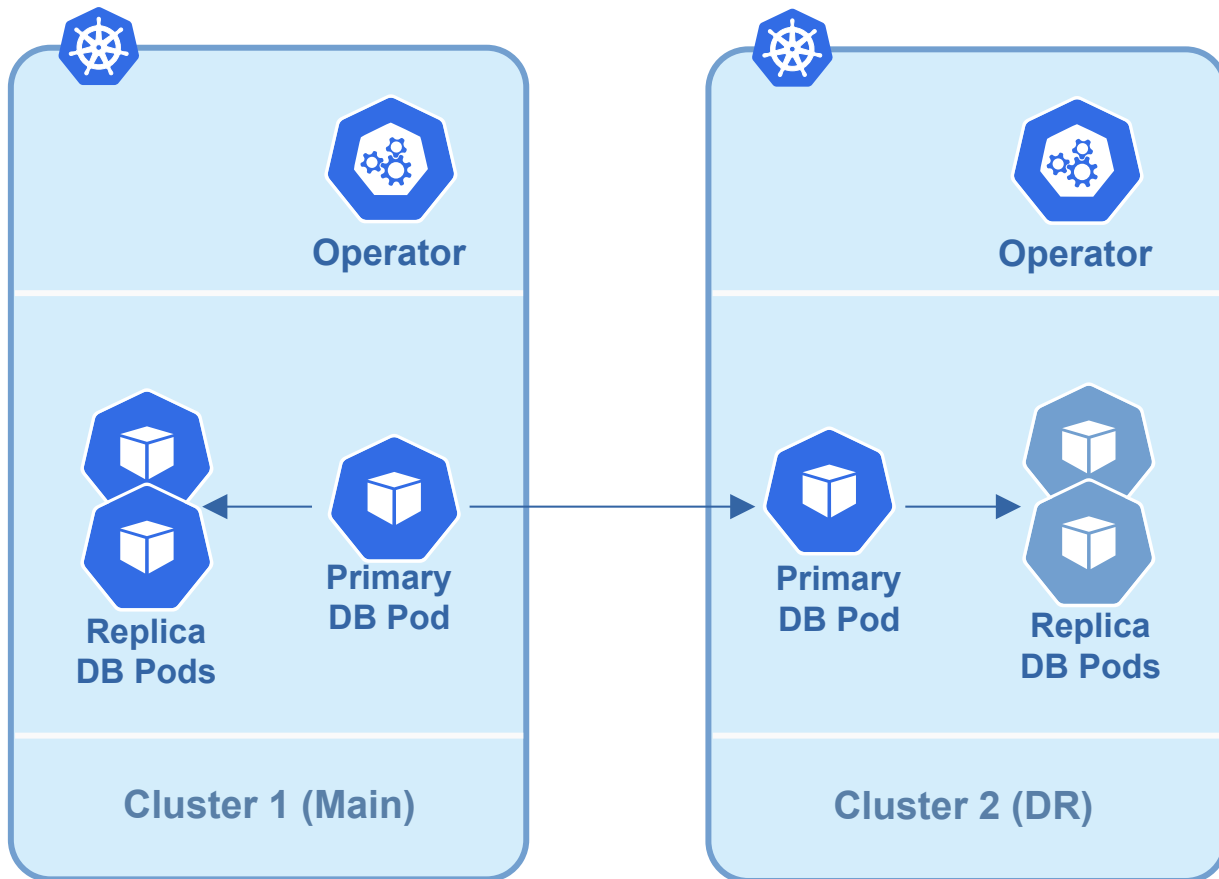
```
kubectl apply -f deploy/cr.yaml
```

 Expected output



Standby cluster deployment based on streaming replication

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



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

Deploy disaster recovery for PostgreSQL on Kubernetes

Configure Main site

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

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

Use here a Service type of your choice. For example, `ClusterIP` is sufficient for two clusters in different Kubernetes namespaces.

3. Once configured, apply the custom resource:

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

Expected output

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

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

Configure DR site

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

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

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

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

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

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

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

You can find more about certificates in the [TLS doc](#).

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

```
standby:
  enabled: true
  host: main-ha.main-pg.svc
  maxAcceptableLag: 10Mi # optional: enables replication lag detection
```

- `standby.enabled` controls if it is a standby cluster or not
- `standby.host` must point to the primary node of a Main cluster. In this example it is a `main-ha` Service in another namespace.
- `standby.maxAcceptableLag` (optional) enables replication lag detection. When the WAL lag exceeds this value, the standby primary pod is marked as unready, the cluster goes into `initializing` state, and a `StandbyLagging` condition is set in the status.

Deploy the standby cluster by applying the manifest:

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

 Expected output



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

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

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

Failover

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

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

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



Now you can start writing to the cluster.

Split brain

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

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

Automate the failover

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

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

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

Scale Percona Distribution for PostgreSQL on Kubernetes

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

Scaling can be [vertical](#) and horizontal. Vertical scaling adds more compute or storage resources to PostgreSQL nodes; horizontal scaling is about adding more nodes to the cluster. High availability looks technically similar, because it also involves additional nodes, but the reason is maintaining liveness of the system in case of server or network failures.

This document focuses on vertical scaling. For deploying high-availability, see [High-availability](#) guide.

Vertical scaling

Scale compute

There are multiple components that the Operator deploys and manages: PostgreSQL instances, pgBouncer connection pooler, pgBackRest and others (see [Architecture](#) for the full list of components).

You can manage compute resources for a specific component using the corresponding section in the Custom Resource manifest. We follow the structure for [requests and limits](#)  that Kubernetes provides.

The most common resources to specify are CPU and memory (RAM).

You can specify a **request** for CPU or memory for a component's Pod. In this case, the Kubernetes scheduler uses these values to decide on which Kubernetes node to place the Pod, ensuring the node has at least the requested resources available. The Pod will only be scheduled on a node that can satisfy all its resource requests.

If you specify a **limit** for the resources, this is the maximum amount of CPU or memory the container is allowed to use. If the container tries to use more than the limit, it may be throttled (for CPU) or terminated (for memory).

You can set both `requests` and `limits` in the `resources` section of your Custom Resource. For example:

```
spec:
  ...
  instances:
  - name: instance1
    replicas: 3
    resources:
      requests:
        cpu: 1.0
        memory: 2Gi
      limits:
        cpu: 2.0
        memory: 4Gi
```



If you only set `limits` and omit `requests`, Kubernetes will default the request to the limit value.

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

Scale storage

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

Scaling with Volume Expansion capability

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

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

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



Expected output



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

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

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

Apply changes as usual:

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



Automated scaling with auto-growable disk

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

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



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

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

High availability

High availability (HA) ensures that your PostgreSQL database remains accessible even in the event of node or pod failures. With the Percona Operator for PostgreSQL, high availability is achieved by running multiple PostgreSQL nodes in a cluster, using the Patroni framework for automated failover and PostgreSQL streaming replication for data consistency.

A PostgreSQL cluster consists of the following members:

- A Primary node handles all write operations. The Primary continuously streams changes to its Standby nodes.
- Read-only (Standby in PostgreSQL terminology) replicas that continuously receive and replay changes from the Primary node. If the Primary fails, one of the Standbys can be automatically promoted to become the new Primary.

Data replication

Percona Operator leverages PostgreSQL streaming replication to keep Standby nodes up-to-date.

By default, **asynchronous replication** is used: the Primary sends changes to Standbys, but does not wait for confirmation before committing transactions. This offers better performance but presents a risk of minimal data loss (transactions not yet copied to a Standby could be lost in a failure).

Synchronous replication is also supported. In this replication type the Primary waits for at least one Standby to acknowledge receipt of data before marking a transaction as committed. This minimizes the risk of data loss, but can be slightly slower since each transaction must wait for a confirmation.

Minimum and recommended number of nodes for high availability:

The absolute minimum that can technically work for high availability is **2 nodes**. But this does not provide full high availability or protection against split-brain scenarios since the loss of either node can impact availability and data safety.

The recommended number of nodes for high availability setups is **3 or more PostgreSQL nodes**.

Adding nodes to a cluster

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

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

Using `spec.instances.replicas`

For example, you have the following Custom Resource manifest:

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



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

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



...and applying the Custom Resource:

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



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

Using `spec.instances`

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

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

For example, you have the following Custom Resource manifest:

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

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

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

Using Synchronous replication

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

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

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

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

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



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

Find more options useful to tune how your database cluster should operate in synchronous mode [in the official Patroni documentation](#) [↗](#).

Using huge pages with Percona Operator for PostgreSQL

Overview

Huge Pages (also called large or super pages) are bigger memory blocks that help reduce CPU overhead. Normally, memory is managed in 4kB chunks, also called “pages”, but when your PostgreSQL workload grows, the CPU has to juggle a lot of these small pages. By switching to larger pages like 2MiB or 1GiB, you reduce the number of pages the CPU needs to track, which can improve efficiency and performance.

For PostgreSQL clusters managed by Percona Operator for PostgreSQL, enabling huge pages is a recommended optimization, especially for memory-intensive workloads.

Why to use huge pages in PostgreSQL

PostgreSQL uses shared memory extensively for:

- Shared buffer pool
- WAL buffers
- Dynamic shared memory segments

When huge pages are enabled:

- PostgreSQL can access memory more efficiently.
- The system spends less time managing memory.
- Performance improves, especially under heavy load.

Configure huge pages for Percona Operator for PostgreSQL

Enable huge pages in your Kubernetes environment

Before configuring your cluster, make sure huge pages are enabled and available on the Kubernetes nodes. This setup is done outside the Operator and depends on your Kubernetes environment, whether you use a cloud-based Kubernetes like GKE, EKS, etc or use a bare-metal one.

Consult the Kubernetes environment’s official documentation for how to enable huge pages there.

For the further setup, you need to keep in mind the following:

- What page sizes are available (e.g., 2MiB vs 1GiB)

- How many pages are preallocated
- Will other workloads compete for these pages
- Do all nodes that will run PostgreSQL pods have huge pages available
- When adding more nodes to your cluster, will they have huge pages available

Request huge pages in your cluster Custom Resource

Once your Kubernetes nodes are ready, you can configure your PostgreSQL cluster to use huge pages.

1. Set the huge pages resource limits in your `deploy/cr.yaml` Custom Resource.

This example configuration tells Kubernetes to allocate 16Mi worth of 2MiB huge pages for this instance. If you're using 1GiB pages, change the key to `hugepages-1Gi`.

```
spec:
  instances:
    - name: instance1
      resources:
        limits:
          hugepages-2Mi: 16Mi
          memory: 4Gi
```



Important

Kubernetes requires that `requests` and `limits` for huge pages match. If you only specify `limits`, Kubernetes will assume the same value for `requests`.

2. Apply the configuration

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



Verify huge pages are reserved

After deploying your cluster with huge pages configured, you can verify that they're being used by checking inside the database container:

```
cat /proc/meminfo | grep HugePages
```



You should see values for `HugePages_Total`, `HugePages_Free`, and `HugePages_Rsvd`, confirming that huge pages are reserved and in use.

A note on default behavior

To avoid unexpected startup failures, Percona Operator disables huge pages by default (`huge_pages = off`). This prevents PostgreSQL from trying to use huge pages when none were requested. Once you explicitly configure huge pages in your spec, the Operator sets `huge_pages = try` , allowing PostgreSQL to use them if available.

If huge pages are enabled on your nodes but not requested by your pods, PostgreSQL might fall back to minimal memory settings. To avoid this, either:

- Enable huge pages properly in your pod spec.
- Schedule pods on nodes without huge pages.
- Or manually set `shared_buffers` to a reasonable value:

```
spec:
  config:
    parameters:
      shared_buffers: 128MB
```



Using sidecar containers

Sidecar containers are extra containers that run alongside the main container in a Pod. They are often used for logging, proxying, or monitoring.

The Operator uses a set of “predefined” sidecar containers to manage the cluster operation:

- `replica-cert-copy` - is responsible for copying TLS certificates needed for replication between PostgreSQL instances
- `pgbouncer-config` - handles configuration management for `pgBouncer`
- `pgbackrest` - runs the main backup/restore agent
- `pgbackrest-config` - handles configuration management for `pgBackRest`

You can also deploy your own 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 use them with caution, only when you are sure what you are doing.

Adding a custom sidecar container

You can add sidecar containers to these Pods:

- a PostgreSQL instance Pod
- a `pgBouncer` Pod
- a `pgBackRest` repo host Pod

To add a sidecar container, use the `instances.sidecars` or `proxy.pgBouncer.sidecars` subsection in the `deploy/cr.yaml` configuration file. Specify this minimum required information in this subsection:

- the container name
- the container image
- a command to run

Note that you cannot reuse the name of the predefined containers. For example, PostgreSQL instance Pods cannot have custom sidecar containers named as `database`, `pgbackrest`, `pgbackrest-config`, and `replica-cert-copy`.

Use the `kubectl describe pod` command to check which names are already in use.

Here is the sample configuration of a sidecar container for a PostgreSQL instance Pod:

```
spec:
  instances:
  - name: instance1
    ....
    sidecars:
    - image: busybox:latest
      command: ["sleep", "30d"]
      args: ["-c", "while true; do echo echo $(date -u) 'test' >> /dev/null; sleep 5;
done"]
      name: my-sidecar-1
    ....
```

Find additional options suitable for the `sidecars` subsection in the [Custom Resource options reference](#) and the [Kubernetes Workload API reference](#) 

Apply your modifications:

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

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

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

 Expected output



Getting shell access to a sidecar container

You can login to your sidecar container as follows:

```
kubectl exec -it cluster1-instance1n8v4-0 -c testcontainer -- sh
/ #
```

Mount volumes into sidecar containers

You can mount volumes into sidecar containers using `sidecarVolumes` and `sidecarPVCs`. These options are available for:

- PostgreSQL instance Pods (`instances[].sidecarVolumes`, `instances[].sidecarPVCs`)
- pgBouncer Pods (`proxy.pgBouncer.sidecarVolumes`, `proxy.pgBouncer.sidecarPVCs`)

- pgBackRest repo host Pods (`backups.pgbackrest.repoHost.sidecarVolumes`, `backups.pgbackrest.repoHost.sidecarPVCs`)

The following subsections describe different [volume types](#) that work with sidecar containers. Use `sidecarVolumes` for existing volumes (secret, configMap, NFS, etc.) and `sidecarPVCs` for operator-managed PersistentVolumeClaims.

Persistent Volume

You can use [Persistent volumes](#) when you need dynamically provisioned storage that does not depend on the Pod lifecycle.

The Operator creates and manages the PVCs you define in `sidecarPVCs`. Claim storage with [persistentVolumeClaim](#) and reference it in your sidecar's `volumeMounts`.

Important

You can use PVCs with sidecar containers only when you deploy a new cluster. Updates to running cluster are not supported.

The following example requests 1Gi storage with `sidecar-volume-claim` and mounts it to the `testcontainer` sidecar under `/volume0`:

```
spec:
  instances:
  - name: instance1
    sidecars:
    - name: testcontainer
      image: busybox:latest
      command: ["sleep", "30d"]
      volumeMounts:
      - name: sidecar-volume-claim
        mountPath: /volume0
    sidecarPVCs:
    - name: sidecar-volume-claim
      spec:
        resources:
          requests:
            storage: 1Gi
        volumeMode: Filesystem
        accessModes:
        - ReadWriteOnce
```

Note

If you set the `percona.com/delete-pvc` [finalizer](#), the Operator deletes sidecar PVCs when the cluster is removed.

Secret

You can use a [secret volume](#) to pass the information which needs additional protection (e.g. passwords), to the sidecar container. Secrets are stored with the Kubernetes API and mounted to the container as RAM-stored files.

Create the [Secret object](#) with the sensitive information that you want to pass to the container.

Define the volume in `sidecarVolumes` and reference the Secret there, Then reference the volume in your sidecar `volumeMounts`:

```
spec:
  instances:
  - name: instance1
    sidecars:
    - name: rs-sidecar-0
      image: busybox
      command: ["/bin/sh"]
      args: ["-c", "while true; do sleep 5; done"]
      volumeMounts:
      - mountPath: /secret
        name: sidecar-secret
    sidecarVolumes:
    - name: sidecar-secret
      secret:
        secretName: mysecret
```



The example creates a `sidecar-secret` volume from the existing `mysecret` [Secret object](#) and mounts it under `/secret`.

configMap

You can use a [configMap volume](#) to pass configuration data to the container. ConfigMaps are stored in the Kubernetes API and mounted as RAM-backed files.

As with the Secret, you must create the [configMap object](#) first.

Then define the volume in `sidecarVolumes` and reference your configMap object. Finally, reference the volume in your sidecar `volumeMounts`.

This example creates a `sidecar-config` volume from the existing `myconfigmap` [configMap object](#) and mounts it under `/config`:

```
spec:
  instances:
  - name: instance1
    sidecars:
    - name: rs-sidecar-0
      image: busybox
      command: ["/bin/sh"]
      args: ["-c", "while true; do sleep 5; done"]
      volumeMounts:
      - mountPath: /config
        name: sidecar-config
    sidecarVolumes:
    - name: sidecar-config
      configMap:
        name: myconfigmap
```

NFS

You can use an [NFS volume](#) to mount a shared NFS export into your sidecar. Add it to `sidecarVolumes`:

```
spec:
  instances:
  - name: instance1
    sidecarVolumes:
    - name: backup-nfs
      nfs:
        server: "nfs-service.storage.svc.cluster.local"
        path: "/pg-some-name"
```

Reference the volume in your sidecar `volumeMounts` with the same `name`.

Pause/resume and standby mode for a PostgreSQL cluster

Pause and resume the cluster

You can temporarily shut down your PostgreSQL cluster and bring it back later without losing data or configuration. You may want to pause the cluster for maintenance tasks, emergency manual intervention or debugging.

When paused, all changes to the cluster's current state are suspended and no statuses other than the "Progressing" condition are updated until you resume the reconciliation.

How to pause

Set the `spec.pause` option to `true` in your `deploy/cr.yaml` Custom Resource:

```
spec:
  pause: true
  # ... rest of your spec
```

Apply the change:

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

The Operator will gracefully stop the cluster (primary, replicas, pgBackRest, and related jobs).

How to resume

Set `spec.pause` back to `false` in the same Custom Resource and apply:

```
spec:
  pause: false
  # ... rest of your spec
```

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

The Operator will start the cluster again using the existing data volumes.

Troubleshooting

The Operator does not pause the cluster

If a backup job is running, the Operator will not pause the cluster and will log a warning. Remove a running backup job so you can pause:

```
kubect1 delete job -l postgres-operator.crunchydata.com/pgbackrest-backup -n  
<namespace>
```



Then retry pausing the cluster.

Standby mode

Standby PostgreSQL clusters provide a continuously replicated copy of your primary cluster, forming the backbone of high-availability and disaster-recovery strategies. They stay in sync through streaming replication, enabling you to quickly promote a standby if the primary becomes unavailable. Standby clusters can also run in separate regions or environments, helping you maintain business continuity during outages.

The standby mode for a cluster is controlled with the `spec.standby.enabled` option plus the `spec.standby.repoName` and/or `spec.standby.host` and `spec.standby.port` options in the Custom Resource. What options to specify depends on the standby cluster type.

Read more about the supported types of standby clusters and their setup in the [Deploy a standby cluster for Disaster Recovery](#) documentation.

Monitor with Percona Monitoring and Management (PMM)

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

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

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

PMM2 has reached the end-of-life stage and is deprecated. We recommend to use the latest PMM 3.

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

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

Considerations

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

Install PMM Server

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

Install PMM Client

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

Configure authentication

PMM3

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

Warning

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

PMM2 (deprecated)

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

From PMM UI

[Generate the PMM API key](#) 

From command line

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

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



Warning

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

Create a secret

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

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

PMM 3

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

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



PMM 2 (deprecated)

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

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



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

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



Expected output



Deploy a PMM Client

1. Update the `pmm` section in the [deploy/cr.yaml](#)  file.
 - Set `pmm.enabled = true`.
 - Specify your PMM Server hostname / an IP address for the `pmm.serverHost` option. The PMM Server IP address should be resolvable and reachable from within your cluster.
 - Specify the name of the Secret object that you created earlier

```
pmm:
  enabled: true
  image: percona/pmm-client:3.7.1
#   imagePullPolicy: IfNotPresent
  secret: cluster1-pmm-secret
  serverHost: monitoring-service
```

2. Update the cluster

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

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

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

Update the secrets file

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

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



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



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

For example, to set the new service account token in the `my-cluster-name-secrets` object, do the following:



```
kubectl patch secret/cluster1-pmm-secret -p '{"data":{"PMM_SERVER_TOKEN": '$(echo -n <new-token> | base64 --wrap=0)'}}'
```



```
kubectl patch secret/cluster1-pmm-secret -p '{"data":{"PMM_SERVER_TOKEN": '$(echo -n <new-token> | base64)'}}'
```

Check the metrics

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

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

Upgrade

Upgrade Percona Operator for PostgreSQL

The upgrade process consists of these steps:

- Upgrade the [Custom Resource Definition \(CRD\)](#). 
- Upgrade the Operator deployment
- Upgrade the database (Percona Distribution for PostgreSQL)



Breaking changes for the upgrade to Operator 3.0.0

From version **3.0.0**, upstream Crunchy CRDs are renamed and use the API group `upstream.pgvector.percona.com` instead of the previous `postgres-operator.crunchydata.com` group. This change enables you to deploy both Percona and Crunchy Operators in the same Kubernetes cluster and provides a way for smooth migration from one to another.

Important points to consider:

- The new CRDs with the API group `upstream.pgvector.percona.com` are created and the Operator migrates all resources to use them automatically. This change affects all PostgreSQL clusters managed by the Operator.
- After you upgraded the Operator to version 3.0.0, it can no longer use the previous version CRDs.

To learn more about this change, see [Upgrading the Operator and CRD](#).

Update scenarios

You can either upgrade both the Operator and the database, or you can upgrade only the database. To decide which scenario to choose, read on.

Full upgrade (CRD, Operator, and the database)

When to use this scenario:

- The new Operator version has changes that are required for new features of the database to work
- The Operator has new features or fixes that enhance automation and management.
- Compatibility improvements between the Operator and the database require synchronized updates.

When going on with this scenario, make sure to test it in a staging or testing environment first. Upgrading the Operator may cause performance degradation.

Upgrade only the database

When to use this scenario:

- The new version of the database has new features or fixes that are not related to the Operator or other

components of your infrastructure

- You have updated the Operator earlier and now want to proceed with the database update.

When choosing this scenario, consider the following:

- Check that the current Operator version supports the new database version.
- Some features may require an Operator upgrade later for full functionality.

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

Upgrades from the Operator version 1.x to 2.x are completely different from the upgrades within 2.x versions due to substantial changes in the architecture.

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

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

Upgrading the Operator and CRD

Upgrading Percona Operator for PostgreSQL and its Custom Resource Definitions (CRDs) lets you use new features, fixes, and supported Kubernetes versions.

Use this page to prepare for the upgrade and choose the right upgrade path.

Quick summary

- Upgrade the CRD and the Operator together whenever possible.
- Upgrade incrementally by minor version (for example `2.7.x -> 2.8.x -> 2.9.x`).
- CRD supports the latest 3 Operator minor versions.
- **If you upgrade to `3.0.0+`**, Crunchy CRDs are renamed to a new API group. This change is irreversible. Review [Renamed upstream CRDs for Operator 3.0.0 and later](#) before running upgrade steps.

[Pre-upgrade checklist](#)

Version and compatibility rules

Kubernetes Cluster versions and upgrades

1. Before upgrading the Kubernetes cluster, make sure you have a tested disaster recovery plan and a fresh backup.
2. Plan your Kubernetes cluster or Operator upgrades using version compatibility. Check supported Kubernetes versions in the Operator [release notes](#).

Note that the Operator might run on unsupported or untested Kubernetes versions, but this is risky and not recommended. The Operator may fail if deprecated API versions have been removed.
3. During a Kubernetes cluster upgrade, also upgrade the `kubelet`. Drain the nodes hosting the database Pods during the upgrade process.
4. During the `kubelet` upgrade, nodes transition between `Ready` and `NotReady` states. Older nodes may also be replaced entirely with new nodes. Ensure that nodes hosting database or proxy pods remain healthy and in a stable state after the upgrade.
5. Regardless of the upgrade approach, pods will be rescheduled or recycled. Plan your Kubernetes cluster upgrade accordingly to minimize downtime and service disruption.

Operator and CRD compatibility

1. The Operator version follows the format `major.minor.patch`. Here's how you can understand the version `2.6.0`:
 - `2` - major version
 - `6` - minor version
 - `0` - patch version
2. You can upgrade the Operator only to the nearest `major.minor.patch` version. For example, you can go directly from 2.6.0 to 2.7.1 without any intermediate steps.

If your target version is more than one minor release ahead, make several incremental upgrades sequentially.

For example, to upgrade the CRD and Operator from the 2.4.0 to 2.6.0, upgrade it as follows: 2.4.0 -> 2.5.1 -> 2.6.0.
3. CRD is compatible **with the newest Operator and the two previous minor versions**. If the Operator is no more than two minor versions behind the CRD, you can use it. Even so, we recommend updating both the CRD and Operator Deployment.
4. You can use newer CRDs with older Operator to upgrade multiple [single-namespace Operator deployments](#) in one Kubernetes cluster, where each Operator controls a database cluster in its own namespace. In this case upgrading Operator deployments will look as follows:
 - upgrade the CRD (not 3 minor versions far from the oldest Operator installation in the Kubernetes cluster) first
 - upgrade the Operators in each namespace incrementally to the nearest minor version (e.g. first 2.4.0 to 2.5.1, then 2.5.1 to 2.6.0)

Renamed upstream CRDs (Operator 3.0.0 and later)

Earlier releases of Percona Operator for PostgreSQL reused upstream Crunchy CRDs that had the `postgres-operator.crunchydata.com` API group. That made it impossible to run Percona Operator for PostgreSQL alongside [Crunchy PostgreSQL Operator](#)  in the same Kubernetes cluster, because both saw the same CRDs and interfered with each other.

Starting with version 3.0.0, Crunchy CRDs are renamed and now have the new API group `upstream.pgv2.percona.com` instead. In this way resources managed by Percona Operator for PostgreSQL are isolated from Crunchy's CRDs. This change enables both Operators to coexist in the same cluster.

When you upgrade to **3.0.0 or newer**, apply the new CRDs and roll out the new Operator image as usual. New CRDs are installed alongside the legacy ones so existing workloads keep running during the transition. This means you can continue using the Operator Deployment version 2.8.0 through 2.9.0.



Important

This change is irreversible. As soon as you update the Operator to version 3.0.0, it can no longer use Crunchy CRDs from previous versions.

Also, renaming upstream CRDs affects all database clusters managed by the Operator.

The Operator detects the legacy upstream CRDs and:

- Finds child objects that belong to a legacy `PostgresCluster`. These child objects are: `StatefulSets`, `Deployments`, `Services`, `Secrets`, `ConfigMaps`, `PVCs`, `ServiceAccounts`, `Endpoints`, `Roles`, `RoleBindings`, `Jobs`, `CronJobs`, `PodDisruptionBudgets`, plus the optional CSI snapshot (`VolumeSnapshot`).
- Updates their `ownerReferences` to the new API group CRDs.

As part of the API group migration, the common name in the `pgBackRest` client certificate is updated. The Operator automatically updates both the `pgBackRest` configuration and the related certificate Secret. However, you may experience brief disruptions to `pgBackRest` operations while Kubernetes propagates these changes to the containers. This process typically completes within 1–2 minutes.

The `PerconaPGCluster` status condition `APIGroupMigration` informs you about the resource migration state. Run the `kubectl describe pg <cluster-name>` command and check the `status.conditions` list to see full details.

If no legacy CRDs exist, the Operator creates resources with the new API group from the start.

Pre-upgrade checklist

- Backup is complete and restorable.
- Target Operator version and Kubernetes version are compatible.
- CRD version is compatible with the oldest Operator still running.
- You know your install method (`kubectl`, Helm, OpenShift).
- For target `3.0.0+`, you reviewed the [rename behavior](#).

Choose your upgrade path

Choose the upgrade instructions below based on how you originally deployed the Operator:

You use	Go to
<code>kubectl</code> manifests	Manual upgrade
Helm chart	Upgrade via Helm

Manual upgrade

You can upgrade the Operator and CRD as follows, considering the Operator uses `postgres-operator` namespace, and you are upgrading it to the version 3.0.0.

1. Update the CRD for the Operator and the Role-based access control. You must use the [server-side](#)  flag when you update the CRD. Otherwise you can encounter a number of errors caused by applying the CRD client-side: the command may fail, the built-in PostgreSQL extensions can be lost during such upgrade, etc.

Take the latest versions of the CRD and Role-based access control manifest from the official repository on GitHub with the following commands:

```
kubectl apply --server-side -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/v3.0.0/deploy/crd.yaml
kubectl apply --server-side -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/v3.0.0/deploy/rbac.yaml -n postgres-operator
```



Note

In case of [cluster-wide installation](#), use `deploy/cw-rbac.yaml` instead of `deploy/rbac.yaml`.

2. Update the Percona Operator for PostgreSQL Deployment in Kubernetes by changing the container image of the Operator Pod to the latest version. Find the image name for the current Operator release [in the list of certified images](#). Use the following command to update the Operator to the `3.0.0` version:

```
kubectl -n postgres-operator patch deployment percona-postgresql-operator \
-p '{"spec":{"template":{"spec":{"containers":
[{"name":"operator","image":"docker.io/percona/percona-postgresql-
operator:3.0.0"}]}}}}'
```

3. The deployment rollout starts automatically after the patch. Track it with:

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



Expected output



4. [Delete the previous version CRDs \(For Operator 3.x.x\)](#)

Upgrade via Helm

If you have [installed the Operator using Helm](#), you can upgrade the Operator deployment with the `helm upgrade` command.

The `helm upgrade` command updates only the Operator deployment. The [update flow for the database management system \(Percona Distribution for PostgreSQL\)](#) is the same for all installation methods, whether it was installed via Helm or `kubectl`.

1. You must have the compatible version of the Custom Resource Definition (CRD) in all namespaces that the Operator manages. Starting with version 2.7.0, you can check it using the following command:

```
kubectl get crd perconapgclusters.pgv2.percona.com --show-labels
```



2. Update the [Custom Resource Definition](#)  for the Operator, taking it from the official repository on GitHub.

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

```
kubectl apply --server-side --force-conflicts -f  
https://raw.githubusercontent.com/percona/percona-postgresql-  
operator/v3.0.0/deploy/crd.yaml
```



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

3. Upgrade the Operator deployment

With default parameters

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

```
helm upgrade my-operator percona/pg-operator --version 3.0.0
```



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

With customized parameters

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

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

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



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

```
helm upgrade my-operator percona/pg-operator --version 3.0.0 -f my-values.yaml
```



During the upgrade, you may see a warning to manually apply the CRD if it has the outdated version. In this case, refer to step 2 to upgrade the CRD and then step 3 to upgrade the deployment.

4. [Delete the previous version CRDs \(For Operator 3.x.x\)](#)

Delete the legacy CRDs (for Operator 3.x.x)

After the upgrade is complete, you can delete the legacy CRDs with the API group `postgres-operator.crunchydata.com`. Make sure to do this only if you have confirmed that no objects (such as clusters or related resources) are still referencing these CRDs as their parent. Deleting the CRDs while objects still depend on them can cause orphaned resources or disruptions.

1. Check if there are clusters that haven't been migrated to use the new API group CRDs. The following command finds all `PerconaPGCluster` objects whose `APIGroupMigration` condition is missing or `False`, and prints their namespace, name, and condition status:

```
kubectl get pg -A -o json | jq -r '
.items[]
| (
  [.status.conditions[]? | select(.type=="APIGroupMigration") | .status]
  ) as $conds
| select(($conds | length == 0) or ($conds[] == "False"))
| "\(.metadata.namespace)\t\(.metadata.name)\t\(($conds[0] // "MISSING"))"
,
```

If the output is empty, you can delete the legacy CRDs.

2. Patch each `PostgresCluster` object and remove the `postgres-operator.crunchydata.com/finalizer` finalizer. This ensures the CRDs can be deleted. Replace the `<cluster-name>` and `<namespace>` placeholders with your values in the following command:

```
kubectl patch postgresclusters.postgres-operator.crunchydata.com <cluster-name> \
-n <namespace> \
--type=json \
-p='[{"op":"remove","path":"/metadata/finalizers"}]'
```

3. Delete the CRDs with the API group `postgres-operator.crunchydata.com`.

```
kubectl delete crd \
crunchybridgeclusters.postgres-operator.crunchydata.com \
pgadmins.postgres-operator.crunchydata.com \
pgupgrades.postgres-operator.crunchydata.com \
postgresclusters.postgres-operator.crunchydata.com
```

Note that deleting CRDs also deletes all `postgresclusters.postgres-operator.crunchydata.com` resources.

Next steps

[Upgrade the database](#)

Upgrade the database

Upgrade Percona Distribution for PostgreSQL

There are two types of PostgreSQL upgrades available through the Operator:

- **Minor upgrade** is upgrading within the same major version. For example, from 15.5 to 15.7 or from 16.1 to 16.3.
- **Major upgrade** is upgrading across major versions, such as from 15.5 to 16.3.

Major upgrades are supported starting from Operator **2.4.0** as a tech preview. Starting with Operator **2.9.0**, major upgrades are **generally available (GA)** and fully supported.

Before Operator 2.4.0, only minor upgrades were allowed.

Select the guide for a required upgrade type:

[Run a minor version upgrade](#)

[Run a major version upgrade](#)

Minor version upgrade of Percona Distribution for PostgreSQL

A minor version upgrade is the upgrade within the same major version. For example, from 17.5.2 to 17.6.1.

Considerations

1. Upgrading a PostgreSQL cluster may result in downtime, as well as in [failover](#) caused by updating the primary instance.
2. Starting with the Operator 2.6.0, PostgreSQL images are based on Red Hat Universal Base Image (UBI) 9 instead of UBI 8. UBI 9 has a different version of collation library `glibc` and this introduces a collation mismatch in PostgreSQL. Collation defines how text is sorted and compared based on language-specific rules such as case sensitivity, character order and the like. PostgreSQL stores the collation version used at database creation. When the collation version changes, this may result in corruption of database objects that use it like text-based indexes. Therefore, you need to identify and reindex objects affected by the collation mismatch.

Before you start

1. We recommend to [update PMM Server](#)  **before** upgrading PMM Client.
2. If you are using PMM server version 2, use a PMM client image compatible with PMM 2. If you are using PMM server version 3, use a PMM client image compatible with PMM 3.
3. PMM2 has reached its end-of-life stage and is deprecated in the Operator. We recommend you to migrate to and use PMM3. See [PMM upgrade documentation](#)  for how to migrate from version 2 to version 3.

Upgrade steps

To make a minor upgrade of Percona Distribution for PostgreSQL, do the following:

- ❶ Check the version of the Operator you have in your Kubernetes environment. If you need to update it, refer to the [Operator upgrade guide](#)
- ❷ Check the current version of the Custom Resource and what versions of the database and cluster components are compatible with it. Replace the Operator version with your value in the following command:

```
curl https://check.percona.com/versions/v1/pg-operator/3.0.0 |jq -r  
' .versions[ ].matrix'
```



You can also find this information in the [Versions compatibility matrix](#).

- 3 Update the database, the backup and PMM Client image names with a newer version tag. Find the image names [in the list of certified images](#).

We recommend to update the PMM Server **before** the upgrade of PMM Client. If you haven't done it yet, exclude PMM Client from the list of images to update.

Since this is a working cluster, the way to update the Custom Resource is to [apply a patch](#)  with the `kubectl patch pg cluster1` command.

This example command updates the cluster with the name `cluster1` in the namespace `postgres-operator` to the `3.0.0` version:

With PMM Client

```
kubectl -n postgres-operator patch pg cluster1 --type=merge --patch '{
  "spec": {
    "crVersion": "3.0.0",
    "image": "docker.io/percona/percona-distribution-postgresql:18.4-1",
    "proxy": { "pgBouncer": { "image": "docker.io/percona/percona-pgbouncer:1.25.2-1" } },
    "backups": { "pgbackrest": { "image": "docker.io/percona/percona-pgbackrest:2.58.0-2" } },
    "pmm": { "image": "docker.io/percona/pmm-client:3.7.1" }
  }
}'
```

The following image names in the above example were taken from the [list of certified images](#):

- `docker.io/percona/percona-distribution-postgresql:18.4-1`,
- `docker.io/percona/percona-pgbouncer:1.25.2-1`,
- `docker.io/percona/percona-pgbackrest:2.58.0-2`,
- `docker.io/percona/pmm-client:3.7.1`.

Without PMM Client

```
kubectl patch pg cluster1 -n postgres-operator --type=merge --patch '{
  "spec": {
    "crVersion": "3.0.0",
    "image": "docker.io/percona/percona-distribution-postgresql:18.4-1",
    "proxy": { "pgBouncer": { "image": "docker.io/percona/percona-pgbouncer:1.25.2-1" } },
    "backups": { "pgbackrest": { "image": "docker.io/percona/percona-pgbackrest:2.58.0-2" } }
  }
}'
```

The following image names in the above example were taken from the [list of certified images](#):

- `docker.io/percona/percona-distribution-postgresql:18.4-1,`
- `docker.io/percona/percona-pgbouncer:1.25.2-1,`
- `docker.io/percona/percona-pgbackrest:2.58.0-2`

with PostGIS

When using the [PostGIS](#) extension, make sure to specify the image that contains it for the `kubectl patch` command. Add PMM client image to the list if you also use it.

```
kubectl patch pg cluster1 -n postgres-operator --type=merge --patch '{
  "spec": {
    "crVersion": "3.0.0",
    "image": "docker.io/percona/percona-distribution-postgresql-with-
postgis:18.4-2",
    "proxy": { "pgBouncer": { "image": "docker.io/percona/percona-
pgbouncer:1.25.2-1" } },
    "backups": { "pgbackrest": { "image": "docker.io/percona/percona-
pgbackrest:2.58.0-2" } }
  }
}'
```

The following image names in the above example were taken from the [list of certified images](#):

- `docker.io/percona/percona-distribution-postgresql-with-postgis:18.4-2,`
- `docker.io/percona/percona-pgbouncer:1.25.2-1,`
- `docker.io/percona/percona-pgbackrest:2.58.0-2`

- 4 After you applied the patch, the deployment rollout will be triggered automatically. The update process is successfully finished when all Pods have been restarted.



Expected output



- 5 Scan for indexes that rely on collations other than `C` or `POSIX` and whose collations were provided by the operating system (`c`) or dynamic libraries (`d`). [Connect to PostgreSQL](#) with the privileges of the superuser or the database owner and run the following query:

```
SELECT DISTINCT
    indrelid::regclass::text,
    indexrelid::regclass::text,
    collname,
    pg_get_indexdef(indexrelid)
FROM (
    SELECT
        indexrelid,
        indrelid,
        indcollation[i] coll
    FROM
        pg_index,
        generate_subscripts(indcollation, 1) g(i)
) s
JOIN pg_collation c ON coll = c.oid
WHERE
    collprovider IN ('d', 'c')
    AND collname NOT IN ('C', 'POSIX');
```

- 6 If you see the list of affected indexes, find the database names where indexes use a different collation version:

```
SELECT * FROM pg_database;
```

 **Sample output**



- 7 Refresh collation metadata and rebuild affected indexes. This command requires the privileges of a superuser or a database owner:

```
ALTER DATABASE cluster1 REFRESH COLLATION VERSION;
```

Major version upgrade

Major version upgrade allows you to jump from one database major version to another (for example, upgrade from PostgreSQL 17.x to PostgreSQL 18.x).

This feature is generally available starting with the Operator version 2.9.0.

Considerations

1. A major upgrade introduces a downtime because the whole cluster is shut down during the upgrade. This flow is planned to be improved in future releases.
2. During the upgrade, the Operator duplicates the data on each PVC and doesn't remove the old version data automatically. Make sure your PVC has enough free space to store data.
3. Starting with the Operator 2.6.0, PostgreSQL images are based on Red Hat Universal Base Image (UBI) 9 instead of UBI 8. UBI 9 has a different version of collation library `glibc` and this introduces a collation mismatch in PostgreSQL. Collation defines how text is sorted and compared based on language-specific rules such as case sensitivity, character order and the like. PostgreSQL stores the collation version used at database creation. When the collation version changes, this may result in corruption of database objects that use it like text-based indexes. Therefore, you need to identify and reindex objects affected by the collation mismatch.

Upgrade steps

To start the upgrade, you need to create a special `PerconaPGUpgrade` resource. This resource refers to the special *Operator upgrade image* and contains the information about the existing and target major versions. Find the example `PerconaPGUpgrade` configuration file in `deploy/upgrade.yaml`:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGUpgrade
metadata:
  name: cluster1-17-to-18
spec:
  postgresClusterName: cluster1
  image: docker.io/percona/percona-distribution-postgresql-upgrade:18.4-17.10-16.14-15.18-14.23-1
  fromPostgresVersion: 17
  toPostgresVersion: 18
  toPostgresImage: docker.io/percona/percona-distribution-postgresql:18.4-1
  toPgBouncerImage: docker.io/percona/percona-pgbouncer:1.25.2-1
  toPgBackRestImage: docker.io/percona/percona-pgbackrest:2.58.0-2
```



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

Apply this manifest to start the upgrade:

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



During the upgrade flow, the Operator:

1. Pauses the cluster, making it unavailable for the duration of the upgrade,
2. Annotates the cluster with a special `pgv2.percona.com/allow-upgrade: <PerconaPGUpgrade.Name>` annotation,
3. Creates jobs to migrate the data,
4. Starts up the cluster after the upgrade finishes.

Post-upgrade steps

1. The `pgaudit` extension is not upgraded automatically. If you use this extension, you must drop and recreate it **in each database** where it is installed. [Connect to PostgreSQL](#) with the privileges of the superuser and run the following commands:

```
DROP EXTENSION pgaudit;  
CREATE EXTENSION pgaudit;
```



Repeat for every database where the `pgaudit` extension is installed.

2. Scan for indexes that rely on collations other than `C` or `POSIX` and whose collations were provided by the operating system (`c`) or dynamic libraries (`d`). [Connect to PostgreSQL](#) with the privileges of the superuser or the database owner and run the following query:

```
SELECT DISTINCT
    indrelid::regclass::text,
    indexrelid::regclass::text,
    collname,
    pg_get_indexdef(indexrelid)
FROM (
    SELECT
        indexrelid,
        indrelid,
        indcollation[i] coll
    FROM
        pg_index,
        generate_subscripts(indcollation, 1) g(i)
) s
JOIN pg_collation c ON coll = c.oid
WHERE
    collprovider IN ('d', 'c')
    AND collname NOT IN ('C', 'POSIX');
```

3. If you see the list of affected indexes, find the database names where indexes use a different collation version:

```
SELECT * FROM pg_database;
```

 **Sample output**



4. Refresh collation metadata and rebuild affected indexes. This command requires the privileges of a superuser or a database owner:

```
ALTER DATABASE cluster1 REFRESH COLLATION VERSION;
```

Cleanup

1. You can remove old version data at your discretion by [executing into containers](#) and running the following commands (example for PostgreSQL 17):

```
rm -rf /pgdata/pg17
rm -rf /pgdata/pg17_wal
```

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

```
kubectl delete perconapgupgrade cluster1-17-to-18
```

Troubleshooting upgrade issues

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

Failed first restore after the upgrade

After a major upgrade, PostgreSQL starts a new WAL timeline. PostgreSQL treats the upgraded cluster as a new logical generation, so it increments the timeline ID and begins writing WAL from that new point.

In clusters with very low write traffic, the upgraded primary may generate very few WAL segments after the upgrade.

When you make a first restore after the upgrade, the restored replicas need to replay WAL from the primary to catch up. To do that, PostgreSQL uses the `pg_rewind` tool. `pg_rewind` searches a common WAL ancestor — a point in history where both the primary and the replica share the same WAL record. If there are few WAL records, there may not be a common WAL ancestor and the replica may fail to rejoin the primary. When this happens, you see the `could not find common ancestor of the source and target cluster's timelines` error in `pg_rewind`.

To address this issue, you must manually [reinitialize](#) the failed replica. Before doing so, check if this replica has any transactions that are not replicated anywhere else. Then remove its data directory and let the instance perform a full copy from the primary.

Alternatively, you can automate replica reinitialization with Patroni. Update the cluster configuration by setting the `spec.patroni.removeDataDirectoryOnDivergedTimelines` in the Custom Resource before the upgrade. When timeline divergence is detected, the Operator instructs Patroni to automatically remove the replica's data and resync it from the primary.

Warning

The `removeDataDirectoryOnDivergedTimelines` option can lead to data loss. When the Operator resyncs the replica automatically, some transactions may be lost. The risk is usually small but not zero. Use this option only if you understand and accept this trade-off.

Upgrade PostgreSQL extensions

Upgrade `pg_stat_monitor` (for Operator earlier than 2.6.0)

`pg_stat_monitor` is the built-in extension, which is used to provide query analytics for Percona Monitoring and Management (PMM). If you [enabled it](#) in the Custom Resource (`deploy/cr.yaml` manifest), you need to manually update it *after the database upgrade* (this manual step is not required for the Operator versions 2.6.0 and newer):

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

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

Sample output

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

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

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

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

3. Execute the following SQL statement:

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

Upgrade PostGIS extension

When you [upgrade your database](#) to a new version, this process does **not** automatically update the PostGIS extension inside PostgreSQL. You need to manually update the PostGIS extension in every database where it is enabled.

To do this, connect to PostgreSQL as a user with `SUPERUSER` privileges. You can use [the same user you used to enable the PostGIS extension](#). Then, execute the following SQL command for each relevant database:

```
SELECT PostGIS_Extensions_Upgrade();
```



Upgrade custom PostgreSQL extensions

If you have installed [custom PostgreSQL extensions](#), you need to build and package each custom extension for the new PostgreSQL major version. During the upgrade, the Operator will install extensions into the upgrade container.

Refer to the [Update custom extensions](#) section for step-by-step instructions.

Upgrade the Operator and the database on OpenShift

The upgrade on OpenShift consists of two steps:

- Upgrade the Operator Deployment
- Upgrade the database cluster

Upgrade the Operator via Operator Lifecycle Manager (OLM)

You can upgrade the Operator Deployment for PostgreSQL that was [installed on the OpenShift platform using OLM](#) directly through the Operator Lifecycle Manager.

If you know the OLM upgrade workflow, jump to the [Operator Deployment upgrade steps](#).

Understand how OLM applies Operator upgrades

OLM manages the Operator using a resource called a `ClusterServiceVersion` (CSV). Each CSV represents a specific version of the Operator and contains:

- the Operator Deployment specification
- required RBAC permissions
- CRD definitions
- metadata and examples

When a new Operator version is available and the upgrade is approved, OLM installs the new CSV and reconciles the Operator Deployment to match it. The following items are replaced with the values defined in the new CSV:

- container image
- command and arguments
- labels and annotations
- probes
- most Deployment fields

If you previously customized the Operator Deployment manually, these changes are overwritten during the upgrade.

The CRD may be updated too, if the new Operator version introduces schema changes. However, OLM doesn't modify the `PerconaPGCluster` Custom Resource. It remains unchanged and continues running with its current configuration. For how to update it, refer to [Upgrade Percona Distribution for PostgreSQL](#).

Persisting custom Operator configuration

If you need to customize the Operator Deployment (for example, to adjust resource limits or set environment variables), you can do it through the Subscription.

A Subscription is the OLM resource that defines which operator you want to install and how you want it to be upgraded. A Subscription connects your cluster to an Operator package in a CatalogSource and ensures that OLM continuously manages that Operator according to your chosen update strategy.

Here's how you can customize the Operator Deployment. This example command sets an environment variable for the Operator:

```
kubectl patch subscription percona-postgresql-operator -n <namespace> \
  --type merge \
  -p '{"spec":{"config":{"env":[{"name":"LOG_LEVEL","value":"DEBUG"}]}}}'
```



OLM supports overriding only the following fields through the Subscription:

- env
- envFrom
- volumes
- volumeMounts
- resources
- nodeSelector
- tolerations
- affinity

These overrides are applied on top of the CSV and persist across upgrades. All other fields are overridden by the values from the new CSV during the Operator Deployment upgrade.

Before you start

Before you upgrade the Operator deployment, prepare your environment depending on what catalogue you used to install it from:

- For **Community catalogues**, verify whether the Operator runs in single-namespace or all-namespaces mode and which namespace it targets. That helps you avoid reconciliation conflicts.

- For **Certified container images** (Red Hat / OperatorHub), the `stable` installation channel supports both single- and all-namespaces modes starting with version 3.0.0. The `stable-cw` channel is therefore deprecated. As the pre-upgrade steps, you must do the following:
 - Update the Subscription to the `stable` channel (recommended)
 - Confirm how the Operator watches namespaces
 - Align `initContainer.image` on each PostgreSQL cluster Custom Resource. Without updating `initContainer.image`, clusters may enter an error state after the Operator upgrade.

Community catalogues

1. Verify the installation mode and target namespace

OpenShift web console

Open the YAML manifest for the Operator installation. Verify the target namespace settings for the `targetNamespaces` option.

Command line

```
oc get operatorgroup -n <operator-namespace> -o yaml
```



Interpret the output:

- A **single-namespaces** install usually defines target namespaces explicitly:

```
spec:
  targetNamespaces:
    - <target-namespace>
```



- An **all-namespaces** install has no `spec.targetNamespaces` configured:

```
spec: {}
```



2. In **all-namespaces** mode, the Operator watches all namespaces on the cluster. If Percona PostgreSQL custom resources already exist outside the operator's installation namespace, the operator may start managing them after the upgrade. This may happen if you deployed several Operators in the same OpenShift cluster, all in the all namespace mode.

To avoid conflicts during reconciliation, repeat step 1 for **every** Operator you deployed. If more than one Operator is installed in all-namespaces mode, adjust the `spec.targetNamespaces` value for each OperatorGroup to the namespace this Operator must manage.

Certified container catalogues

1. Update the `initContainer` image on each cluster

You must manually set `spec.initContainer.image` on each `PerconaPGCluster` (pg) Custom Resource that this Operator manages. Without this, clusters may enter an error state after the Operator upgrade.

- a. Export namespaces you use for commands (use your Operator namespace and the namespace where each cluster CR lives; they may differ in multi-namespace setups):

```
export OPERATOR_NAMESPACE=postgres-operator
export CLUSTER_NAMESPACE=postgres-operator
```



- b. Retrieve the current Operator image to reuse for `initContainer.image`:

```
oc get deploy percona-postgresql-operator -n "$OPERATOR_NAMESPACE" \
-o jsonpath='{.spec.template.spec.containers[*].image}'
```



Find the `image` value in the output, for example:

```
registry.connect.redhat.com/percona/percona-postgresql-
operator@sha256:986941a8c5f5d00a0c9cc7bd12acc9f78aa51fdcc98c7d0acddff05392d4b9a0
```



- c. Patch each PostgreSQL cluster Custom Resource, replacing `cluster1` with your cluster name and using the correct cluster namespace:

```
oc patch pg cluster1 -n "$CLUSTER_NAMESPACE" --type=merge --patch '{
  "spec": {
    "initContainer": { "image": "<IMAGE_FROM_PREVIOUS_COMMAND>" }
  }
}'
```



- d. Repeat the previous command for every cluster managed by this Operator.

Upgrade the Operator

1. Log in to the OpenShift web console and check the list of installed Operators in your namespace to see if upgrades are available.

Installed Operators

Installed Operators are represented by ClusterServiceVersions within this Namespace.

Name ▾	Search by name...	
Name ↑		Status ↑
	Percona Operator for PostgreSQL 2.8.0 provided by Percona	✓ Succeeded ⬆ Upgrade available

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

Upgrade Percona Distribution for PostgreSQL

Before you start

1. We recommend to [update PMM Server](#) [↗](#) **before** upgrading PMM Client.
2. If you are using PMM server version 2, use a PMM client image compatible with PMM 2. If you are using PMM server version 3, use a PMM client image compatible with PMM 3. See [PMM upgrade documentation](#) [↗](#) for how to migrate from version 2 to version 3.

Upgrade steps

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

```
export CLUSTER_NAMESPACE=<my-namespace>
```



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

```
kubectl get deploy percona-postgresql-operator -n <operator-namespace> -o yaml
```



Expected output



3. [Apply a patch](#) [↗](#) to update your cluster’s Custom Resource. Set the `crVersion` field to match the Operator version and update the images as needed.

Depending on whether you've already updated the PMM client, either include its image in the list of images to update in your patch command or exclude the PMM client image from the patch.

If your cluster is named `cluster1`, use the following command as an example:

With PMM Client

```
kubectl patch pg cluster1 -n $CLUSTER_NAMESPACE --type=merge --patch '{
  "spec": {
    "crVersion": "3.0.0",
    "initContainer": { "image": "registry.connect.redhat.com/percona/percona-postgresql-
operator@sha256:ae9b319eaf3367f73d135fdda4ce69f58bcb9a2b05eea71903b7d631bd8b56c2" },
    "image": "registry.connect.redhat.com/percona/percona-postgresql-operator-
containers@sha256:2092b8badac196100a70e708e18ef6c70f9f398c99431f3905e8394b9cadd91a",
    "proxy": { "pgBouncer": { "image": "registry.connect.redhat.com/percona/percona-
postgresql-operator-
containers@sha256:73916031a5b9a033efdf86597b9df58837336ae208a8743d4c70874d459daeda"
} },
    "backups": { "pgbackrest": { "image": "registry.connect.redhat.com/percona/percona-
postgresql-operator-
containers@sha256:5937c9778be5c94acb4be81d979b6e5503f85dea1196f20f435b19467e56d1d0"
} },
    "pmm": { "image": "registry.connect.redhat.com/percona/percona-postgresql-operator-
containers@sha256:05dc00f69ed0fae48453476ace93bd43c046bf07a511cdca16e2fcad29c53805"
}
  }
}'
```

Without PMM Client

```
kubectl patch pg cluster1 -n $CLUSTER_NAMESPACE --type=merge --patch '{
  "spec": {
    "crVersion": "3.0.0",
    "initContainer": { "image": "registry.connect.redhat.com/percona/percona-postgresql-
operator@sha256:ae9b319eaf3367f73d135fdda4ce69f58bcb9a2b05eea71903b7d631bd8b56c2" },
    "image": "registry.connect.redhat.com/percona/percona-postgresql-operator-
containers@sha256:2092b8badac196100a70e708e18ef6c70f9f398c99431f3905e8394b9cadd91a",
    "proxy": { "pgBouncer": { "image": "registry.connect.redhat.com/percona/percona-
postgresql-operator-
containers@sha256:73916031a5b9a033efdf86597b9df58837336ae208a8743d4c70874d459daeda"
} },
    "backups": { "pgbackrest": { "image": "registry.connect.redhat.com/percona/percona-
postgresql-operator-
containers@sha256:5937c9778be5c94acb4be81d979b6e5503f85dea1196f20f435b19467e56d1d0"
} }
  }
}'
```

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

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



 Expected output



Migrate from Crunchy Operator for PostgreSQL

Migrate from Crunchy Postgres Operator to Percona Operator for PostgreSQL

If you run PostgreSQL on Kubernetes with the Crunchy Operator, migrating to Percona Operator for PostgreSQL is straightforward and low-risk because the operational model is familiar.

Why migrate to Percona Operator for PostgreSQL

Percona Operator for PostgreSQL is based on the Crunchy Operator codebase, so core workflows are already known and predictable for teams that use Crunchy today.

Key benefits include:

- **Truly open source, no vendor lock-in:** Percona Operator for PostgreSQL is a fully open source solution with the mission of keeping software free from proprietary licensing. You keep flexibility in tooling, support, and long-term platform decisions.
- **Known and predictable workflows:** Operational approach is familiar, enabling your team to start using Percona Operator for PostgreSQL without relearning day-2 operations from scratch.
- **One operating model across your database stack:** If you already run Percona Operators for MySQL or MongoDB, adding PostgreSQL gives you a consistent approach to monitoring, backups, upgrades, and security policy.
- **Lower load on SRE and DevOps teams:** A shared workflow across databases reduces operational complexity, context switching, and maintenance overhead.
- **Migration with native PostgreSQL techniques:** The migration methods use standard PostgreSQL approaches, making the process transparent and easier to validate.

Choose a migration method

This guide provides three migration methods. Pick the one that best matches your requirements for:

- acceptable downtime,
- rollback options,
- infrastructure overhead.

Review the method comparison first, then follow the detailed steps for your selected path.

Migration Method	Pros	Cons
Migrate using a standby cluster – set up a new standby cluster in Percona Operator and replicate from Crunchy	- Near-zero downtime - Validate new cluster	- Requires both clusters running in parallel

	while old remains online	- Higher resource usage
Migrate with backup and restore – back up data in Crunchy and restore it in Percona Operator	- Safer migration path - Allows test runs of migration - Can be rolled back	- Introduces downtime, which depends on data size
Migrate reusing data volumes – reuse the Persistent Volume Claims (PVCs) from a Crunchy cluster	- Simple and straightforward - No need to move large data over the network	- Requires downtime - Irreversible and requires thorough testing

Migrate from Crunchy to Percona Operator for PostgreSQL using a standby cluster

A standby cluster is a PostgreSQL cluster that doesn't have a writable primary. Instead, it applies changes from backups or from streaming replication from another database that acts as the source.

Using a standby cluster when migrating your PostgreSQL workloads, you maintain a live, synchronized copy of your original Crunchy cluster in Percona Operator for PostgreSQL. This approach minimizes downtime and gives you a way to validate the new cluster before switching over.

In this migration guide, we deploy Percona Operator for PostgreSQL as a standby cluster for a Crunchy PostgreSQL cluster by using a combination of backup and streaming replication. This approach ensures a fast initial data sync and keeps your data up to date with minimal downtime during cutover. For more information about standby cluster types, see [standby cluster types](#).

Then we cut over the Crunchy PostgreSQL cluster and promote Percona Operator for PostgreSQL.

This migration provides minimal downtime that happens only during the cutover.

Versions used in this guide

For this migration tutorial, we use the following versions of the Operators:

- Crunchy PostgreSQL Operator 5.8.x
- Percona Operator for PostgreSQL 3.x.x
- An AWS S3 storage for backups. If you use another remote storage, configure access to it for both clusters.

Prerequisites

Ensure you have the following:

- A Kubernetes or OpenShift cluster with permissions to create namespaces, CRDs, Roles, and Deployments.
- [Helm](#) 3.x.
- `kubectl` or `oc` command line tools
- [yq](#) YAML processor.
- A remote backup storage reachable from Pods in both the Crunchy and Percona namespaces.

The PostgreSQL **major version** on the Percona cluster must match the Crunchy source cluster.

Before you start

1. Create the namespaces for Crunchy Operator for PostgreSQL and Percona Operator for PostgreSQL and export them as the environment variables. If you already run Crunchy Operator for PostgreSQL, export its namespace as the environment variable.

```
export CRUNCHY_NS=postgres-operator
export NAMESPACE=pgo
```



2. Clone the repositories to edit the manifests during the migration setup:

- For Crunchy PostgreSQL Operator:

```
git clone https://github.com/CrunchyData/postgres-operator.git
```



- For Percona Operator for PostgreSQL:

```
git clone -b v3.0.0 https://github.com/percona/percona-postgresql-operator
```



Deploy Crunchy Operator for PostgreSQL

Skip this step if you already run Crunchy Operator for PostgreSQL.

If you don't, install the Operator Deployment following the [official Crunchy documentation](#)

Adapt the configuration for Crunchy PostgreSQL cluster

Whether you are deploying a new Crunchy PostgreSQL cluster or already have one running, make sure that:

1. `pgBackRest` is configured to send backups to a remote storage that will be accessible by the Percona standby cluster.
2. The Crunchy PostgreSQL cluster is exposed for external access so that the Percona cluster can connect to it for streaming replication. This typically involves making the PostgreSQL HA service accessible over the network.

To meet these requirements, do the following:

1. Create a Secret containing the access credentials for your backup storage. This example configuration uses AWS S3 as the backup storage and the `repo1` as the `pgBackRest` repository for storing backups there:

```
storage-access.yaml
```

```
apiVersion: v1
kind: Secret
metadata:
  name: crunchy-pgbackrest-secret
stringData:
  s3.conf: |
    [global]
    repo1-s3-key="<your-access-key>"
    repo1-s3-key-secret="<your-access-secret>"
```



For more configuration examples for other storages, see [Configure backup storage](#)

2. Apply the file to create the Secret:

```
kubectl apply -f storage-access.yaml -n $CRUNCHY_NS
```



3. Edit the `PostgresCluster` Custom Resource configuration to include the following configuration:

For `pgBackRest` configuration:

- The name of the Secret
- The name of the bucket
- The endpoint on the storage
- The shared path for both Crunchy and Percona standby clusters

For external access, specify the type `LoadBalancer` for the service.

Here's the example configuration for Crunchy PostgreSQL cluster showing these important settings:

```
postgres.yaml
```



```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: crunchy-source
spec:
  postgresVersion: 18
  instances:
    - name: instance1
      replicas: 3
      dataVolumeClaimSpec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 10Gi
  backups:
    pgbackrest:
      configuration:
        - secret:
            name: crunchy-pgbackrest-secret
  manual:
    repoName: repo1
    options:
      - --type=full
  global:
    repo1-path: /crunchy-to-percona/repo1
    repo1-s3-uri-style: path
    repo1-s3-verify-tls: "n" # Skip if you use AWS S3
  repos:
    - name: repo1
      s3:
        bucket: <YOUR_BUCKET_HERE>
        endpoint: s3.amazonaws.com
        region: <YOUR_REGION_HERE>
  proxy:
    pgBouncer:
      replicas: 1
```

4. Apply the configuration to create the new or update the existing Crunchy PostgreSQL cluster:



```
kubectl apply -f postgres.yaml -n $CRUNCHY_NS
```

5. Wait for the cluster to report the "Ready" status:



```
kubectl wait pod \
  -l postgres-operator.crunchydata.com/cluster=crunchy-source,postgres-
operator.crunchydata.com/data=postgres \
  -n "${CRUNCHY_NS}" \
  --for=condition=Ready \
  --timeout=300s
```

Sample output



Make a full backup on the Crunchy PostgreSQL cluster before the migration

Make a full backup. This ensures the standby cluster has a recent restore point and needs to replay a small amount of WAL files to synchronize.

1. Make a full backup of your Crunchy PostgreSQL cluster:

```
kubectl annotate postgrescluster crunchy-source \
  --namespace "${CRUNCHY_NS}" \
  postgres-operator.crunchydata.com/pgbackrest-backup="$(date +%s)"
```



2. Wait for the backup job to complete:

```
kubectl wait job \
  -l postgres-operator.crunchydata.com/cluster=crunchy-source \
  -l postgres-operator.crunchydata.com/pgbackrest-backup=manual \
  -n "${CRUNCHY_NS}" \
  --for=condition=complete \
  --timeout=600s
```



Expected output



Export data for the standby cluster

If the Percona cluster runs in a **different** namespace than Crunchy, copy the Crunchy TLS secrets into the Percona namespace for mutual TLS on the replication connection:

1. Retrieve the Secrets:

```
kubectl get secrets -n $CRUNCHY_NS
```



Expected output



The Secrets we are interested in are `crunchy-source-cluster-cert` and `crunchy-source-replication-cert`

2. Export Secrets to a file:

```
kubectl get secret crunchy-source-cluster-cert -o json -n "$CRUNCHY_NS" \
  | yq -o yaml '
{
  "apiVersion": .apiVersion,
  "kind": .kind,
  "data": .data,
  "metadata": {"name": .metadata.name},
  "type": .type
}' > ~/crunchy-source-cluster-cert.yaml

kubectl get secret crunchy-source-replication-cert -o json -n "$CRUNCHY_NS" \
  | yq -o yaml '
{
  "apiVersion": .apiVersion,
  "kind": .kind,
  "data": .data,
  "metadata": {"name": .metadata.name},
  "type": .type
}' > ~/crunchy-source-replication-cert.yaml
```

Deploy Percona Operator for PostgreSQL and PostgreSQL cluster in standby mode

1. Deploy Percona Operator for PostgreSQL:

```
kubectl apply --server-side -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/v3.0.0/deploy/bundle.yaml -n $NAMESPACE
```

2. Import the TLS Secrets using the configuration files you created earlier:

```
kubectl apply -f ~/crunchy-source-cluster-cert.yaml -n $NAMESPACE
kubectl apply -f ~/crunchy-source-replication-cert.yaml -n $NAMESPACE
```

3. Create the shared Secret for the backup storage. You can reuse the same Secret configuration file from the [Adapt the configuration for Crunchy PostgreSQL cluster](#) step:

```

echo "apiVersion: v1
kind: Secret
metadata:
  name: percona-pgbackrest-secret
stringData:
  s3.conf: |
    [global]
    repo1-s3-key=<your-access-key>
    repo1-s3-key-secret=<your-access-secret>
" | kubectl apply -n $NAMESPACE -f -

```

4. Edit the `deploy/cr.yaml` and specify the following configuration:

- For the standby cluster:
 - Set `standby.enabled` to `true`.
 - Specify the repository name to restore from in the `standby.repoName` option.
 - Specify the name of the service of the Crunchy PostgreSQL to stream data from. The Service name follows the pattern `<service-name>.<namespace>.svc.cluster.local`.
 - Specify the port of this service.

Replace example names such as `crunchy-source-ha`, `crunchy-source-pgbackrest`, and `cluster1-pgbouncer` if you use different cluster names. The service name `crunchy-source-ha` in the following example is derived from the source `PostgresCluster` name `crunchy-source`.

```

standby:
  enabled: true
  host: crunchy-source-ha.postgres-operator.svc.cluster.local
  port: 5432
  repoName: repo1

```

- For the mutual TLS communication, reference the TLS Secrets you created for the `secrets.customTLSSecret` and `secrets.customReplicationTLSSecret` options

```

secrets:
  customTLSSecret:
    name: crunchy-source-cluster-cert
  customReplicationTLSSecret:
    name: crunchy-source-replication-cert

```

- Configure the backup storage to exactly match the configuration on the Crunchy PostgreSQL cluster:

```
backups:
  pgbackrest:
    ....
    configuration:
      - secret:
          name: percona-pgbackrest-secret
    global:
      # Must match repo1-path in the Crunchy source cluster exactly.
      repo1-path: /crunchy-to-percona/repo1
      repo1-s3-uri-style: path
      repo1-s3-verify-tls: "n" # Skip for AWS S3
    repos:
      - name: repo1
        s3:
          bucket: <YOUR-BUCKET-HERE>
          endpoint: s3.amazonaws.com
          region: <YOUR-REGION-HERE>
#The rest of your configuration
```

5. Create the Percona PostgreSQL cluster

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

The Percona Operator deploys the cluster and:

- Restores the base backup from the S3 bucket.
- Replays WAL from S3 until it catches up with the live Crunchy cluster.
- Switches to streaming replication from `crunchy-source-ha`.

6. Wait for the cluster to report the `Ready` status:

```
kubectl wait perconapgcluster/cluster1 \
  -n $NAMESPACE \
  --for=jsonpath='{.status.state}'=ready \
  --timeout=600s
```

 Expected output



7. Verify the data is replicated on the standby.

- Export one of the Pods as an environment variable

```
STANDBY_POD=$(kubectl get pod -n "${NAMESPACE}" \
  -l postgres-operator.crunchydata.com/cluster=cluster1,postgres-
operator.crunchydata.com/data=postgres \
  -o jsonpath='{.items[0].metadata.name}')
```

- Connect to the pod and verify the cluster is replicating:

```
kubectl -n "${NAMESPACE}" exec "${STANDBY_POD}" -c database -- \
psql -t -c "SELECT pg_is_in_recovery(), pg_last_wal_replay_lsn();" 
```

You should see `t` for `pg_is_in_recovery()` and a non-null LSN.



Sample output



Verify streaming replication lag before cutover

On the Crunchy **primary** Pod, confirm the Percona standby is connected and lag is acceptable:

1. Identify the primary:

```
CRUNCHY_PRIMARY=$(kubectl get pod \
-l postgres-operator.crunchydata.com/cluster=crunchy-source,postgres-
operator.crunchydata.com/role=master \
-n "${CRUNCHY_NS}" \
-o jsonpath='{.items[0].metadata.name}')
```



2. Exec to the primary and verify the replication lag:

```
kubectl -n "${CRUNCHY_NS}" exec "${CRUNCHY_PRIMARY}" -c database -- \
psql -c "
SELECT client_addr, state,
       pg_wal_lsn_diff(sent_lsn, replay_lsn) AS byte_lag,
       write_lag, flush_lag, replay_lag
FROM pg_stat_replication;
"
```



Sample output



Proceed only when `write_lag` and `replay_lag` are null or within your policy (for example a few seconds).

Cut over the Crunchy PostgreSQL cluster

Now you are ready for the cutover. Note that this step causes downtime as both clusters will be in standby mode and will not accept writes.

1. Patch the Crunchy PostgreSQL cluster to the standby mode. Specify `repo1` so that Patroni archives the final WAL:

```
kubectl patch postgrescluster crunchy-source \
  -n $CRUNCHY_NS \
  --type=merge \
  -p '{"spec": {"standby": {"enabled": true, "repoName": "repo1"}}}'
```

2. Check that the former primary reports recovery:

```
kubectl -n $CRUNCHY_NS exec "${CRUNCHY_PRIMARY}" -c database -- \
  psql -t -c "SELECT pg_is_in_recovery();" 
```

 Expected output



(Optional) Shut down the Crunchy cluster

After the Percona standby has replayed WAL, shut down Crunchy to avoid split-brain:

```
kubectl patch postgrescluster crunchy-source \
  -n $CRUNCHY_NS \
  --type=merge \
  -p '{"spec": {"shutdown": true}}'
```

Promote Percona PostgreSQL cluster

Before promoting the Percona PostgreSQL cluster, make sure it has finished replaying all WAL (Write-Ahead Log) data.

After the Crunchy cluster is shut down, no new WAL files will be generated. The Percona standby will keep replaying existing WAL files from S3 object storage, and its Log Sequence Number (LSN) will advance each time new WAL is applied.

Once the LSN value stops changing and stays the same across two consecutive checks, it means all available WAL has been replayed and it's safe to promote the standby cluster to be the new primary.

1. Verify the Percona standby cluster has stopped replaying WAL files by running:

```
kubectl -n "${NAMESPACE}" exec "${STANDBY_POD}" -c database -- \
  psql -t -c "SELECT pg_last_wal_replay_lsn();" 
```

Repeat this command several times until the LSN value stops advancing.

2. Promote the Percona PostgreSQL cluster to be the new primary by patching the cluster:

```
kubectl patch perconapgcluster cluster1 \
  -n $NAMESPACE \
  --type=merge \
  -p '{"spec": {"standby": {"enabled": false}}}'
```

3. Wait for the cluster to report the `Ready` status:

```
kubectl wait perconapgcluster/cluster1 \
  -n $NAMESPACE \
  --for=jsonpath='{.status.state}'=ready \
  --timeout=600s
```

4. Confirm the cluster is not in a recovery mode and can accept writes:

- Identify the primary pod:

```
PERCONA_PRIMARY=$(kubectl get pod -n $NAMESPACE \
  -l postgres-operator.crunchydata.com/cluster=cluster1,postgres-
operator.crunchydata.com/role=primary \
  -o jsonpath='{.items[0].metadata.name}')
```

- Connect to the Pod and check if it is in the recovery mode:

```
kubectl -n $NAMESPACE exec "${PERCONA_PRIMARY}" -c database -- \
  psql -t -c "SELECT pg_is_in_recovery();" 
```

You should see `f`. This means the instance is read-write.

5. After promoting the Percona cluster, it is important to confirm that the `pgBackRest` stanza has been created. The stanza is required for managing backups and enabling restores; without it, regular backup operations and recovery from backup will not be possible. Waiting for `stanzaCreated` to become true ensures that backup tooling has been initialized properly before proceeding.

```
kubectl wait perconapgcluster/cluster1 \
  -n "${NAMESPACE}" \
  --for=jsonpath='{.status.pgbackrest.repos[0].stanzaCreated}'=true \
  --timeout=300s
```

 Expected output



Take a post-migration backup

This creates a clean recovery point on the new timeline. Note that after you make the backup, you can no longer roll back to the Crunchy PostgreSQL.

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

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGBBackup
metadata:
  name: post-migration-backup
spec:
  pgCluster: cluster1
  repoName: repo1
```

2. Apply the configuration to start the backup:

```
kubectl apply -f deploy/post-migration-backup.yaml -n "${NAMESPACE}"
```

3. Wait for the backup job to complete:

```
kubectl wait perconapgbbackup/post-migration-backup \
  -n "${NAMESPACE}" \
  --for=jsonpath='{.status.state}'=Succeeded \
  --timeout=600s
```

Update the application connection string

After you migrated to Percona Operator for PostgreSQL, update your application connection string to use the new Percona cluster:

1. Get the `pgBouncer` service exposed by the Percona Operator:

```
kubectl get service -n $NAMESPACE cluster1-pgbouncer
```

The output will show the `cluster1-pgbouncer` service, along with its ClusterIP and port.

1. Update your application's connection string to use the `pgBouncer` service name (from the previous step) as the PostgreSQL host. For example:

```
Host:    cluster1-pgbouncer.$NAMESPACE.svc.cluster.local
Port:    5432
```

Replace `${NAMESPACE}` with your cluster's namespace. Update your application configuration to point to this service, ensuring it now connects to your Percona Operator for PostgreSQL deployment.

Rollback

The standby path is the most rollback-friendly of the [migration options](#): Crunchy and Percona share the same `pgBackRest` repository layout until you establish a new baseline on Percona.

Before the post-migration backup

Until you complete [Take a post-migration backup](#), you can usually return write traffic to Crunchy if the Crunchy `PostgresCluster` and its `pgBouncer` Service still exist and you have **not** deleted `crunchy-source`.

Any writes that landed on `cluster1` **after** you promoted Percona are **not** on Crunchy. Treat rollback as abandoning that window unless you copy data out manually.

1. Stop application writes to `cluster1`.
2. Put `cluster1` back into standby so it stops acting as the primary (Patroni will follow the Custom Resource). The `standby.host`, `standby.port`, and `standby.repoName` you configured earlier should still be present; merge only toggles standby on:

```
kubectl patch perconapgcluster cluster1 \
  -n "${NAMESPACE}" \
  --type=merge \
  -p '{"spec": {"standby": {"enabled": true, "repoName": "repo1"}}}'
```



3. If you applied [\(Optional\) Shut down the Crunchy cluster](#), clear shutdown before promoting Crunchy again:

```
kubectl patch postgrescluster crunchy-source \
  -n "${CRUNCHY_NS}" \
  --type=merge \
  -p '{"spec": {"shutdown": false}}'
```



4. Take Crunchy out of standby so Patroni can promote it back to primary:

```
kubectl patch postgrescluster crunchy-source \
  -n "${CRUNCHY_NS}" \
  --type=merge \
  -p '{"spec": {"standby": {"enabled": false}}}'
```



5. Confirm roles: Crunchy should accept writes and `cluster1` should be in recovery until replication catches up.



```
CRUNCHY_PRIMARY=$(kubectl get pod \
  -l postgres-operator.crunchydata.com/cluster=crunchy-source,postgres-
operator.crunchydata.com/role=master \
  -n "${CRUNCHY_NS}" \
  -o jsonpath='{.items[0].metadata.name}')

kubectl -n "${CRUNCHY_NS}" exec "${CRUNCHY_PRIMARY}" -c database -- \
  psql -t -c "SELECT pg_is_in_recovery();"

```

You should see `f` on Crunchy.



```
STANDBY_POD=$(kubectl get pod -n "${NAMESPACE}" \
  -l postgres-operator.crunchydata.com/cluster=cluster1,postgres-
operator.crunchydata.com/data=postgres \
  -o jsonpath='{.items[0].metadata.name}')

kubectl -n "${NAMESPACE}" exec "${STANDBY_POD}" -c database -- \
  psql -t -c "SELECT pg_is_in_recovery();"

```

You should see `t` on Percona while it follows Crunchy.

6. Point applications back at the Crunchy `pgBouncer` Service:



```
kubectl get service -n "${CRUNCHY_NS}" \
  -l postgres-operator.crunchydata.com/cluster=crunchy-source,postgres-
operator.crunchydata.com/role=pgbouncer

```

After the post-migration backup

After you take the post-migration backup, timelines and `pgBackRest` history on Percona have diverged from the pre-cutover Crunchy archive. You should treat Crunchy as a historical reference, not a live paired primary.

Rolling back from here is the same idea as a fresh restore: deploy or restore a Crunchy cluster from the repository state you need, or follow [Migrate from Crunchy using backup and restore](#) in reverse order for your environment. Plan the rollback **before** cutover if you require a guaranteed fast path.

Troubleshooting

Percona standby cannot resolve or reach the Crunchy primary

Confirm the PostgreSQL HA Service for `crunchy-source` resolves from the Percona database Pod. Replace `crunchy-source-ha` if your Service name differs.



```
kubectl -n "${NAMESPACE}" exec "${STANDBY_POD}" -c database -- \
  bash -c "getent hosts crunchy-source-ha.${CRUNCHY_NS}.svc.cluster.local"

```

If DNS fails, check Service names and namespaces. If DNS succeeds but TCP fails, check NetworkPolicies, service mesh rules, and firewalls between namespaces.

Replication authentication errors

The Percona standby connects to Crunchy as the PostgreSQL role `_crunchyreplication` using the replication TLS material. Ensure `crunchy-source-replication-cert` exists in `"${NAMESPACE}"` and matches what Crunchy generated (re-export from `"${CRUNCHY_NS}"` if you rotated certificates).

```
kubectl get secret crunchy-source-replication-cert -n "${NAMESPACE}"
```



Verify the `PerconaPGCluster` still references it under `secrets.customReplicationTLSSecret` and `secrets.customTLSSecret` as in [Deploy Percona Operator for PostgreSQL and PostgreSQL cluster in standby mode](#).

pgBackRest restore or WAL archive problems

Symptoms include failed restore Jobs, missing `archive.info`, or the standby never leaving base backup replay.

1. Confirm `repo1-path` is **identical** on Crunchy and on `cluster1` (for example `/crunchy-to-percona/repo1` in this guide).

```
kubectl get postgrescluster crunchy-source -n "${CRUNCHY_NS}" \
  -o jsonpath='{.spec.backups.pgbackrest.global.repo1-path}'

kubectl get perconapgcluster cluster1 -n "${NAMESPACE}" \
  -o jsonpath='{.spec.backups.pgbackrest.global.repo1-path}'
```



2. Confirm both clusters use credentials that can read and write the same bucket (equivalent Secrets). See [Configure backup storage](#) for TLS flags such as `repo1-s3-verify-tls` and path-style settings.
3. From a throwaway Pod in `"${NAMESPACE}"`, verify object storage reachability with your endpoint, bucket, and keys (adapt for your provider):



```
kubect1 run -i --rm s3-check \  
  --image=perconalab/awsc1i \  
  --restart=Never \  
  -n "${NAMESPACE}" \  
  -- bash -c "  
    AWS_ACCESS_KEY_ID='<your-access-key>' \  
    AWS_SECRET_ACCESS_KEY='<your-secret-key>' \  
    AWS_DEFAULT_REGION='<your-region>' \  
    aws --endpoint-url 'https://<your-s3-endpoint>' \  
      --no-verify-ssl \  
      s3 ls s3://<your-bucket>  
  "
```

Timeline history file missing after promotion (Crunchy archive)

Some Crunchy PGO configurations using async archive mode can hit a missing timeline history file on the archive after promotion (see [Crunchy PGO issue #4472](#) ). If `pgBackRest` or replicas complain about `*.history` on the archive, push the file once from the **new** primary:



```
kubect1 -n "${NAMESPACE}" exec "${PERCONA_PRIMARY}" -c database -- \  
  bash -c "  
    pgbackrest --stanza=db --no-archive-async \  
      archive-push "\"${PGDATA}/pg_wal/00000002.history\" || true  
  "
```

Adjust the filename to match the timeline file reported in your logs.

Migrate from Crunchy to Percona Operator for PostgreSQL using backup and restore

You can migrate by taking a full `pgBackRest` backup on your Crunchy `PostgresCluster` and creating a new `PerconaPGCluster` that restores from that backup before its first start. This path is straightforward when you can accept downtime between the final backup and an application cutover.

Benefits	Trade-offs
- A clear cutover point (the migration backup) - Room to rehearse restore and validation before switching applications - The option to roll back to Crunchy until you decommission the source cluster and redirect traffic	- Any data written after the migration backup and before cutover to Percona is not carried over unless you take another backup and re-bootstrap. - Downtime spans at least that backup window plus restore and validation time.

For **near-zero downtime**, use [Migrate from Crunchy using a standby cluster](#) instead.

Versions used in this guide

For this migration tutorial, we use the following versions of the Operators:

- Crunchy PostgreSQL Operator 5.8.x
- Percona Operator for PostgreSQL 3.x.x
- An AWS S3 storage for backups. If you use another remote storage, configure access to it for both clusters.

Prerequisites

Ensure you have the following:

- A Kubernetes or OpenShift cluster with permissions to create namespaces, CRDs, Roles, and Deployments.
- [Helm](#) 3.x (if you install operators or optional in-cluster object storage with Helm).
- `kubectl` or `oc`.
- `yq` YAML processor (optional, for manifest tricks used in other migration guides).
- A remote backup storage reachable from Pods in both the Crunchy and Percona namespaces.

The PostgreSQL **major version** on the Percona cluster must match the Crunchy source cluster.

Before you start

1. Create or choose namespaces for Crunchy and Percona and export them.

```
export CRUNCHY_NS=postgres-operator
export NAMESPACE=pgo
```



2. Clone the Percona Operator repository at the release you will deploy so you can base your `PerconaPGCluster` manifest on the default `deploy/cr.yaml`:

```
git clone -b v3.0.0 https://github.com/percona/percona-postgresql-operator
```



Configure Crunchy PostgreSQL cluster for remote backups

If you already run Crunchy PGO, configure the `PostgresCluster` object so that `pgBackRest` writes backups and WAL (Write-Ahead Logs) to remote storage that the Percona cluster can access and read. This ensures that Percona can restore from your latest backups when you perform the migration.

If you are deploying Crunchy for testing or migration rehearsal, make sure your `PostgresCluster` manifest configures `pgBackRest` to use the same remote object storage (for example, S3 or S3-compatible, Google Cloud Storage, or compatible endpoint) for both backups and WAL archives. The Percona PostgreSQL cluster will need access to this storage location for a successful restore.

This involves:

- Creating a Secret with the required credentials for your object storage (e.g., S3 access keys or GCS service account).
- Referencing that Secret under the `spec.backups.pgbackrest.repos` section in your Crunchy `PostgresCluster` manifest.
- Setting options such as `repo1-path`, bucket, region, endpoint, and other storage details to match what you plan to use on the Percona side under `spec.dataSource.pgbackrest`.
- Ensuring network access between Crunchy and Percona namespaces to the object storage endpoint.

See the [standby migration guide](#) for an example manifest that creates the required Secret and configures remote backups and WAL archiving.

Here's the example configuration of the Crunchy `PostgresCluster` object:

```
postgres.yaml
```

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: crunchy-source
spec:
  postgresVersion: 18
  instances:
    - name: instance1
      replicas: 3
      dataVolumeClaimSpec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 10Gi
  backups:
    pgbackrest:
      configuration:
        - secret:
            name: crunchy-pgbackrest-secret
      manual:
        repoName: repo1
        options:
          - --type=full
      global:
        repo1-path: /crunchy-to-percona/repo1
        repo1-s3-uri-style: path
        repo1-s3-verify-tls: "n"
      repos:
        - name: repo1
          s3:
            bucket: <YOUR-BUCKET-HERE>
            endpoint: s3.amazonaws.com
            region: <YOUR-REGION-HERE>
  proxy:
    pgBouncer:
      replicas: 1
```



Apply the configuration to create a new or update an existing cluster:

```
kubectl apply -f postgres.yaml -n $CRUNCHY_NS
```



Wait for the cluster to report the “Ready” status:

```
kubectl wait pod \
  -l postgres-operator.crunchydata.com/cluster=crunchy-source,postgres-
operator.crunchydata.com/data=postgres \
  -n "${CRUNCHY_NS}" \
  --for=condition=Ready \
  --timeout=300s
```



Expected output



Wait for the `pgBackRest` stanza to be created. The stanza initializes the backup and restore infrastructure for further backups or restore operations.

```
kubectl wait postgrescluster/crunchy-source \
  -n "${CRUNCHY_NS}" \
  --for=jsonpath='{.status.pgbackrest.repos[0].stanzaCreated}'=true \
  --timeout=300s
```



Expected output



The `stanzaCreated` status confirms that `pgBackRest` has scanned the configuration and the data directory and is ready to manage backups for your cluster.

Create the migration backup

Add a few test records to your database, then trigger a full backup.

This backup is the restore point for `cluster1` on Percona. **Stop application writes** before you trigger it if you need a consistent cutover; otherwise treat everything after this backup as disposable until cutover.

1. Identify the primary Pod of the the Crunchy PostgreSQL cluster and export it as an environment variable:

```
CRUNCHY_PRIMARY=$(kubectl get pod \
  -l postgres-operator.crunchydata.com/cluster=crunchy-source,postgres-
operator.crunchydata.com/role=master \
  -n "${CRUNCHY_NS}" \
  -o jsonpath='{.items[0].metadata.name}')
```



2. Connect to the Crunchy PostgreSQL cluster and insert some sample data. Use the following command to exec to the primary Pod, create a database and a table and insert some data to it:

```
kubectl exec -n "${CRUNCHY_NS}" "${CRUNCHY_PRIMARY}" -c database -- bash -c "
  psql -c 'CREATE DATABASE migrationtest;'
  psql -d migrationtest -c 'CREATE TABLE migration_data (id int PRIMARY KEY,
value text);'
  psql -d migrationtest -c \"INSERT INTO migration_data VALUES (1, 'initial-
data-before-migration');\"
"
```



3. Trigger a full backup:

```
kubectl annotate postgrescluster crunchy-source \
  -n "${CRUNCHY_NS}" \
  postgres-operator.crunchydata.com/pgbackrest-backup="$(date +%s)"
```

4. Wait for the backup Job to complete:

```
kubectl wait job \
  -l postgres-operator.crunchydata.com/pgbackrest-backup=manual,postgres-
operator.crunchydata.com/cluster=crunchy-source \
  -n "${CRUNCHY_NS}" \
  --for=condition=Complete \
  --timeout=600s
```

 Expected output



Prepare pgBackRest credentials in the Percona namespace

Create a Secret in `$NAMESPACE` with the **same** credentials and storage information such as endpoint and region that you used to configure `pgBackRest` in the Crunchy PostgreSQL cluster.

The following example command creates a Secret with the name `percona-pgbackrest-secret` for the Percona cluster; keep that name if you copy the manifest below.

```
kubectl apply -n "$NAMESPACE" -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: percona-pgbackrest-secret
stringData:
  s3.conf: |
    [global]
    repo1-s3-key="<your-access-key>"
    repo1-s3-key-secret="<your-access-secret>"
EOF
```

If you use different Secret names, use your name consistently in

`spec.dataSource.pgbackrest.configuration` and `spec.backups.pgbackrest.configuration` on `PerconaPGCluster` `cluster1`.

Deploy Percona Operator for PostgreSQL

1. Deploy Percona Operator for PostgreSQL:

```
kubect1 apply --server-side -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/v3.0.0/deploy/bundle.yaml -n $NAMESPACE
```

2. Wait for the Percona Operator Deployment to become available:

```
kubect1 wait deployment percona-postgresql-operator \
  -n "${NAMESPACE}" \
  --for=condition=Available \
  --timeout=120s
```

 Expected output



Create the Percona PostgreSQL cluster from the Crunchy backup

1. Edit the `deploy/cr.yaml` manifest and specify the following:
 - a. Set `spec.postgresVersion` to the same major version as `crunchy-source`.
 - b. The `spec.dataSource.pgbackrest` configuration to reference the Secret with the credentials to the object store and to point at the **same** repository path and object store settings as the Crunchy cluster's `repo1`. (See [Understand the dataSource options](#) for more information about these options.)
 - c. Under `spec.backups.pgbackrest`, use a **different** `repo1-path` than the Crunchy cluster so new backups do not overwrite the migration archive.

The following fragment shows the important pieces; merge them with the rest of `deploy/cr.yaml` (images, users, resources) for your environment:



```
metadata:
  name: cluster1
spec:
  ....
  dataSource:
    pgbackrest:
      stanza: db
      configuration:
        - secret:
            name: percona-pgbackrest-secret
      global:
        # Must exactly match the pgBackrest settings defined in Crunchy
`spec.backups.pgbackrest.global.repo1-path`
        repo1-path: /crunchy-to-percona/repo1
        repo1-s3-uri-style: path
        repo1-s3-verify-tls: "n" # Omit or set to "y" for AWS S3 with valid TLS
      repo:
        name: repo1
        s3:
          bucket: <YOUR_BUCKET_HERE>
          endpoint: s3.amazonaws.com
          region: <YOUR_REGION_HERE>
  backups:
    pgbackrest:
      configuration:
        - secret:
            name: percona-pgbackrest-secret
      global:
        # Must be different from the Crunchy repo1-path
        repo1-path: /percona/cluster1/repo1
        repo1-s3-uri-style: path
        repo1-s3-verify-tls: "n"
      repos:
        - name: repo1
        # The storage is shared, use the same settings here
        s3:
          bucket: <YOUR_BUCKET_HERE>
          endpoint: s3.amazonaws.com
          region: <YOUR_REGION_HERE>
  # The rest of your configuration
```

2. Apply the manifest:



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

3. Wait until the cluster reports the `Ready` state:

```
kubectl wait perconapgcluster/cluster1 \
  -n "${NAMESPACE}" \
  --for=jsonpath='{.status.state}'=ready \
  --timeout=300s
```



The Percona Operator deploys the cluster and bootstraps it from the storage referenced in the `dataSource` section.

4. Verify that the data was restored successfully:

- Identify the primary Pod and export it as an environment variable:

```
PERCONA_PRIMARY=$(kubectl get pod -n "${NAMESPACE}" \
  --selector postgres-operator.crunchydata.com/cluster=cluster1,postgres-
operator.crunchydata.com/role=primary \
  -o jsonpath='{.items[0].metadata.name}')
```



- Verify the data insertion:

```
kubectl exec -n "${NAMESPACE}" "${PERCONA_PRIMARY}" -c database -- bash -c \
  "psql -q -t -d migrationtest -c 'SELECT id, value FROM migration_data ORDER BY
  id;'"
```



??? example “Expected output”

```
```{.sql .no-copy}
1 | initial-data-before-migration
```
```



- Exec into the Pod and check that it is not in recovery state:

```
kubectl -n "${NAMESPACE}" exec "${PERCONA_PRIMARY}" -c database -- \
  psql -t -c "SELECT pg_is_in_recovery();"
```



You should see `f`: the instance is read-write.

5. Wait until the new cluster’s `pgBackRest` stanza exists for ongoing backups:

```
kubectl wait perconapgcluster/cluster1 \
  -n "${NAMESPACE}" \
  --for=jsonpath='{.status.pgbackrest.repos[0].stanzaCreated}'=true \
  --timeout=300s
```



 Expected output



Take a post-migration backup

Create a baseline backup on Percona's own repository path so future restores do not depend on the Crunchy archive.

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

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGBBackup
metadata:
  name: post-migration-backup
spec:
  pgCluster: cluster1
  repoName: repo1
```



2. Apply the configuration to start the backup:

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



3. Wait for the backup job to complete:

```
kubectl wait perconapgbbackup/post-migration-backup \
  -n "${NAMESPACE}" \
  --for=jsonpath='{.status.state}'=Succeeded \
  --timeout=600s
```



Point applications to Percona

1. Discover the `pgBouncer` Service for `cluster1`:

```
kubectl get service -n "${NAMESPACE}" \
  -l postgres-operator.crunchydata.com/cluster=cluster1,postgres-
operator.crunchydata.com/role=pgbouncer
```



2. Update your application's connection string to use the `pgBouncer` service name (from the previous step) as the PostgreSQL host. For example:

```
Host:    cluster1-pgbouncer.${NAMESPACE}.svc.cluster.local
Port:    5432
```



Replace `${NAMESPACE}` with your cluster's namespace. Update your application configuration to point to this service, ensuring it now connects to your Percona Operator for PostgreSQL deployment.

Decommission the Crunchy cluster

After traffic runs on Percona and you are satisfied with validation, you can delete the Crunchy PostgreSQL cluster:

```
kubectl delete postgrescluster crunchy-source -n "${CRUNCHY_NS}"
```



Uninstall Crunchy Operator from its namespace when no `PostgresCluster` resources remain.

Rollback

Until you move write traffic to `cluster1`, rollback is to point applications back at the Crunchy `pgBouncer` Service. After cutover, new writes exist only on Percona; reconciling them back to Crunchy is a manual process.

After you delete the `crunchy-source` cluster, rollback means restoring a new Crunchy cluster from the **original** `repo1` content in object storage, if you still have that repository.

Troubleshooting

`archive.info` missing or restore errors

The value of `spec.dataSource.pgbackrest.global.repo1-path` must match the Crunchy cluster's `spec.backups.pgbackrest.global.repo1-path` exactly.

```
kubectl get postgrescluster crunchy-source -n "${CRUNCHY_NS}" \
-o jsonpath='{.spec.backups.pgbackrest.global.repo1-path}{"\n"}'

kubectl get perconapgcluster cluster1 -n "${NAMESPACE}" \
-o jsonpath='{.spec.dataSource.pgbackrest.global.repo1-path}{"\n"}'
```



Restore job failures (TLS, endpoint)

`pgBackRest` uses HTTPS for S3-compatible endpoints. Fix TLS trust (correct CA or `repo1-s3-verify-tls`) and confirm network reachability from the restore Pod to the bucket endpoint. See [Configure backup storage](#).

Cluster stuck in restore

Inspect restore Pods or Jobs and `pgBackRest` logs:

```
kubect1 logs \  
  -l postgres-operator.crunchydata.com/cluster=cluster1,postgres-  
operator.crunchydata.com/pgbackrest-restore=cluster1 \  
  -n "${NAMESPACE}" \  
  -c pgbackrest
```



Wrong or incomplete data

If you need a fresher cutover point, stop writes, take another full backup on Crunchy, delete `cluster1` on Percona, and re-apply the restore manifest so it bootstraps again from the newer backup.

Migrate from Crunchy to Percona Operator for PostgreSQL by reusing persistent volumes

This guide shows how to migrate to Percona Operator for PostgreSQL using the **primary** PostgreSQL data volume from your Crunchy `PostgresCluster`. In this approach, Percona `PerconaPGCluster` (`cluster1`) starts on the existing `PGDATA` data directory without running a backup-restore cycle over the network.

| Benefits | Trade-offs |
|--|--|
| <ul style="list-style-type: none">- Avoids large data copies through object storage when the persistent volume already contains the dataset.- Provides a direct migration path when it is important to keep the exact data directory intact.- Allows migrating even if the source cluster is deleted but the PV is preserved | <ul style="list-style-type: none">- Downtime occurs from the moment you stop writes on Crunchy until the Percona cluster is brought up.- Rollback is difficult once Percona has advanced the database on that volume; take a Crunchy <code>pgBackRest</code> backup before destructive steps if you may need to restore Crunchy.- Requires manual management of PV reclaim policy, clearing <code>claimRef</code>, and setting a PVC selector to ensure the new claim binds only to the intended volume. |

If the backing PV was already deleted with the default **Delete** reclaim policy, use [Migrate from Crunchy using backup and restore](#) instead.

Versions used in this guide

For this migration tutorial, we use the following versions of the Operators:

- Crunchy PostgreSQL Operator 5.8.x
- Percona Operator for PostgreSQL 3.x.x

Prerequisites

Ensure you have the following:

- A Kubernetes or OpenShift cluster with permissions to patch PersistentVolumes, delete `PostgresCluster` resources, and deploy the Percona Operator.
- `kubectl` or `oc`.
- Storage where you can set the PV reclaim policy to `Retain` and clear `claimRef` (cluster-admin privileges are often required).

The PostgreSQL **major version** on the Percona cluster must match the Crunchy source cluster.

Before you start

1. Create or choose namespaces for Crunchy and Percona and export them as the environment variables.

```
export CRUNCHY_NS=postgres-operator
export NAMESPACE=pgo
```



2. Clone the Operator repository to edit the manifests during the migration setup:

```
git clone -b v3.0.0 https://github.com/percona/percona-postgresql-operator
```



Deploy Operators

1. Install Crunchy Operator for PostgreSQL into `$CRUNCHY_NS` if it is not already installed. See the [Crunchy quickstart](#)  for the steps.
2. Install Percona Operator for PostgreSQL into `$NAMESPACE`:

```
kubectl apply --server-side \
  -f https://raw.githubusercontent.com/percona/percona-postgresql-
operator/v3.0.0/deploy/bundle.yaml \
  -n "${NAMESPACE}"
```



3. Wait for the Percona Operator Deployment to become available:

```
kubectl wait deployment percona-postgresql-operator \
  -n "${NAMESPACE}" \
  --for=condition=Available \
  --timeout=120s
```



Expected output



Deploy the Crunchy source cluster

If you already have a PostgreSQL cluster running in the Crunchy namespace, make sure it is set up so that only one database server (replica) is running. This means changing the configuration so `instances.replicas` is set to `1`.

If you are deploying Crunchy for testing or migration rehearsal, create it with **one** Postgres instance (`replicas: 1` on the primary set) so a single data PVC represents the primary.

This is the sample `PostgresCluster` Custom Resource configuration:

postgres.yaml

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: crunchy-source
spec:
  postgresVersion: 18
  instances:
    - name: instance1
      replicas: 1
      dataVolumeClaimSpec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 10Gi
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - ReadWriteOnce
              resources:
                requests:
                  storage: 10Gi
  proxy:
    pgBouncer:
      replicas: 1
```

1. Apply the configuration to create or update the Crunchy PostgreSQL cluster.
2. Wait for the cluster to report the `Ready` status:

```
kubectl wait pod \
  -l postgres-operator.crunchydata.com/cluster=crunchy-source,postgres-
operator.crunchydata.com/data=postgres \
  -n "${CRUNCHY_NS}" \
  --for=condition=Ready \
  --timeout=300s
```

 Expected output



Stop writes and identify the primary's Persistent Volume

Stop application from writing to the database. This marks the **start of downtime**.

Next, identify the primary Pod, its data PVC, and backing PV

1. Identify the primary Pod:

```
CRUNCHY_PRIMARY=$(kubectl get pod -n "${CRUNCHY_NS}" \
  --selector postgres-operator.crunchydata.com/cluster=crunchy-source,postgres-
operator.crunchydata.com/role=master \
  -o jsonpath='{.items[0].metadata.name}')
```

2. Retrieve the PVC:

```
PVC_NAME=$(kubectl get pod -n "${CRUNCHY_NS}" "${CRUNCHY_PRIMARY}" \
  -o jsonpath='{.spec.volumes[?(@.name=="postgres-
data")].persistentVolumeClaim.claimName}')
```

3. Retrieve the PV this PVC is bound to:

```
PV_NAME=$(kubectl get pvc -n "${CRUNCHY_NS}" "${PVC_NAME}" \
  -o jsonpath='{.spec.volumeName}')
```

4. Verify the retrieved data:

```
echo "Primary pod: ${CRUNCHY_PRIMARY}"
echo "PVC:        ${PVC_NAME}"
echo "PV:         ${PV_NAME}"
```

 Sample output



If your Crunchy Pod uses a volume name other than `postgres-data`, adjust the JSONPath filter.

Retain the PV and remove the Crunchy cluster

The default reclaim policy for a dynamically PersistentVolume is `Delete`. This means that as soon as there are no Persistent Volume Claims (PVC) bound to this PV, Kubernetes deletes it and all data is lost. To prevent this, change the PV reclaim policy to `Retain`.

1. Set the PV reclaim policy to `Retain` so the disk survives PVC deletion:

```
kubectl patch pv "${PV_NAME}" \
  -p '{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'
```



2. Patch the Crunchy `PostgresCluster` object to remove finalizers. Kubernetes “finalizers” block resource deletion until certain cleanup steps finish, such as the operator deleting related pods, PVCs, and secrets. Since we want to keep the PersistentVolume (PV) and avoid extra cleanup (like deleting the PV or data), we remove the finalizers to force Kubernetes to delete the `PostgresCluster` resource immediately. This lets us proceed with detaching the data volume for migration without waiting for the Crunchy Operator’s normal deletion behavior.

```
kubectl patch postgrescluster crunchy-source -n "${CRUNCHY_NS}" \
  --type=json \
  -p='[{"op":"remove","path":"/metadata/finalizers"}]'
```



3. Now, delete the `PostgresCluster`:

```
kubectl delete postgrescluster crunchy-source -n "${CRUNCHY_NS}"
```



4. Uninstall Crunchy Operator from `$CRUNCHY_NS` when no `PostgresCluster` resources remain.
5. After the data PVC is gone, the PV enters the `Released` state. However, it still references the old claim and cannot be claimed by a new PVC. Therefore, you need to clear its `claimRef`:

```
kubectl patch pv "${PV_NAME}" --type=json \
  -p='[{"op":"remove","path":"/spec/claimRef"}]'
```



6. Wait until the PV status becomes `Available`:

```
kubectl wait pv "${PV_NAME}" \
  --for=jsonpath='{.status.phase}'=Available \
  --timeout=60s
```



Expected output



7. Label the PV so the Percona PVC selector binds to it exclusively, avoiding accidental binding to another released volume:

```
kubectl label pv "${PV_NAME}" percona-pv-migration=migrated
```



Create the Percona cluster bound to the retained volume

1. Edit the `deploy/cr.yaml` and specify the following settings:

- Give a name to your cluster in the `metadata.name` (for example, `cluster1`)
- Align `spec.postgresVersion` with the old cluster.
- For the first cutover, set **one** replica on the instance that should attach to the migrated disk
- Add a `selector` on `dataVolumeClaimSpec` matching the PV label. This ensures the controller binds the PVC to the correct PV that has the specified label.

Here is the example configuration:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGCluster
metadata:
  name: cluster1
spec:
  crVersion: 3.0.0
  postgresVersion: 18
  instances:
    - name: instance1
      replicas: 1
      dataVolumeClaimSpec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 10Gi
        selector:
          matchLabels:
            percona-pv-migration: migrated
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - ReadWriteOnce
              resources:
                requests:
                  storage: 10Gi
```

Tune `storage`, `storageClassName`, and other fields to match your environment and the Operator's usual `deploy/cr.yaml` requirements (users, images, resources).

2. Apply the manifest:

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

The Percona Operator creates a PersistentVolumeClaim (PVC) using the selector you specified in the manifest. This PVC binds to the labelled PersistentVolume (PV) that contains the migrated PostgreSQL data. PostgreSQL then starts directly on the existing `PGDATA` directory, so no backup restore, WAL download, or object storage is needed.

By default, `pgBackRest` uses a local, PVC-backed repository (`repo1.volume`) for backups and restores, so you do not need to configure S3 credentials or external storage for this migration path (unless you choose to use S3 for backups).

3. Wait for the cluster to have the `Ready` status:

```
kubectl wait perconapgcluster/cluster1 \
  -n "${NAMESPACE}" \
  --for=jsonpath='{.status.state}'=ready \
  --timeout=600s
```



Expected output



4. Verify that the data is intact by checking the primary pod state:

- Identify the primary pod:

```
PERCONA_PRIMARY=$(kubectl get pod -n "${NAMESPACE}" \
  --selector postgres-operator.crunchydata.com/cluster=cluster1,postgres-
operator.crunchydata.com/role=primary \
  -o jsonpath='{.items[0].metadata.name}')
```



- Check the primary pod state:

```
kubectl -n "${NAMESPACE}" exec "${PERCONA_PRIMARY}" -c database -- \
  psql -t -c "SELECT pg_is_in_recovery();" 
```



Expect `f` for a writable primary.

Scale out the cluster

You started the cluster with a single replica, so the Operator created a PersistentVolumeClaim (PVC) that bound to your labelled PersistentVolume (PV) containing the migrated data. Once you have confirmed that the primary instance is healthy, scale out your cluster for high availability.

While the selector is present in the cluster configuration, it blocks the Operator to create new replica PVCs because no other PV carries `percona-pv-migration=migrated`. Therefore, you need to remove the selector. Doing so enables the Operator to create new PVCs for each replica using your default StorageClass.

1. Update the cluster configuration by removing the selector and increasing the number of replicas:

```
kubectl patch perconapgcluster cluster1 \
  --namespace "${NAMESPACE}" \
  --type=json \
  -p='[
    {"op":"remove","path":"/spec/instances/0/dataVolumeClaimSpec/selector"},
    {"op":"replace","path":"/spec/instances/0/replicas","value":3}
  ]'
```

2. Wait for the cluster to report the `Ready` status:

```
kubectl wait perconapgcluster/cluster1 \
  --namespace "${NAMESPACE}" \
  --for=jsonpath='{.status.state}'=ready \
  --timeout=300s
```

 Expected output



Take a post-migration backup

Create a new baseline for further point-in-time restores by making a full backup. You can reuse the volume-based backup repository, such as `repo1`, that this guide uses to keep local backups on persistent storage within your cluster. Or see [Configure backup storage](#) to stream backups there.

1. Edit the `deploy/backup.yaml`. This example configures backups to a volume-based backup repository following the cluster configuration in this guide:

deploy/backup.yaml

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGBBackup
metadata:
  name: post-migration-backup
spec:
  pgCluster: cluster1
  repoName: repo1
```

2. Apply the configuration to start a backup:

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

3. Wait for the backup Job to complete:

```
kubectl wait perconaapgbbackup/post-migration-backup \
  -n "${NAMESPACE}" \
  --for=jsonpath='{.status.state}'=Succeeded \
  --timeout=600s
```



Point applications to Percona

1. Discover the `pgBouncer` Service for `cluster1`:

```
kubectl get service -n "${NAMESPACE}" \
  -l postgres-operator.crunchydata.com/cluster=cluster1,postgres-
operator.crunchydata.com/role=pgbouncer
```



2. Update your application's connection string to use the `pgBouncer` service name (from the previous step) as the PostgreSQL host. For example:

```
Host:    cluster1-pgbouncer.${NAMESPACE}.svc.cluster.local
Port:    5432
```



Replace `${NAMESPACE}` with your cluster's namespace. Update your application configuration to point to this service, ensuring it now connects to your Percona Operator for PostgreSQL deployment.

Cleanup

Remove the migration label from the PV after you no longer need to pin binding:

```
kubectl label pv "${PV_NAME}" percona-pv-migration-
```



Rollback

PV reuse is the least rollback-friendly path: once `cluster1` runs on the old directory, the previous Crunchy timeline is no longer authoritative. If you need a fallback, take a full Crunchy `pgBackRest` backup **before** you delete `crunchy-source`, and treat that backup as your recovery source for a new Crunchy deployment.

Troubleshooting

PVC stays Pending

Confirm the PV label and phase:

```
kubectl get pv "${PV_NAME}" --show-labels
kubectl get pv "${PV_NAME}" -o jsonpath='{.status.phase}'{"\n"}
```



PostgreSQL fails to start

Inspect database container logs:

```
kubectl -n "${NAMESPACE}" logs "${PERCONA_PRIMARY}" -c database
```



Unclean shutdowns may require crash recovery; follow Patroni and PostgreSQL logs for progress.

PV was deleted

If the dynamic provisioner deleted the volume when the PVC disappeared, data is gone from that path. Use [Migrate from Crunchy using backup and restore](#) from your latest remote backup instead.

How-to

Install Percona Operator for PostgreSQL with customized parameters

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

To check available configuration options, see [deploy/cr.yaml](#) and [Custom Resource Options](#).



To customize the configuration when installing with `kubect1`, do the following:

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

```
git clone -b v3.0.0 https://github.com/percona/percona-postgresql-operator
```



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

```
kubect1 apply -f deploy/cr.yaml -n postgres-operator
```



You can install the Operator deployment and the Percona Distribution for PostgreSQL cluster with custom parameters using Helm. Find what options you can customize in the [Operator chart documentation](#) and the [Percona Distribution for PostgreSQL chart documentation](#).

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

Using `--set` flags

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

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

```
helm install my-db percona/pg-db --version 3.0.0 --namespace my-namespace \
  --set postgresVersion=18.4-1 \
  --set pmm.enabled=true
```



Using a `values.yaml` file

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

```
helm install my-db percona/pg-db --version 3.0.0 --namespace my-namespace -f values.yaml
```

Example `values.yaml`:

```
postgresVersion: 18.4-1
pmm:
  enabled: true
```

Naming conventions for Helm resources

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

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

| Option | Effect | Example |
|-------------------------------|--|---|
| <code>nameOverride</code> | Replaces the chart name but keeps the release name in the generated name | <code>release-name-name-override</code> |
| <code>fullnameOverride</code> | Replaces the entire generated name with the specified value | <code>fullname-override</code> |

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

```
helm install my-operator percona/pg-operator --namespace my-namespace \
--set nameOverride=postgres-operator
```

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

```
helm install cluster1 percona/pg-db -n my-namespace \
--set nameOverride=postgres
```

Cluster name: `cluster1-postgres`.

Using `fullnameOverride` – replaces the full resource name:

```
helm install my-operator percona/pg-operator --namespace my-namespace \
--set fullnameOverride=percona-postgresql-operator
```



Deployment name: `percona-postgresql-operator`.

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



Cluster name: `my-db`.

Cluster name length

For the pg-db chart, the cluster name is limited to 21 characters, must consist of lowercase alphanumeric characters, '-' or '.', and must start and end with an alphanumeric character. Keep this in mind when using `fullnameOverride` or long release names.

Common Helm values reference

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

| Value | Charts | Description |
|---------------------------------|---|---|
| <code>nameOverride</code> | pg-operator , pg-db | Replaces the chart name in generated resource names |
| <code>fullnameOverride</code> | pg-operator , pg-db | Replaces the entire generated resource name |
| <code>watchAllNamespaces</code> | pg-operator | Deploy the Operator in cluster-wide mode to watch all namespaces |
| <code>disableTelemetry</code> | pg-operator | Disable telemetry collection. See Telemetry for details |

How to run initialization SQL commands at cluster creation time

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

psql interactive terminal [will execute](#)  these initialization statements when the cluster is created, [after creating custom users and databases](#) specified in the Custom Resource.

To set SQL initialization sequence you need creating a special [ConfigMap](#) , and reference this ConfigMap in the `databaseInitSQL` subsection of your Custom Resource options.

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

1. Create YAML manifest for the ConfigMap as follows:

my_init.yaml

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



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

Create the ConfigMap by applying your manifest:

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



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

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



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

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



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



Note

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

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



Change the PostgreSQL primary instance

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

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

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

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

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

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

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



Sample output



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

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

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

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



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

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



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

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



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

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

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



 Sample output



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

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



 Sample output



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



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

Use Docker images from a private registry

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

Prerequisites

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

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



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

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

The token can be found with the following command:

```
oc whoami -t
AD08CqCDappWR4hxjFDqwijEHei31yXAvWg61Jg210s
```



And the following one tells you the registry IP address:

```
kubectl get services/docker-registry -n default
```

| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) | AGE |
|-----------------|-----------|----------------|-------------|----------|-----|
| docker-registry | ClusterIP | 172.30.162.173 | <none> | 5000/TCP | 1d |



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

```
docker login -u admin -p AD08CqCDappWR4hxjFDqwijEHei31yXAvWg61Jg210s
172.30.162.173:5000
```



Expected output



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

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



Expected output



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

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

```
docker tag \  
    docker.io/perconalab/percona-postgresql-  
operator@sha256:991d6049059e5eb1a74981290d829a5fb4ab0554993748fde1e67b2f46f26bf0 \  
    172.30.162.173:5000/psmdb/percona-postgresql-operator:18.4-1  
docker push 172.30.162.173:5000/pg/percona-postgresql-operator:18.4-1
```



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

```
oc get is
```



Expected output



7. When the custom registry image is available, edit the the `image:` option in `deploy/operator.yaml` configuration file with a Docker Repo + Tag string (it should look like `docker-registry.default.svc:5000/pg/percona-postgresql-operator:18.4-1`)

Note

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

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

Manage PostgreSQL extensions

One of the specific PostgreSQL features is the ability to provide it with additional functionality via [Extensions](#) . Percona Distribution for PostgreSQL [comes with a number of extensions](#) . These extensions are available for the database cluster managed by the Operator as well.

Built-in extensions

You can enable or disable built-in extensions in the `extensions.builtin` section of your `deploy/cr.yaml` file. Set an option to `true` to enable an extension, or to `false` to disable it. To see which extensions are enabled by default, check the [deploy/cr.yaml](#)  Custom Resource manifest.

```
extensions:
  ...
  builtin:
    pg_stat_monitor: false
    pg_audit: true
    pgvector: false
    pg_repack: false
```



Apply changes after editing with `kubectl apply -f deploy/cr.yaml` command. This causes the Operator to restart the Pods of your cluster.

Add custom extensions

The needed extension may not be in the list of extensions supplied with Percona Distribution for PostgreSQL, or it's a custom extension developed by the end-user. To add such a custom extension is not straightforward in a containerized database in a Kubernetes environment. It requires building a custom PostgreSQL image.

Starting with version 2.3, the Operator provides an alternative way to extend Percona Distribution for PostgreSQL by downloading pre-packaged extensions from an external storage on the fly.



Advanced configuration

Custom extensions configuration is an advanced feature that requires careful consideration. Adding custom extensions may violate the immutability of Pod images, which can lead to unexpected behavior and maintenance challenges. Use this feature only if you are certain what you are doing and understand the implications. Or [reach out to our experts](#) for assistance with adding custom extensions into your infrastructure.

Here's how it works:

1. You build and package a custom extension. The package must have a strict structure. See [Packaging requirements](#) for details.

2. You upload the extension to a cloud storage.
3. In the `extensions` section of the Custom Resource, specify the storage configuration and the extension information.
4. The Operator downloads the extension and installs it.
5. In PostgreSQL, you create the extension for every database where you want to use it.

Understanding which files are required for a given extension may not be easy. To figure this out, you can spin up a Docker container or a virtual machine, install Percona Distribution for PostgreSQL and developer tools there, then build and install the extension from source. Then copy all the installed files to the archive.

Check the [Example configuration](#) for the steps that can help you in building and adding your own custom extension.

Packaging requirements

Custom extensions require specific packaging for the Operator to use them. The package must be a `.tar.gz` archive that follows this naming format:

```
${EXTENSION}-pg${PG_MAJOR}-${EXTENSION_VERSION}
```

The archive must be created with `usr` at the root and must include all the required files in the correct directory structure:

1. The control file and any shared library must be in the `LIBDIR` directory
2. All required SQL script files must be in the `SHAREDIR/extension` directory. At least one SQL script is required.

The `SHAREDIR` corresponds to `/usr/pgsql-${PG_MAJOR}/share` and `LIBDIR` to `/usr/pgsql-${PG_MAJOR}/lib`.

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

```
tree ~/pg_cron-1.6.7/
/home/user/pg_cron-1.6.7/
├── usr
│   ├── pgsql-17
│   │   ├── lib
│   │   │   └── pg_cron.so
│   │   └── share
│   │       └── extension
│   │           ├── pg_cron--1.0--1.1.sql
│   │           ├── pg_cron--1.0.sql
│   │           ├── pg_cron--1.1--1.2.sql
│   │           ├── pg_cron--1.2--1.3.sql
│   │           ├── pg_cron--1.3--1.4.sql
│   │           ├── pg_cron--1.4--1.4-1.sql
│   │           ├── pg_cron--1.4-1--1.5.sql
│   │           ├── pg_cron--1.5--1.6.sql
│   │           └── pg_cron.control
```

The resulting `.tar` archive has the name `pg_cron-pg17-1.6.7.tar.gz`.

Example configuration

The following is an **example workflow** showing how to build and package the `pg_cron` extension. This example is intended to illustrate the general process and give you an idea of the required steps. However, the exact workflow and specifics may differ for your custom extension. Always review your extension's build and packaging requirements and adapt accordingly.

Considerations

1. You must build your extension on a host **with the same operating system and architecture** as the one used for Percona Distribution for PostgreSQL images to prevent library incompatibility. Otherwise, your extension may not load or may not function correctly.

To check the operating system, do the following:

- a. Connect to one of the database Pods:

```
kubectl exec -it cluster1-instance1-xrcf-0 -n <namespace> -c database -- bash
```

- b. List the installed packages:

```
rpm -qa|grep percona
```

 **Sample output**



- c. Check the operating system version:

```
cat /etc/redhat-release
```



 Sample output



2. Your extension must be compatible with PostgreSQL version you are running. To check the version, run the following command:

```
kubectl -n <namespace> get pg cluster1 -o go-template='{{.spec.image}}'
```



 Sample output



3. In this example configuration, we use a Docker container to build the `pg_cron` extension. However, you can use any environment that matches the distribution's operating system, such as a virtual machine or a Kubernetes Pod, not just Docker.
4. We assume you have deployed a Percona Distribution for PostgreSQL cluster in Kubernetes. If not, use the [Quickstart guide](#) to deploy it.

Prepare your build environment

Run the following commands as the root user or with `sudo` privileges.

1. Start a Docker container and establish a shell session inside. In this example we use a Red Hat Universal Base Image 9 on `x86_64` architecture.

```
docker run -it --name pg redhat/ubi9:latest /bin/bash
```



2. Install basic tools:

```
dnf install git make 'dnf-command(config-manager)'
```



3. Install additional PostgreSQL packages:

- Add the Extra Packages for Enterprise Linux by installing the `epel-release` package:

```
dnf install -y https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
```



- Add the codeready builder repository that contains additional packages for use by developers:

```
dnf config-manager --add-repo https://dl.rockylinux.org/pub/rocky/9/CRB/x86_64/os
```



- Import GPG keys

```
rpm --import https://dl.rockylinux.org/pub/rocky/RPM-GPG-KEY-Rocky-9
```

- Install `perl-IPC-Run` to run and interact with child processes:

```
dnf install perl-IPC-Run -y
```

4. Install build tools:

```
dnf groupinstall "Development tools"
```

Troubleshooting tip: If development tools fail to install, add BaseOS and AppStream repos:

```
dnf config-manager --add-repo  
https://dl.rockylinux.org/pub/rocky/9/BaseOS/x86_64/os/  
dnf config-manager --add-repo  
https://dl.rockylinux.org/pub/rocky/9/AppStream/x86_64/os/  
dnf clean all && dnf makecache
```

Then retry the installation.

5. Install PostgreSQL developer packages from Percona repositories:

- Install `percona-release` repository management tool:

```
dnf install https://repo.percona.com/yum/percona-release-latest.noarch.rpm
```

- Enable PostgreSQL repository:

```
percona-release setup pg17
```

- Disable the `postgresql` module supplied with the operating system:

```
dnf -qy module disable postgresql
```

- Install PostgreSQL developer packages:

```
dnf install percona-postgresql17-devel percona-postgresql17-libs percona-  
postgresql17
```

Build the extension

1. Download the extension source:

```
git clone https://github.com/citusdata/pg_cron.git
```

2. Navigate to the cloned extension and switch to the desired version. In this example we use version 1.6.7:

```
cd pg_cron
git checkout v1.6.7
```

3. Ensure `pg_config` is in your path:

```
export PATH=/usr/pgsql-17/bin:$PATH
```

4. Build and install the extension

```
make && sudo PATH=$PATH make install
```

As the result you should see the binaries in the following paths: `/usr/pgsql-17/share/extension/pg_cron` and `/usr/pgsql-17/lib/`.

Package the extension

1. Create a `.tar` archive of the extension:

```
tar -czvf pg_cron-pg17-1.6.7.tar.gz \
  /usr/pgsql-17/lib/pg_cron.so \
  /usr/pgsql-17/share/extension/pg_cron*
```

2. Check that the package structure follows the [requirements](#).
3. Copy the archive to the local machine. Run this command on the local machine:

```
docker cp pg:/pg_cron-pg17-1.6.7.tar.gz ./
```

Upload a custom extension to the cloud storage

After packaging the extension, upload it to a cloud storage. In our example we use AWS S3 storage. You can upload the extension via the Amazon web interface or using the `aws` command line tool as shown below:

1. Export the AWS S3 access credentials as the environment variables:

```
export AWS_ACCESS_KEY_ID=<your-access-key-id-here>
export AWS_SECRET_ACCESS_KEY=<your-secret-key-here>
```

2. Upload the extension to your storage. Use your value for the bucket and specify your path to the archive:

```
aws s3 cp path/to/pg_cron-pg17-1.6.7.tar.gz s3://my-bucket
```



Create a Secret with the storage credentials

After the upload is complete, place the access credentials for the cloud storage in a Secret.

1. Create a Secrets file with the credentials that the Operator needs to access extensions stored on Amazon S3:

- The `metadata.name` key is the name you will use to refer to your Kubernetes Secret.
- The `data.AWS_ACCESS_KEY_ID` and `data.AWS_SECRET_ACCESS_KEY` keys contain base64-encoded credentials used to access the storage.

To encode credentials, use this command:

in Linux

For GNU/Linux:

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



in macOS

For Apple macOS:

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



Here's the example Secrets file `extensions-secret.yaml`:

extensions-secret.yaml

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



2. Create the Secrets object from this file:

```
kubectl apply -f extensions-secret.yaml -n <namespace>
```



Configure the Operator to load and install the custom extension

Specify both the storage and extension details in the Custom Resource so the Operator can download and install it.

1. In the `extensions.storage` subsection of the Custom Resource, specify the following information:
 - storage details such as the bucket where your extension resides, region and endpoint to access the storage
 - the Secret name with the storage credentials that you created before.

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



2. In the `extensions.custom` subsection, specify the extension name and version:

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



3. Some extensions (such as `pg_cron` in our example) may require additional shared memory. If this is the case, you need to configure PostgreSQL to preload it at startup:

```
```yaml ... patroni: dynamicConfiguration: postgresql: parameters: shared_preload_libraries: pg_cron ...
```

4. Apply the configuration:

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



This causes the Operator to restart the Pods of your cluster.

## Enable custom extension in PostgreSQL

The installed extension is not enabled by default. You need to explicitly enable it in PostgreSQL for all databases where you want to use it.

Here's how to do it:

1. Connect to the primary Pod:

```
kubectl exec -it cluster1-instance1-69r8-0 -c database -n <namespace> -- bash
```



2. Connect to the required database in PostgreSQL and create the extension for this database using the `CREATE EXTENSION` statement:

```
CREATE EXTENSION pg_cron;
```



## Update custom extensions

To update your custom extension inside the Operator, do the following:

1. Prepare the `*.tar` archive of the extension's new version. See the [Packaging requirements](#) section for the archive's structure and naming format
2. Reference the new version of the extension in the Custom Resource. For example, you update `pg_cron` extension to version 1.6.8. Then your configuration looks like this:

```
extensions:
 ...
 custom:
 - name: pg_cron
 version: 1.6.8
```



3. Apply the configuration for the changes to come into place:

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



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

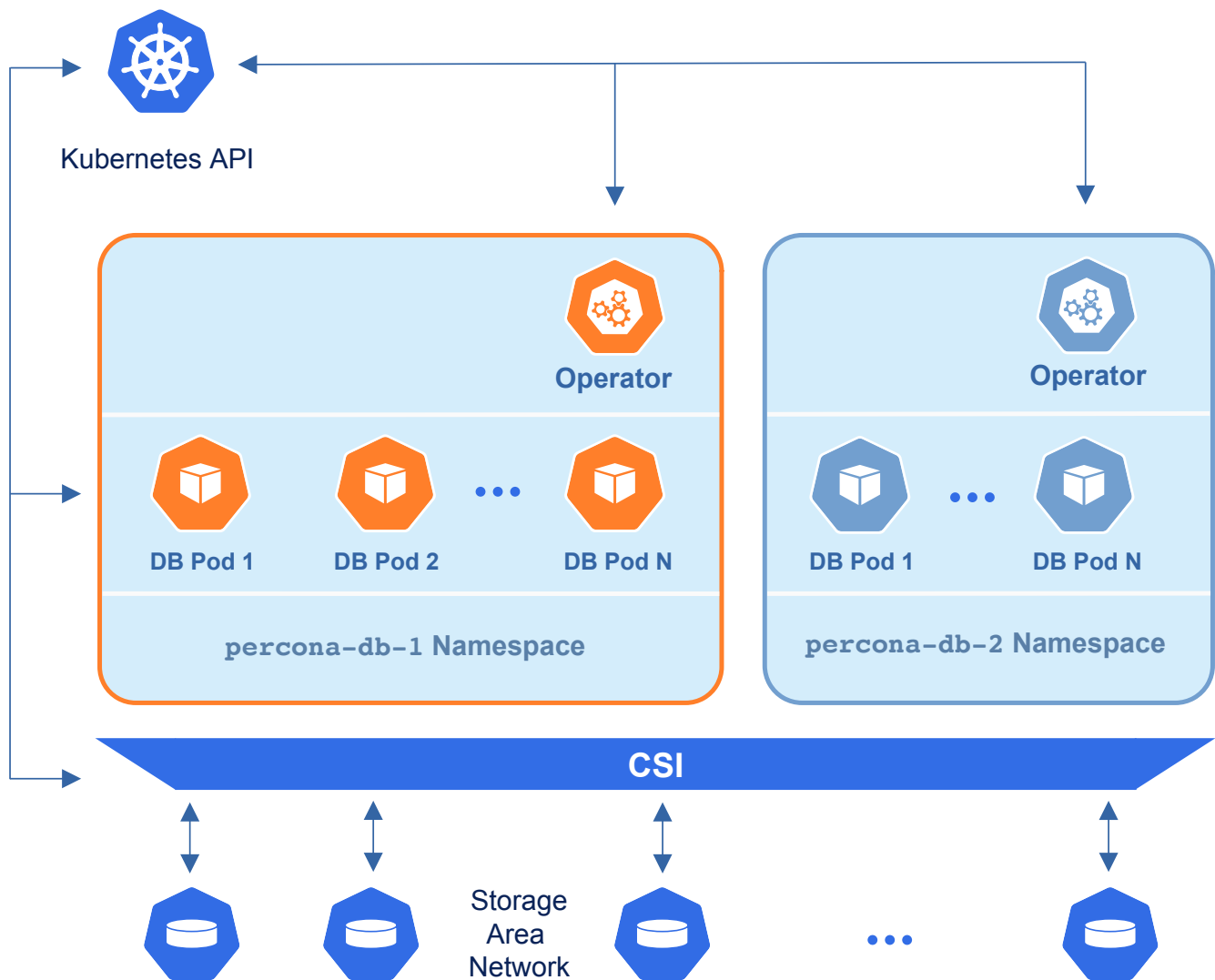
There are two design patterns that you can choose from when deploying Percona Operator for PostgreSQL and PostgreSQL clusters in Kubernetes:

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

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

## Namespace-scope

By default, Percona Operator for PostgreSQL functions in a specific Kubernetes namespace. You can create one during the installation (like it is shown in the [installation instructions](#)) or just use the default namespace. This approach allows several Operators to co-exist in one Kubernetes-based environment, being separated in different namespaces:



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

Let's say you will use a Kubernetes Namespace called `percona-db-1`.

1. Clone `percona-postgresql-operator` repository:

```
git clone -b v3.0.0 https://github.com/percona/percona-postgresql-operator
cd percona-postgresql-operator
```



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

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



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

```
kubectl apply --server-side -f deploy/bundle.yaml -n percona-db-1
```



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

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



You can deploy multiple clusters in this namespace.

## Add more namespaces

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

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

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



2. Deploy the Operator:

```
kubectl apply --server-side -f deploy/bundle.yaml -n percona-db-2
```



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

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



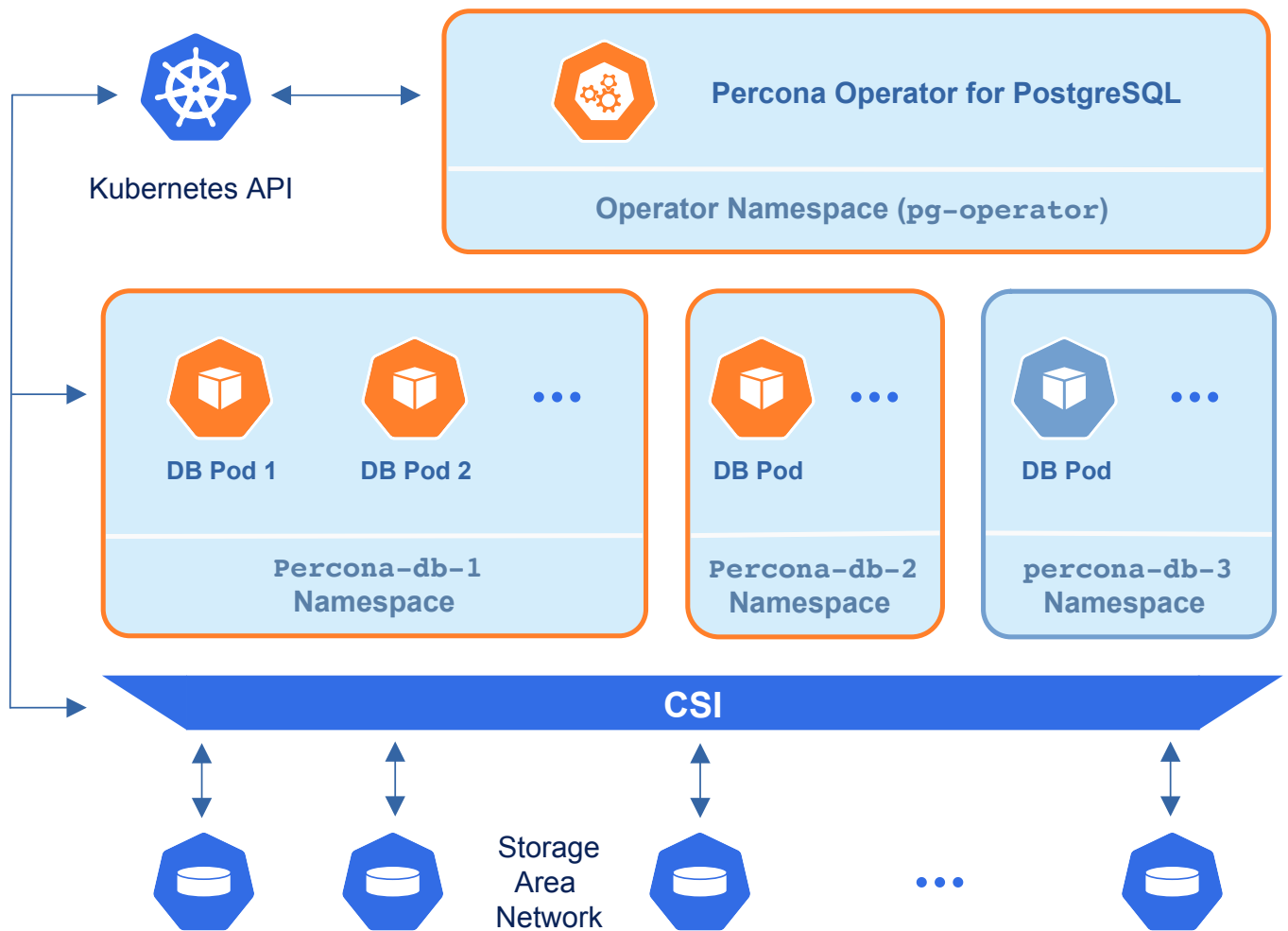
### Note

Cluster names may be the same in different namespaces.

## Install the Operator cluster-wide

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

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



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

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

- `subjects.namespace` option should contain the namespace which will host the Operator,
- The `WATCH_NAMESPACE` environment variable determines which namespaces the Operator will monitor for custom resources. By default, it is defined by the `metadata.namespace` option via a downward API `fieldRef` and points to the Operator's own namespace.

Configure the `WATCH_NAMESPACE` environment variable in the Operator Deployment's `env` section and choose the option that matches your desired Operator scope:

- **Comma-separated list of namespaces:** Specify the comma-separated list of namespaces, including the namespace where the Operator itself is running (e.g., `"pg-operator, percona-db-1"`). The Operator will watch only the specified namespaces.
- **Empty string (""):** Set the `value` to an empty string. The Operator watches **all namespaces** in the cluster.

For the full list of Operator environment variables, see [Define Operator environment variables](#).

 **Note**

Installing the Operator cluster-wide on OpenShift via the the Operator Lifecycle Manager (OLM) requires [making different selections in the OLM web-based UI](#) instead of patching YAML files.

The following simple example shows how to install Operator cluster-wide on Kubernetes.

1. Clone `percona-postgresql-operator` repository:

```
git clone -b v3.0.0 https://github.com/percona/percona-postgresql-operator
cd percona-postgresql-operator
```

2. Let's say you will use `pg-operator` namespace for the Operator, and `percona-db-1` namespace for the cluster. Create these namespaces, if needed:

```
kubectl create namespace pg-operator
kubectl create namespace percona-db-1
```

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

```
...
subjects:
- kind: ServiceAccount
 name: percona-postgresql-operator
 namespace: pg-operator
...
spec:
 containers:
 - env:
 - name: WATCH_NAMESPACE
 value: "pg-operator,percona-db-1"
...
```

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

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

With this configuration, the Operator monitors only the specified namespaces.

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

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

# Verifying the cluster operation

When creation process is over, connect to the cluster to verify it is operational.

When the Operator deploys a database cluster, it generates several [Secrets](#). Among them there is the Secret with the credentials of the default PostgreSQL user. This default user has the same username as the cluster name.

1 Use `kubectl get secrets -n <namespace>` command to see the list of Secrets objects. The Secrets object you are interested in is named in the format `<cluster_name>-pguser-<cluster_name>` (where the `<cluster_name>` is the [name of your Percona Distribution for PostgreSQL Cluster](#)). For example, if your cluster name is `cluster1`, the Secret name will be `cluster1-pguser-cluster1`.

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

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

3 To connect to PostgreSQL, you will use the `pgbouncer` service as the entry point to your cluster. To find this service, use the following command:

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

Look for the service named `<cluster-name>-pgbouncer`.

 Sample output



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

```
kubectl run -n <namespace> -i --rm --tty pg-client --image=percona/percona-
distribution-postgresql:18.4-1 --restart=Never -- bash -il
```

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

5 Run a container with `psql` tool and connect its console output to your terminal. Substitute the `<namespace>` placeholder with your value in the following command to connect as a `cluster1` user to the `cluster1` database via the PostgreSQL interactive terminal.

```
[postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer.
<namespace>.svc.cluster.local -p 5432 -U cluster1 cluster1
```



Sample output



# Using PostgreSQL tablespaces with Percona Operator for PostgreSQL

Tablespaces allow DBAs to store a database on multiple file systems within the same server and to control where (on which file systems) specific parts of the database are stored. You can think about it as if you were giving names to your disk mounts and then using those names as additional parameters when creating database objects.

PostgreSQL supports this feature, allowing you to *store data outside of the primary data directory*, and Percona Operator for PostgreSQL is a good option to bring this to your Kubernetes environment when needed.

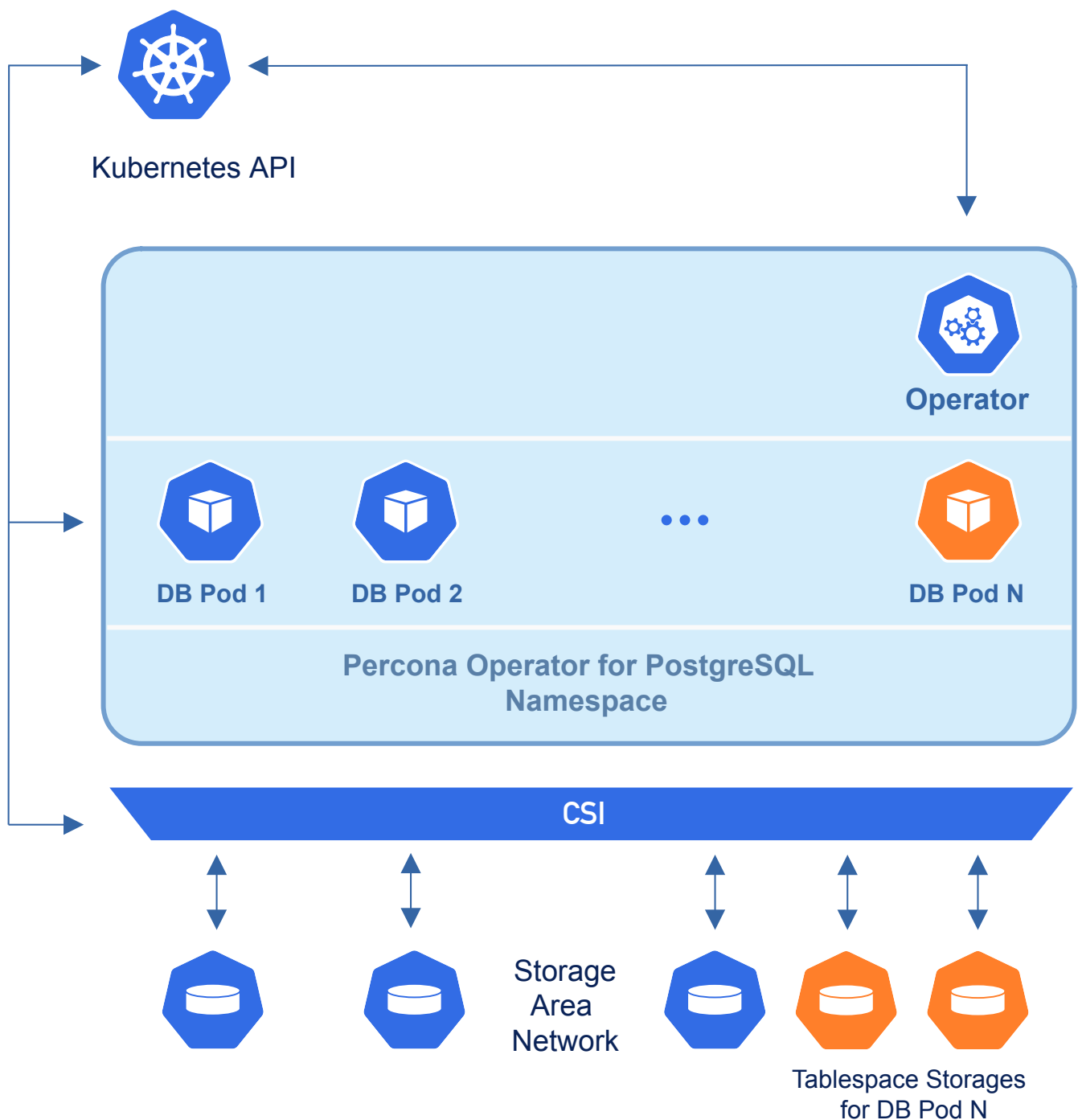
## Possible use cases

The most obvious use case for tablespaces is performance optimization. You place appropriate parts of the database on fast but expensive storage and engage slower but cheaper storage for lesser-used database objects. The classic example would be using an SSD for heavily-used indexes and using a large slow HDD for archive data.

Of course, the Operator [already provides](#) you with [traditional Kubernetes approaches](#) to achieve this on a per-Pod basis (Tolerations, etc.). But if you would like to go deeper and make such differentiation at the level of your database objects (tables and indexes), tablespaces are exactly what you would need for that.

Another well-known use case for tablespaces is quickly adding a new partition to the database cluster when you run out of space on the initially used one and cannot extend it (which may look less typical for cloud storage). Finally, you may need tablespaces when migrating your existing architecture to the cloud.

Each tablespace created by Percona Operator for PostgreSQL corresponds to a separate Persistent Volume, mounted in a container to the `/tablespaces` directory.



## Creating a new tablespace

To provide a new tablespace for your database in Kubernetes, you should do the following:

1. Configure the new tablespace storage with the Operator,
2. Create database objects in this tablespace with PostgreSQL.

### Configure the new tablespace storage with the Operator

Modify the Custom Resource and configure the [spec.instances.tablespaceVolumes](#) section for tablespaces.



1. Edit the `deploy/cr.yaml` configuration file. Specify the volume name, access mode and resource requests. Refer to [official Kubernetes documentation on Persistent Volumes](#) for details.

This example configuration shows how to create tablespace storage 1Gi in size:

```
spec:
 instances:
 ...
 tablespaceVolumes:
 - name: user
 dataVolumeClaimSpec:
 accessModes:
 - 'ReadWriteOnce'
 resources:
 requests:
 storage: 1Gi
```

2. Apply the configuration:

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



1. Create or update the existing `values.yaml` file and specify the tablespace storage configuration:
  - the name for the storage
  - access mode
  - resource requests

Refer to [official Kubernetes documentation on Persistent Volumes](#) for details.

This example configuration shows how to create tablespace storage 1Gb in size:

```
my-values.yaml
```

```
instances:
- name: instance1
 replicas: 3
 dataVolumeClaimSpec:
 accessModes:
 - ReadWriteOnce
 resources:
 requests:
 storage: 5Gi
 tablespaceVolumes:
 - name: user
 dataVolumeClaimSpec:
 accessModes:
 - 'ReadWriteOnce'
 resources:
 requests:
 storage: 1Gi
```

2. Update your release object for the database cluster:

```
helm upgrade -f my-values.yaml <release-name> percona/pg-db -n <namespace>
```

3. Check if your settings took effect:

```
helm get values <release-name> -n <namespace>
```

#### Expected output



After you apply the new configuration, the new `/tablespaces/user/` mountpoint will appear for your database. Please take into account that if you add your new tablespace to the already existing PostgreSQL cluster, it may take time for the Operator to create Persistent Volume Claims and get Persistent Volumes actually mounted.

Now you should actually create your tablespace on this volume with the `CREATE TABLESPACE <tablespace name> LOCATION <mount point>` command, and then create objects in it (of course, your user should have appropriate `CREATE` privileges to make it possible):

```
CREATE TABLESPACE user121
LOCATION '/tablespaces/user/data';
```

Now when the tablespace is created you can append `TABLESPACE <tablespace_name>` to your `CREATE` SQL statements to implicitly create tables, indexes, or even entire databases in specific tablespace.

Let's create an example table in the already mentioned `user121` tablespace:

```
CREATE TABLE products (
 product_sku character(10),
 quantity int,
 manufactured_date timestamptz)
TABLESPACE user121;
```



It is also possible to set a default tablespace with the `SET default_tablespace = <tablespace_name>;` statement. It will affect all further `CREATE TABLE` and `CREATE INDEX` commands without an explicit tablespace specifier, until you unset it with an empty string.

As you can see, Percona Operator for PostgreSQL simplifies tablespace creation by carrying on all necessary modifications with Persistent Volumes and Pods. The same would not be true for the deletion of an already existing tablespace, which is not automated, neither by the Operator nor by PostgreSQL.

## Deleting an existing tablespace

Deleting an existing tablespace from your database in Kubernetes also involves two parts:

- Delete related database objects and tablespace with PostgreSQL,
- Delete tablespace storage in Kubernetes.

To make tablespace deletion with PostgreSQL possible, you should make this tablespace empty (it is impossible to drop a tablespace until *all objects in all databases using this tablespace* have been removed). Tablespace are listed in the `pg_tablespace` table, and you can use it to find out which objects are stored in a specific tablespace. The example command for the `lake` tablespace will look as follows:

```
SELECT relname FROM pg_class WHERE reltablespace=(SELECT oid FROM pg_tablespace WHERE
spcname='user121');
```



When your tablespace is empty, you can log in to the *PostgreSQL Primary instance* as a *superuser*, and then execute the `DROP TABLESPACE <tablespace_name>;` command.

Now, when the PostgreSQL part is finished, you can remove the tablespace entry from the `tablespaceStorages` section (don't forget to run the `kubectl apply -f deploy/cr.yaml` command to apply changes).

# Monitor Kubernetes

Monitoring the state of the database is crucial to timely identify and react to performance issues. [Percona Monitoring and Management \(PMM\) solution enables you to do just that.](#)

However, the database state also depends on the state of the Kubernetes cluster itself. Hence it's important to have metrics that can depict the state of the Kubernetes cluster.

This document describes how to set up monitoring of the Kubernetes cluster health. This setup has been tested with the [PMM Server](#)  as the centralized data storage and the Victoria Metrics Kubernetes monitoring stack as the metrics collector. These steps may also apply if you use another Prometheus-compatible storage.

The Operator is compatible with both PMM versions 2 and 3. PMM2 has reached end-of-life and is deprecated. Therefore, we recommend using the latest PMM version 3 for optimal monitoring capabilities.

The steps in this tutorial are for PMM 3.

## Pre-requisites

To set up monitoring of Kubernetes, you need the following:

1. PMM Server up and running. You can run PMM Server as a Docker image, a virtual appliance, or on an AWS instance. Please refer to the [official PMM documentation](#)  for the installation instructions.
2. [Helm v3](#) .
3. [kubecttl](#) .
4. PMM 3 Service account token (or PMM2 API key).

## Configure authentication

## PMM3 (recommended)

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

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

### Warning

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

## PMM2 (deprecated)

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

### From PMM UI

[Generate the PMM API key](#) 

### From command line

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

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



### Warning

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

## Install the Victoria Metrics Kubernetes monitoring stack

# Install the Victoria Metrics Kubernetes monitoring stack

## Quick install

1. To install the Victoria Metrics Kubernetes monitoring stack with the default parameters, use the quick install command. Replace the following placeholders with your values:

- `PMM-SERVER-TOKEN` - The [PMM Server service account token](#)
- `PMM-SERVER-URL` - The URL to access the PMM Server
- `UNIQUE-K8s-CLUSTER-IDENTIFIER` - Identifier for the Kubernetes cluster. It can be the name you defined during the cluster creation.

You should use a unique identifier for each Kubernetes cluster. The use of the same identifier for more than one Kubernetes cluster will result in the conflicts during the metrics collection.

- `NAMESPACE` - The namespace where the Victoria metrics Kubernetes stack will be installed. If you haven't created the namespace before, it will be created during the command execution.

We recommend to use a separate namespace like `monitoring-system`.

```
curl -fsL https://raw.githubusercontent.com/Percona-Lab/k8s-monitoring/refs/tags/v0.1.2/vm-operator-k8s-stack/quick-install.sh | bash -s -- --api-key <PMM-SERVER-TOKEN> --pmm-server-url <PMM-SERVER-URL> --k8s-cluster-id <UNIQUE-K8s-CLUSTER-IDENTIFIER> --namespace <NAMESPACE>
```



### Note

The Prometheus node exporter is not installed by default since it requires privileged containers with the access to the host file system. If you need the metrics for Nodes, add the `--node-exporter-enabled` flag as follows:

```
curl -fsL https://raw.githubusercontent.com/Percona-Lab/k8s-monitoring/refs/tags/v0.1.2/vm-operator-k8s-stack/quick-install.sh | bash -s -- --api-key <PMM-SERVER-TOKEN> --pmm-server-url <PMM-SERVER-URL> --k8s-cluster-id <UNIQUE-K8s-CLUSTER-IDENTIFIER> --namespace <NAMESPACE> --node-exporter-enabled
```



## Install manually

You may need to customize the default parameters of the Victoria metrics Kubernetes stack.

- Since we use the PMM Server for monitoring, there is no need to store the data in Victoria Metrics Operator. Therefore, the Victoria Metrics Helm chart is installed with the `vm-single.enabled` and `vm-cluster.enabled` parameters set to `false` in this setup.
- [Check all the role-based access control \(RBAC\) rules](#)  of the `victoria-metrics-k8s-stack` chart and the dependencies chart, and modify them based on your requirements.

## Configure authentication in PMM

To access the PMM Server resources and perform actions on the server, configure authentication.

1. Encode the PMM Server token key with base64.



```
$ echo -n <API-key> | base64 --wrap=0
```



```
echo -n <API-key> | base64
```



2. Create the Namespace where you want to set up monitoring. The following command creates the Namespace `monitoring-system`. You can specify a different name. In the latter steps, specify your namespace instead of the `<namespace>` placeholder.

```
kubectl create namespace monitoring-system
```



3. Create the YAML file for the [Kubernetes Secrets](#)  and specify the base64-encoded API key value within. Let's name this file `pmm-api-vmoperator.yaml`.

**pmm-api-vmoperator.yaml**

```
apiVersion: v1
data:
 api_key: <base-64-encoded-pmm-server-token>
kind: Secret
metadata:
 name: pmm-token-vmoperator
 #namespace: default
type: Opaque
```



4. Create the Secrets object using the YAML file you created previously. Replace the `<filename>` placeholder with your value.

```
kubectl apply -f pmm-api-vmoperator.yaml -n <namespace>
```



5. Check that the secret is created. The following command checks the secret for the resource named `pmm-token-vmoperator` (as defined in the `metadata.name` option in the secrets file). If you defined another resource name, specify your value.

```
kubectl get secret pmm-token-vmoperator -n <namespace>
```



## Create a ConfigMap to mount for kube-state-metrics

The [kube-state-metrics \(KSM\)](#)  is a simple service that listens to the Kubernetes API server and generates metrics about the state of various objects - Pods, Deployments, Services and Custom Resources.

To define what metrics the `kube-state-metrics` should capture, create the [ConfigMap](#)  and mount it to a container.

Use the [example configmap.yaml configuration file](#)  to create the ConfigMap.

```
kubectl apply -f https://raw.githubusercontent.com/Percona-Lab/k8s-monitoring/refs/tags/v0.1.2/vm-operator-k8s-stack/ksm-configmap.yaml -n <namespace>
```

As a result, you have the `customresource-config-ksm` ConfigMap created.

## Install the Victoria Metrics Kubernetes monitoring stack

1. Add the dependency repositories of [victoria-metrics-k8s-stack](#)  chart.

```
helm repo add grafana https://grafana.github.io/helm-charts
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
```

2. Add the Victoria Metrics Kubernetes monitoring stack repository.

```
helm repo add vm https://victoriametrics.github.io/helm-charts/
```

3. Update the repositories.

```
helm repo update
```

4. Install the Victoria Metrics Kubernetes monitoring stack Helm chart. You need to specify the following configuration:

- the URL to access the PMM server in the `externalVM.write.url` option in the format `<PMM-SERVER-URL>/victoriametrics/api/v1/write`. The URL can contain either the IP address or the hostname of the PMM server.
- the unique name or an ID of the Kubernetes cluster in the `vmagent.spec.externalLabels.k8s_cluster_id` option. Ensure to set different values if you are sending metrics from multiple Kubernetes clusters to the same PMM Server.
- the `<namespace>` placeholder with your value. The Namespace must be the same as the Namespace for the Secret and ConfigMap

```
helm install vm-k8s vm/victoria-metrics-k8s-stack \
-f https://raw.githubusercontent.com/Percona-Lab/k8s-
monitoring/refs/tags/v0.1.2/vm-operator-k8s-stack/values.yaml \
--set externalVM.write.url=<PMM-SERVER-URL>/victoriametrics/api/v1/write \
--set vmagent.spec.externalLabels.k8s_cluster_id=<UNIQUE-CLUSTER-IDENTIFIER/NAME>
\
-n <namespace>
```



To illustrate, say your PMM Server URL is `https://pmm-example.com`, the cluster ID is `test-cluster` and the Namespace is `monitoring-system`. Then the command would look like this:

```
$ helm install vm-k8s vm/victoria-metrics-k8s-stack \
-f https://raw.githubusercontent.com/Percona-Lab/k8s-
monitoring/refs/tags/v0.1.2/vm-operator-k8s-stack/values.yaml \
--set externalVM.write.url=https://pmm-example.com/victoriametrics/api/v1/write \
--set vmagent.spec.externalLabels.k8s_cluster_id=test-cluster \
-n monitoring-system
```

## Validate the successful installation

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



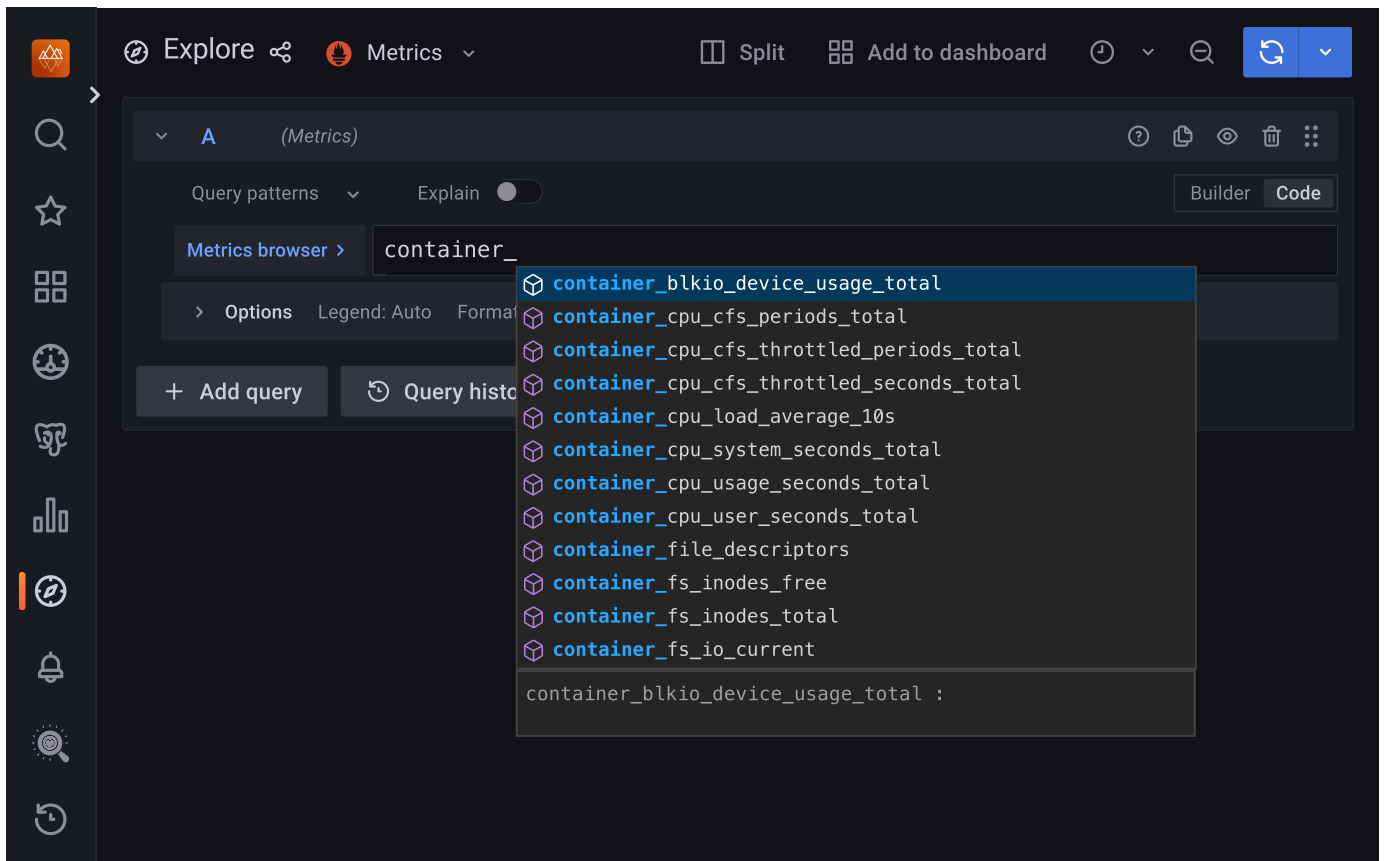
### Sample output



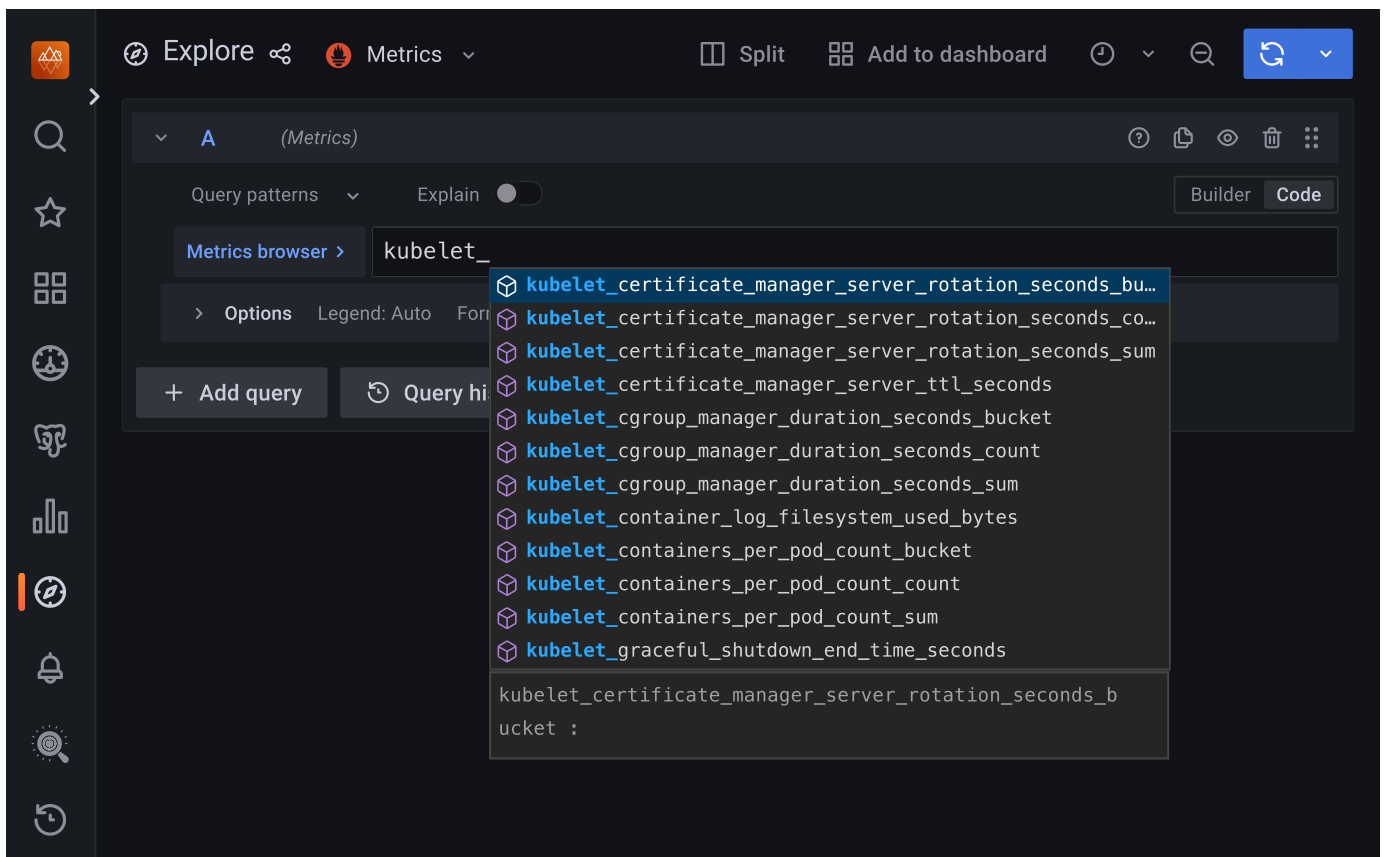
What Pods are running depends on the configuration chosen in values used while installing `victoria-metrics-k8s-stack` chart.

## Verify metrics capture

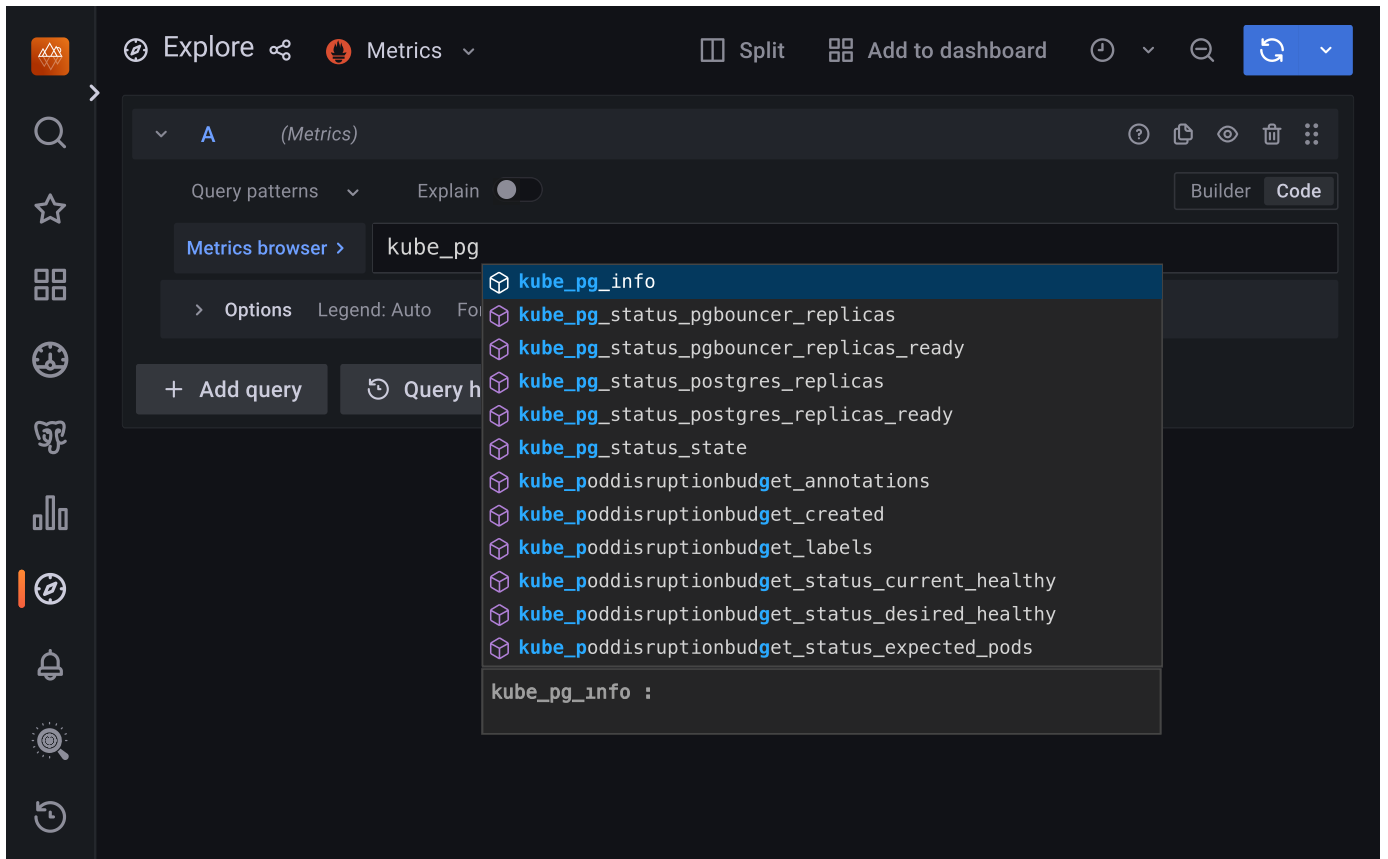
1. Connect to the PMM server.
2. Click **Explore** and switch to the **Code** mode.
3. Check that the required metrics are captured, type the following in the Metrics browser dropdown:
  - [cadvisor](#) .



- kubelet:



- [kube-state-metrics](#) metrics that also include Custom resource metrics for the Operator and database deployed in your Kubernetes cluster:



## Uninstall Victoria metrics Kubernetes stack

To remove Victoria metrics Kubernetes stack used for Kubernetes cluster monitoring, use the cleanup script. By default, the script removes all the [Custom Resource Definitions\(CRD\)](#) and Secrets associated with the Victoria metrics Kubernetes stack. To keep the CRDs, run the script with the `--keep-crd` flag.

## Remove CRDs

Replace the `<NAMESPACE>` placeholder with the namespace you specified during the Victoria metrics Kubernetes stack installation:

```
bash <(curl -fsL https://raw.githubusercontent.com/Percona-Lab/k8s-monitoring/refs/tags/v0.1.2/vm-operator-k8s-stack/cleanup.sh) --namespace <NAMESPACE>
```



## Keep CRDs

Replace the `<NAMESPACE>` placeholder with the namespace you specified during the Victoria metrics Kubernetes stack installation:

```
bash <(curl -fsL https://raw.githubusercontent.com/Percona-Lab/k8s-monitoring/refs/tags/v0.1.2/vm-operator-k8s-stack/cleanup.sh) --namespace <NAMESPACE> --keep-crd
```



Check that the Victoria metrics Kubernetes stack is deleted:

```
helm list -n <namespace>
```



The output should provide the empty list.

If you face any issues with the removal, uninstall the stack manually:

```
helm uninstall vm-k8s-stack -n < namespace>
```



# Use PostGIS extension with Percona Distribution for PostgreSQL

[PostGIS](#)  is a PostgreSQL extension that adds GIS capabilities to this database.

You can deploy and manage PostGIS-enabled PostgreSQL starting from the Operator version 2.3.0. Here's how to do it:

## Prepare your environment

1. Clone the `percona-postgresql-operator` repository because you will need to modify the cluster's Custom Resource. Specify your desired version for the `-b` flag:

```
git clone -b v3.0.0 https://github.com/percona/percona-postgresql-operator
cd percona-postgresql-operator
```



2. Create the namespace where the Operator and the cluster will run. Export it as an environment variable. Replace the `<namespace>` placeholder with your value:

```
kubectl create namespace <namespace>
export NAMESPACE=<namespace>
```



## Deploy Percona Operator for PostgreSQL

Create the Custom Resource Definition, role-based access control (RBAC) and the Operator deployment. To do that in one go, apply the `deploy/bundle.yaml` manifest:

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



## Deploy PostGIS-enabled database cluster

1. Modify the `deploy/cr.yaml` configuration file. Specify the image for PostGIS in the `spec.image` option. Use `docker.io/percona/percona-distribution-postgresql-with-postgis:18.4-2` instead of `docker.io/percona/percona-distribution-postgresql:18.4-1`

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGCluster
metadata:
 name: cluster1
spec:
 ...
 image: docker.io/percona/percona-distribution-postgresql-with-postgis:18.4-2
 ...
```

2. Create a cluster with the following command:

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

3. The creation process may take some time. Check the cluster status with the following command:

```
kubectl get pg -n $NAMESPACE
```

When the process is over your cluster will obtain the `ready` status.

 Expected output



## Enable PostGIS extension

To use the PostGIS extension, you should enable it for specific databases. To do that, your PostgreSQL user must be a superuser because only superusers can create extensions. By default, the Operator creates a standard database user without superuser privileges. You have several options to enable PostGIS in your databases:

1. **Connect as the `postgres` superuser to the primary Pod.** When you execute to the primary Pod and establish the `psql` session, you are connected as the `postgres` user. Then you can create or alter databases and enable the PostGIS extension as needed.
2. **Grant superuser privileges to the default user.** Connect to the database Pod as the `postgres` user and grant superuser privileges to the default Operator-created user. After this, you can log in with this user and enable the PostGIS extension for the databases this user has access to.
3. **Create a separate superuser.** You can create a dedicated superuser with access to the needed databases via the Custom Resource. Use this user to connect to PostgreSQL and enable the PostGIS extension for the databases this user has access to.

#### Note

By default, superusers cannot connect to PostgreSQL via `pgBouncer` due to security restrictions. To connect to PostgreSQL as the superuser you can:

- Update the `pgBouncer` configuration to allow superuser access (not recommended for general security reasons), or
- Connect to PostgreSQL primary Pod using the `<cluster-name>-primary` Kubernetes Service bypassing `pgBouncer`.

For details on managing users, roles, and configuring `pgBouncer` access, refer to the [Application and system users](#) documentation.

## Option 1. Connect to the primary Pod

1. Find the primary Pod:

```
kubectl get pods -n $NAMESPACE -l postgres-operator.crunchydata.com/role=primary 
```

#### Sample output

2. Exec into the primary Pod and establish the `psql` session:

```
kubectl -n $NAMESPACE exec -it <primary-pod-name> -- psql 
```

3. For example, you can create the new database named `mygisdata` as follows:

```
CREATE database mygisdata; 
```

4. Connect to this database and create a schema to separate your spacial data.

```
\c mygisdata;
CREATE SCHEMA gis; 
```

5. Enable the `postgis` extension for this database. Make sure you are connected to the database you created earlier and run the following command:

```
CREATE EXTENSION postgis; 
```

6. Check that the extension is enabled:

```
SELECT postgis_full_version(); 
```

The output should resemble the following:

## Expected output



## Option 2. Grant superuser privileges to default user

1. Connect to the primary Pod following the steps from [Option 1](#)
2. Grant your default user superuser privileges. The following command updates privileges for the `cluster1` user. Replace with your user name of needed:

```
ALTER USER cluster1 WITH SUPERUSER;
```



Exit the pod.

3. Retrieve the connection string URL from a Secret. List the Secrets:

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



Look for the Secret named after this pattern: `<cluster-name>-pguser-<username>`. For example, if your cluster name is `cluster1`, the Secret name is `cluster1-pguser-cluster1`.

4. The `data.uri` value in the Secret contains the connection string URL to the primary service. Export it as an environment variable:

```
URI=$(kubectl get secret `<cluster-name>-pguser-<username>` --namespace $NAMESPACE
-o jsonpath='{.data.uri}' | base64 --decode) | echo $URI
```



5. Create a Pod where you start Percona Distribution for PostgreSQL and connect to the database. The following command starts a Pod `pg-client` and connects it to the database using the connection string URL:

```
kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-
postgresql:18.4-1 --restart=Never -- psql $URI
```



## Sample output



6. When the Operator creates a user, it also creates the database and schema with the name that matches the username. You are now connected to this database. Create the `postgis` extension for it.

```
CREATE EXTENSION postgis;
```



7. Check that the extension is enabled:

```
SELECT postgis_full_version();
```



The output should resemble the following:

 Expected output



## Option 3. Create a dedicated superuser

1. Edit the Custom Resource and add a user that you will use to manage spatial data. For example, this user is called `gis`. List the databases this user has access to and grant it superuser privileges:

```
spec:
 users:
 - name: gis
 databases:
 - mygisdata
 - test
 options: "SUPERUSER"
```



2. Update the cluster configuration:

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



3. The Operator creates a user and a Secret object with user credentials named `<cluster-name>-pguser-<username>`. List the Secrets to verify it is created:

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



4. Retrieve the connection string URI to the primary Pod from the Secret and export it as an environment variable:

```
URI=$(kubectl get secret `<cluster-name>-pguser-<username>` --namespace $NAMESPACE \
-o jsonpath='{.data.uri}' | base64 --decode) | echo $URI
```



5. Create a Pod where you start Percona Distribution for PostgreSQL and use it to connect to the database. The following command starts a Pod `pg-client` and connects it to the database using the connection string URL:

```
kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-
postgresql:18.4-1 --restart=Never -- psql $URI
```





#### Sample output



6. When the Operator creates a user, it also creates the database and schema with the name that matches the username. You are now connected to this database. Create the `postgis` extension for it.

```
CREATE EXTENSION postgis;
```



7. Check that the extension is enabled:

```
SELECT postgis_full_version();
```



The output should resemble the following:



#### Expected output



You can find more about using PostGIS in the official Percona Distribution for PostgreSQL [documentation](#) , as well as in this [blogpost](#) .

# Configure DNS suffix for service discovery

In Kubernetes, services are assigned a DNS name to be accessible within the cluster. The domain name follows the pattern `<service-name>.<namespace>.svc.<cluster-domain>`. The default cluster domain is `cluster.local`, so a typical FQDN looks like `<service-name>.<namespace>.svc.cluster.local`.

When you refer to a service using only its short name, Kubernetes automatically expands it with this domain so the name resolves inside the cluster. This enables workloads to communicate without the need to specify fully-qualified domain names.

Clusters with custom DNS configurations may use a domain suffix different from the default `cluster.local`. This includes scenarios such as running inside a vcluster, which may respond to DNS queries with records that do not match the actual cluster domain.

The Operator tries to auto-detect the cluster's domain suffix by performing a CNAME lookup on the service `kubernetes.default.svc` (that is, a service named `kubernetes` in the `default` namespace). If the lookup returns a value, the Operator uses that as the domain suffix. If the Operator fails to auto-detect the DNS suffix, it falls back to using the default `cluster.local` domain. This may lead to services not resolving properly within the cluster.

To address this, you can set your custom DNS suffix for the Operator to use when it constructs service names. As a result, the Operator produces hostnames that match your cluster's DNS configuration, ensuring correct service resolution and discovery.

## How to configure

Add `clusterServiceDNSSuffix` under `spec` in your Custom Resource. Set it to your cluster's DNS suffix—for example, `cluster.local` for a standard cluster, or the host cluster's suffix when the Operator runs in a vcluster:

```
spec:
 ...
 clusterServiceDNSSuffix: cluster.local
 # ... rest of your spec
```



Apply the change:

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



# Delete Percona Operator for PostgreSQL

When cleaning up your Kubernetes environment (e.g., moving from a trial deployment to a production one, or testing experimental configurations), you may need to remove some (or all) of the following objects:

- Percona Distribution for PostgreSQL cluster managed by the Operator
- Percona Operator for PostgreSQL itself
- Custom Resource Definition deployed with the Operator
- Resources like PVCs and Secrets

## Delete a database cluster

You can delete the Percona Distribution for PostgreSQL cluster managed by the Operator by deleting the appropriate Custom Resource.

### Note

There are two [finalizers](#) defined in the Custom Resource, which define whether TLS-related objects and data volumes should be deleted or preserved when the cluster is deleted.

- `finalizers.percona.com/delete-ssl`: if present, deletes [objects, created for SSL](#) (Secret, certificate, and issuer) when the cluster deletion occurs.
- `finalizers.percona.com/delete-pvc`: if present, deletes [Persistent Volume Claims](#) for the database cluster Pods and user Secrets when the cluster deletion occurs.

Both finalizers are off by default in the `deploy/cr.yaml` configuration file, and this allows you to recreate the cluster without losing data, credentials for the system users, etc.

Here's a sequence of steps to follow:

- 1 List Custom Resources, replacing the `<namespace>` placeholder with your namespace.

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



### Sample output



- 2 Delete the Custom Resource with the name of your cluster (for example, let's use the default `cluster1` name).

```
kubectl delete pg cluster1 -n <namespace>
```





Sample output



- 3 Check that the cluster is deleted by listing the available Custom Resources once again.

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



Sample output



## Delete the Operator

You can uninstall the Operator by deleting the [Deployments](#)  related to it.

- 1 List the deployments. Replace the `<namespace>` placeholder with your namespace.

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



Sample output



- 2 Delete the `percona-*` deployment

```
kubectl delete deploy percona-postgresql-operator -n <namespace>
```



- 3 Check that the Operator is deleted by listing the Pods. As a result you should have no Pods related to it.

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



Sample output



## Delete Custom Resource Definition

If you are not just deleting the Operator and PostgreSQL cluster from a specific namespace, but want to clean up your entire Kubernetes environment, you can also delete the [CustomResourceDefinitions \(CRDs\)](#) .

### Warning

CRDs in Kubernetes are non-namespaced but are available to the whole environment. This means that you shouldn't delete CRD if you still have the Operator and database cluster in some namespace.

You can delete CRD as follows:

#### 1 List the CRDs:

```
kubectl get crd
```



 Sample output



#### 2 Now delete the `percona*.pgv2.percona.com` CRDs:

```
kubectl delete crd perconapgb backups.pgv2.percona.com
perconapgclusters.pgv2.percona.com perconapgstores.pgv2.percona.com
```



 Sample output



## Clean up resources

By default, TLS-related objects and data volumes remain in Kubernetes environment after you delete the cluster to allow you to recreate it without losing the data.

You can automate resource cleanup by turning on `percona.com/delete-pvc` and/or `percona.com/delete-ssl` [finalizers](#)). You can also delete TLS-related objects and PVCs manually.

To manually clean up resources, do the following:

#### 1 Delete Persistent Volume Claims.

##### 1 List PVCs. Replace the `<namespace>` placeholder with your namespace:

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



 Sample output



- 2 Delete PVCs related to your cluster. The following command deletes PVCs for the `cluster1` cluster:

```
kubectl delete pvc cluster1-instance1-mkwh-pgdata cluster1-instance1-nvh4-pgdata cluster1-instance1-qknb-pgdata cluster1-repo1 -n <namespace>
```



#### Sample output



Note that if your Custom Resource manifest includes the `percona.com/delete-pvc` finalizer, all user Secrets will be automatically deleted when you delete the PVCs. To prevent this from happening, disable the finalizer.

- 3 Delete the Secrets

- 1 List Secrets:

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



- 2 Delete the Secret:

```
kubectl delete secret <secret_name> -n <namespace>
```



# Retrieve Percona certified images

When preparing for the upgrade, you must have the list of compatible images for a specific Operator version and the database version you wish to update to. You can either manually find the images in the [list of certified images](#) or you can get this list by querying the **Version Service** server.

## What is the Version Service?

The **Version Service** is a centralized repository that the Percona Operator for PostgreSQL connects to at scheduled times to get the latest information on compatible versions and valid image paths. This service is a crucial part of the automatic upgrade process, and it is enabled by default. Its landing page, [check.percona.com](https://check.percona.com), provides more details about the service itself.

## How to query the Version Service

You can manually query the Version Service using the `curl` command. The basic syntax is:

```
curl https://check.percona.com/versions/v1/pg-operator/<operator-version>/<pg-version> | jq -r '.versions[].matrix'
```



where:

- **<operator-version>** is the version of the Percona Operator for PostgreSQL you are using.
- **<pg-version>** is the version of PostgreSQL you want to get images for. This part is optional and helps filter the results. It can be a specific PostgreSQL version (e.g.16.3), a recommended version (e.g. 16-recommended), or the latest available version (e.g. 16-latest).

For example, to retrieve the list of images for Operator version `2.4.0` for PostgreSQL version `16.3`, use the following command:

```
curl https://check.percona.com/versions/v1/pg-operator/2.4.0/16.3 | jq -r '.versions[].matrix'
```



### Sample output



To narrow down the results to the recommended version of PostgreSQL 16, you can use:

```
curl https://check.percona.com/versions/v1/pg-operator/2.4.0/16-recommended | jq -r '.versions[].matrix'
```



This command helps you retrieve the PostgreSQL images available for a specific Operator version (`2.4.0` in the following example):

```
curl -s https://check.percona.com/versions/v1/pg-operator/2.4.0 | jq -r
' .versions[0].matrix.postgresql | to_entries[] | "\(.key)\t\(.value.imagePath)\t\
(.value.status)" '
```



Sample output



# Troubleshooting

# Percona Operator troubleshooting

This section provides information on how to troubleshoot issues when you install Percona Operator for PostgreSQL.

Make sure you have CLI tool `kubectl` installed to interact with Kubernetes API.

## Check connection to Kubernetes cluster

It may happen that `kubectl` you installed locally is not connected to your Kubernetes cluster.

To check connectivity to your Kubernetes API, run the following command:

```
kubectl cluster-info
```



If you see the output similar to the following, it means that `kubectl` is connected to your Kubernetes cluster:

 Sample output



If multiple Kubernetes configurations are present in `kubeconfig`, check if you have set the correct context. If the context is wrong, switch it. Here's how:

1. Check the current context:

```
kubectl config current-context # Get the current Context
```



2. Switch the context :

```
kubectl config use-context <Context-To-Be-Used>
```



3. Run the `kubectl cluster-info` command again to verify that `kubectl` is connected to your Kubernetes cluster.

If you are still running into issues, check with your Kubernetes cluster administrator to resolve the connectivity or configuration issues.

## Troubleshoot Operator installation issues

1. Check the Operator logs

```
kubectl logs deploy/<operator-deployment-name>
```



2. Installing the Operator requires specific privileges, such as the ability to create custom resource definitions and other Kubernetes objects.

To verify that you have the necessary privileges, run the following script:

```
bash <(curl -s
https://gist.githubusercontent.com/cshiv/6048bdd0174275b48f633549c69d0844/raw/fd54
7b783a30b827362ee9f9ec03436f9bc79524/check_privileges.sh)
```



#### Sample output



If you have insufficient permissions, the script will show you which ones are missing for installing a particular Operator. In this case, contact the Kubernetes cluster administrator.

3. If you have the necessary privileges but the installation is still failing, review the Kubernetes Events for more details. Keep in mind that Kubernetes Events are retained for only 60 minutes.

```
kubectl get events --sort-by=".lastTimestamp"
```



Events provide good information about affinity issues, resource issues etc.

## Troubleshooting database cluster issues

1. The Operator deployment must be in the `Running` state for the database cluster to function properly. Check the Operator Pod for restarts to identify potential issues.

```
kubectl get pod <operator-pod-name>
```



2. Check the status of the database cluster

```
kubectl get pg <database-cluster-name>
```



The cluster should typically be in the `Running` state. It may briefly enter the `initializing` state while reconciling changes. If the cluster remains in the `initializing` state for an extended period, investigate further to identify any underlying issues.

Additionally, you can describe the database cluster and search for the information in the `State` and `State Description` fields:

```
kubectl describe pg <database-cluster-name>
```



### 3. Check the Operator logs

```
kubectl logs deploy/<operator-deployment-name>
```



### 4. Check the events

```
kubectl get events --sort-by=".lastTimestamp"
```



Events can provide information like storage class issues, PVC binding issues etc

### 5. Check for the PVC, PV. Both of them should be in `Bound` status

```
kubectl get pvc
```



```
kubectl get pv
```



### 6. Check for logs of database pods / Proxy pods

```
kubectl logs <database-pod-name>
```



```
kubectl logs <proxy-pod-name>
```



To check logs of `init` containers or other sidecar containers, use the option `-c` with the container name:

```
kubectl logs <proxy-pod-name> -c postgres-startup
```



### 7. Check for error details. Run the `kubectl describe` command:

```
kubectl describe <database-pod-name>
```



```
kubectl describe <proxy-pod-name>
```



Check the information in the `Status` section. The `State` and `State Description` fields explain why the Pod reports errors.

### 8. To run commands inside a container, use the `kubectl exec` command:

```
kubectl exec <pod-name> -- <command>
```



If you need an interactive shell to run multiple commands, use the `-it` flag for an interactive terminal:

```
kubectl exec -it <pod-name> -- sh
```



9. If the pods are not running, it may not be possible to execute commands or open an interactive shell. In such cases, consider using a `sleep-forever` script to prevent the containers from restarting repeatedly.

See the [Disable health check probes for maintenance](#) section for steps.

## Profiling the Operator with pprof

When you need to investigate Operator performance issues, high CPU usage, or memory leaks, you can collect CPU and memory profiles using Go's pprof tooling.

1. Enable pprof by setting the `PPROF_BIND_ADDRESS` environment variable in the Operator Deployment. For example, set it to `127.0.0.1:6060` to bind pprof to localhost on port 6060. See [PPROF\\_BIND\\_ADDRESS](#) for configuration details.
2. Restart the Operator so the new configuration takes effect.
3. Set up a port-forward to the Operator Pod:

```
kubectl port-forward deploy/percona-postgresql-operator 6060:6060 -n <operator-namespace>
```

4. In another terminal, collect profiles using `go tool pprof`:

**CPU profile** (30 seconds):

```
go tool pprof http://localhost:6060/debug/pprof/profile?seconds=30
```

**Heap (memory) profile:**

```
go tool pprof http://localhost:6060/debug/pprof/heap
```

**Goroutine profile** (useful for deadlock investigation):

```
go tool pprof http://localhost:6060/debug/pprof/goroutine
```

5. After collecting a profile, you can analyze it interactively. For example, type `top` to see the functions consuming the most CPU or memory, or `web` to generate a flame graph (requires Graphviz).



### Note

Disable pprof when you finish troubleshooting by setting `PPROF_BIND_ADDRESS` to `"0"` or `" "`. Keeping it enabled in production is not recommended for security reasons.

# Initial troubleshooting

Percona Operator for PostgreSQL uses [Custom Resources](#) to manage options for the various components of the cluster.

- `PerconaPGCluster` Custom Resource with Percona PostgreSQL Cluster options (it has handy `pg` shortname also),
- `PerconaPGBackup` and `PerconaPGRestore` Custom Resources contain options for `pgBackRest` used to backup PostgreSQL Cluster and to restore it from backups (`pg-backup` and `pg-restore` shortnames are available for them).

The first thing you can check for the Custom Resource is to query it with `kubectl get` command:

```
kubectl get pg
```



Expected output



The Custom Resource should have `Ready` status.

## Note

You can check which Percona's Custom Resources are present and get some information about them as follows:

```
kubectl api-resources | grep -i percona
```



Expected output



## Check the Pods

If Custom Resource is not getting `Ready` status, it makes sense to check individual Pods. You can do it as follows:

```
kubectl get pods
```





#### Expected output



NAME	READY	STATUS	RESTARTS	AGE
cluster1-backup-4vwt-p5d9j	0/1	Completed	0	97m
cluster1-instance1-b5mr-0	4/4	Running	0	99m
cluster1-instance1-b8p7-0	4/4	Running	0	99m
cluster1-instance1-w7q2-0	4/4	Running	0	99m
cluster1-pgbouncer-79bbf55c45-62x1k	2/2	Running	0	99m
cluster1-pgbouncer-79bbf55c45-9g4cb	2/2	Running	0	99m
cluster1-pgbouncer-79bbf55c45-9nrmd	2/2	Running	0	99m
cluster1-repo-host-0	2/2	Running	0	99m
percona-postgresql-operator-79cd8586f5-2qzcs	1/1	Running	0	120m

The above command provides the following insights:

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

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

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



#### Expected output



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

# Check Storage-related objects

Storage-related objects worth to check in case of problems are the following ones:

- [Persistent Volume Claims \(PVC\) and Persistent Volumes \(PV\)](#), which are playing a key role in stateful applications.
- [Storage Class](#), which automates the creation of Persistent Volumes and the underlying storage.

It is important to remember that PVC is namespace-scoped, but PV and Storage Class are cluster-scoped.

## Check the PVC

You can check all the PVC with the following command (use different namespace name instead of `postgres-operator`, if needed):

```
kubectl get pvc -n postgres-operator
```

Expected output

NAME	STATUS	VOLUME	CAPACITY
ACCESS MODES      STORAGECLASS      AGE			
cluster1-instance1-4xkv-pgdata RWO                  standard                  11h	Bound	pvc-2d20abb7-5350-4810-a098-fbdfbffa041	1Gi
cluster1-instance1-njt9-pgdata RWO                  standard                  11h	Bound	pvc-f2e9a722-fd30-435b-ade4-9edf20b2104b	1Gi
cluster1-instance1-qhh6-pgdata RWO                  standard                  11h	Bound	pvc-7228300b-81de-4a60-a615-8ca935c95139	1Gi
cluster1-repo1 RWO                  standard                  11h	Bound	pvc-b2e0bac3-993d-499e-b311-3aa7b9851bc2	1Gi

- **STATUS:** shows the [state](#) of the PVC:
  - For normal working of an application, the status should be `Bound`.
  - If the status is not `Bound`, further investigation is required.
- **VOLUME:** is the name of the Persistent Volume with which PVC is Bound to. Obviously, this field will be occupied only when a PVC is Bound.
- **CAPACITY:** it is the size of the volume claimed.
- **STORAGECLASS:** it indicates the [Kubernetes storage class](#) used for dynamic provisioning of Volume.
- **ACCESS MODES:** [Access mode](#) indicates how Volume is used with the Pods. Access modes should have write permission if the application needs to write data, which is obviously true in case of databases.

Now you can check a specific PVC for more details using its name as follows:

```
kubectl get pvc cluster1-instance1-4xkv-pgdata -n postgres-operator -oyaml # output
```

stripped for brevity, name of PVC may vary

#### Expected output

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 ...
 name: cluster1-instance1-4xkv-pgdata
 namespace: postgres-operator
 ...
spec:
 accessModes:
 - ReadWriteOnce
 resources:
 requests:
 storage: 1G
 storageClassName: standard
 volumeMode: Filesystem
 volumeName: pvc-2d20abb7-5350-4810-a098-fbdfbffd041
status:
 accessModes:
 - ReadWriteOnce
 capacity:
 storage: 24Gi
 phase: Bound
```

You can use a number of Custom Resource options to tweaking PVC for the components of your cluster:

- options under `instances.walVolumeClaimSpec` allow you to set [access modes](#) and [requested storage size](#) for PostgreSQL Write-ahead Log storage,
- options under `instances.dataVolumeClaimSpec` allow you to set [access modes](#) and also [requests](#) and [limits](#) for PostgreSQL database storage,
- options under `instances.tablespaceVolumes.dataVolumeClaimSpec` allow you to set [access modes](#) and [requested storage size](#) for PostgreSQL [tablespace](#) volumes,
- options under `backups.pgbackrest.repos.volume.volumeClaimSpec` allow you to set [access modes](#) and [requested storage size](#) for the pgBackRest storage.

## Check the PV

It is important to remember that PV is a cluster-scoped Object. If you see any issues with attaching a Volume to a Pod, PV and PVC might be looked upon.

Check all the PV present in the Kubernetes cluster as follows:

```
kubectl get pv
```



## Expected output



NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS
CLAIM		STORAGECLASS	REASON AGE	
pvc-2d20abb7-5350-4810-a098-fbdfbffa041	1Gi	RWO	Delete	Bound
postgres-operator/cluster1-instance1-4xkv-pgdata		standard	11h	
pvc-7228300b-81de-4a60-a615-8ca935c95139	1Gi	RWO	Delete	Bound
postgres-operator/cluster1-instance1-qhh6-pgdata		standard	11h	
pvc-b2e0bac3-993d-499e-b311-3aa7b9851bc2	1Gi	RWO	Delete	Bound
postgres-operator/cluster1-repo1		standard	11h	
pvc-f2e9a722-fd30-435b-ade4-9edf20b2104b	1Gi	RWO	Delete	Bound
postgres-operator/cluster1-instance1-njt9-pgdata		standard	11h	

Now you can check a specific PV for more details using its name as follows:

```
kubect1 get pv pvc-2d20abb7-5350-4810-a098-fbdfbffa041 -oyaml
```





## Expected output



```
apiVersion: v1
kind: PersistentVolume
metadata:
 ...
 name: pvc-f3e7097f-accd-4f5d-9c9d-6f29b54a368b
 ...
spec:
 accessModes:
 - ReadWriteOnce
 capacity:
 storage: 1Gi
 claimRef:
 apiVersion: v1
 kind: PersistentVolumeClaim
 name: cluster1-instance1-4xkv-pgdata
 namespace: postgres-operator
 resourceVersion: "912868"
 uid: f3e7097f-accd-4f5d-9c9d-6f29b54a368b
 gcePersistentDisk:
 fsType: ext4
 pdName: pvc-f3e7097f-accd-4f5d-9c9d-6f29b54a368b
 nodeAffinity:
 required:
 nodeSelectorTerms:
 - matchExpressions:
 - key: topology.kubernetes.io/zone
 operator: In
 values:
 - us-central1-a
 - key: topology.kubernetes.io/region
 operator: In
 values:
 - us-central1
 persistentVolumeReclaimPolicy: Delete
 storageClassName: standard
 volumeMode: Filesystem
status:
 phase: Bound
```

Fields to check if there are any issues in binding with PVC, are the `claimRef` and `nodeAffinity`.

The `claimRef` one indicates to which PVC this volume is bound to. This means that if by any chance PVC is deleted (e.g. by the appropriate finalizer), this section needs to be modified so that it can bind to a new PVC.

The `spec.nodeAffinity` field may influence the PV availability as well: for example, it can make Volume accessed in one availability zone only.

## Check the StorageClass

StorageClass is also a cluster-scoped object, and it indicates what type of underlying storage is used for the Volumes.

You can set StorageClass in `instances.dataVolumeClaimSpec.storageClassName`, `instances.walVolumeClaimSpec.storageClassName`, and `backups.pgbackrest.repos.volume.volumeClaimSpec.storageClassName` Custom Resource options.

The following command checks all the storage class present in the Kubernetes cluster, and allows to see which storage class is the default one:

```
kubectl get sc
```



#### Expected output



NAME	PROVISIONER	RECLAIMPOLICY	VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION	AGE		
premium-rwo	pd.csi.storage.gke.io	Delete	WaitForFirstConsumer true
44d			
standard (default)	kubernetes.io/gce-pd	Delete	Immediate true
44d			
standard-rwo	pd.csi.storage.gke.io	Delete	WaitForFirstConsumer true
44d			

If some PVC does not refer any storage class explicitly, it means that the default storage class is used. Ensure there is only one default Storage class.

You can check a specific storage class as follows:

```
kubectl get sc standard -oyaml
```



#### Expected output



```
allowVolumeExpansion: true
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
 annotations:
 storageclass.kubernetes.io/is-default-class: "true"
 creationTimestamp: "2022-10-09T06:28:03Z"
 labels:
 addonmanager.kubernetes.io/mode: EnsureExists
 name: standard
 resourceVersion: "906"
 uid: 933d37db-990b-4b2d-bf3a-dd091d0b00ae
parameters:
 type: pd-standard
provisioner: kubernetes.io/gce-pd
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

Important things to observe here are the following ones:

- Check if the provisioner and parameters are indicating the type of storage you intend to provision.
- Check the [volumeBindingMode](#)  especially if the storage cannot be accessed across availability zones. “WaitForFirstConsumer” volumeBindingMode ensures volume is provisioned only after a Pod requesting the Volume is created.
- If you are going to rely on the Operator [storage scaling functionality](#), ensure the storage class supports PVC expansion (it should have `allowVolumeExpansion: true` in the output of the above command).

You can set PVC storage class with the following Custom Resource options:

- `instances.walVolumeClaimSpec.storageClassName` allows you to set storage class for PostgreSQL Write-ahead Log storage,
- `instances.dataVolumeClaimSpec.storageClassName` allows you to set storage class for PostgreSQL database storage,
- `instances.tablespaceVolumes.dataVolumeClaimSpec.storageClassName` allows you to set storage class for PostgreSQL [tablespace](#) volumes,
- `backups.pgbackrest.repos.volume.volumeClaimSpec.storageClassName` allows you to set storage class for the pgBackRest storage.

# Exec into the containers

If you want to examine the contents of a container “in place” using remote access to it, you can use the `kubectl exec` command. It allows you to run any command or just open an interactive shell session in the container. Of course, you can have shell access to the container only if container supports it and has a “Running” state.

In the following examples we will access the container `database` of the `cluster1-instance1-b5mr-0` Pod.

- Run `date` command:

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



## Expected output



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

```
OCI runtime exec failed: exec failed: unable to start container process: exec:
"time": executable file not found in $PATH: unknown command terminated with exit
code 126
```

- Print log files to a terminal:

```
kubectl exec -ti cluster1-instance1-b5mr-0 -c database -- cat
/pgdata/pg16/log/postgresql-*.log
```



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

```
kubectl exec -ti cluster1-instance1-b5mr-0 -c database -- bash
bash-4.4$ hostname
cluster1-pxc-0
bash-4.4$ ls /pgdata/pg16/log/
postgresql-Wed.log
bash-4.4$ exit
exit
$
```



# Check the logs

Logs provide valuable information. It makes sense to check the logs of the database Pods and the Operator Pod. Following flags are helpful for checking the logs with the `kubectl logs` command:

Flag	Description
<code>-c, --container=&lt;container-name&gt;</code>	Print log of a specific container in case of multiple containers in a Pod
<code>-f, --follow</code>	Follows the logs for a live output
<code>--since=&lt;time&gt;</code>	Print logs newer than the specified time, for example: <code>--since="10s"</code>
<code>--timestamps</code>	Print timestamp in the logs (timezone is taken from the container)
<code>-p, --previous</code>	Print previous instantiation of a container. This is extremely useful in case of container restart, where there is a need to check the logs on why the container restarted. Logs of previous instantiation might not be available in all the cases.

In the following examples we will access containers of the `cluster1-instance1-b5mr-0` Pod.

- Check logs of the `database` container:

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



- Check logs of the `pgbackrest` container:

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



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

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



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

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



# Increase pgBackRest log verbosity

The pgBackRest tool used for backups [supports different log verbosity levels](#) . By default, it logs warnings and errors, but sometimes fixing backup/restore issues can be simpler when you get more debugging information from it.

Log verbosity is controlled by pgBackRest [-log-level-stderr](#)  option.

You can add it to the `deploy/backup.yaml` file to use it with [on-demand backups](#) as follows:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGBBackup
metadata:
 name: backup1
spec:
 pgCluster: cluster1
 repoName: repo1
 options:
 - --log-level-stderr=debug
```



# Manual management of database clusters deployed with Percona Operator for PostgreSQL

The purpose of the Operator is to automate database management tasks for you. However, you may need to manage the database cluster manually. For example, to troubleshoot issues or for maintenance.

The following sections explain how you can manage your cluster manually.

## Disable health check probes for maintenance

Probes are tasks Kubernetes runs to gather information about the health and status of containers running within Pods. They serve as a mechanism to ensure the system is running smoothly by periodically checking the state of applications and services.

Kubernetes has various types of probes:

- Startup probe verifies whether the application within a container is started
- Liveness probe determines when to restart a Pod
- Readiness probe checks that the container is ready to start accepting traffic

Sometimes it's necessary to take a manual control over the `postgres` process for maintenance. This means you need to disable a Kubernetes liveness probe so that it doesn't restart the database container during the maintenance period.

Here's what you need to do:

1. Create a `sleep-forever` file in the data directory with the following command:

```
kubectl exec cluster1-instance1-24b8-0 -- touch /pgdata/sleep-forever
```



2. Delete the Pod:

```
kubectl delete pod cluster1-instance1-24b8-0
```



3. After the Pod restarts, it won't start PostgreSQL. You can check it with the following command:

```
kubectl logs cluster1-instance1-24b8-0 database
```



Expected output



4. Now you can start PostgreSQL manually:

```
kubectl exec cluster1-instance1-24b8-0 -- pg_ctl -D /pgdata/pg17 start
```



 Expected output



- When you are done with the maintenance, remove the `sleep-forever` file to re-enable the liveness probe.

```
kubectl exec cluster1-instance1-24b8-0 -- rm /pgdata/sleep-forever
```



## Stop reconciliation by putting a cluster into an unmanaged mode

The Operator reconciles the database cluster to ensure its current state doesn't differ from the state defined in the configuration. It can automatically install, update, or repair the cluster when needed.

By doing this, the Operator might interfere with your operations during the maintenance. Therefore, you can put a cluster in an unmanaged mode to stop the Operator from reconciling the cluster at all.

Edit the `deploy/cr.yaml` Custom Resource manifest and set the `spec.unmanaged` option to `true`:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGCluster
metadata:
 name: cluster1
spec:
 unmanaged: true
 ...
```



Apply the changes:

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



### Warning

Putting a cluster in an unmanaged mode doesn't disable any of the health check probes already configured for containers. The Operator is only responsible for configuring the probes, not for running them. Refer to the [Disabling health check probes for maintenance](#) section for the steps.

## Override Patroni configuration

## For a whole cluster

The Operator creates a ConfigMap called `<cluster-name>-config` to store a Patroni cluster configuration. If you just edit the ConfigMap contents, the Operator will immediately rewrite and remove your changes. To override anything in this ConfigMap and keep the changes, you need to annotate it using a special annotation `pgv2.percona.com/override-config`.

Here is the example command for the cluster named `cluster1`:

```
kubectl annotate cm cluster1-config pgv2.percona.com/override-config=true
```



### Expected output



As long as the ConfigMap has this `pgv2.percona.com/override-config` annotation, the Operator doesn't rewrite your changes. You can edit the ConfigMap's contents however you want.

### Warning

The Operator does not validate your configuration changes.

Before applying any changes, consult the [Patroni documentation](#)  to ensure your configuration is correct. This will help you avoid issues caused by invalid settings.

It takes some time for your changes of ConfigMap to propagate to running containers. You can verify if changes are propagated by checking the mounted file in containers. For example:

```
kubectl exec -it cluster1-instance1-24b8-0 -- cat /etc/patroni/~postgres-operator_cluster.yaml
```



Operator doesn't apply a new configuration for Patroni automatically. You must run `patronictl reload <cluster_name> <pod-name>` to apply it after your changes are propagated to the container.

### Warning

Don't forget to remove this annotation once you've finished. It's not recommended to use this feature to permanently override Patroni configuration. As long as this annotation exists, the Operator won't touch the ConfigMap and you might have problems with your cluster.

To remove the annotation, use the following command:

```
kubectl annotate cm cluster1-config pgv2.percona.com/override-config-
```



## For an individual Pod

Operator creates a ConfigMap called `<pod-name>-config` to store Patroni instance configuration for each Pod. If you just edit the ConfigMap contents, the Operator will immediately rewrite and remove your changes. To override anything in these ConfigMaps and keep the changes, you need to annotate them using a special annotation:

```
kubectl annotate cm cluster1-instance1-24b8-config pgv2.percona.com/override-config=true
```



#### Expected output



As long as the ConfigMap has the `pgv2.percona.com/override-config` annotation, the Operator doesn't rewrite your changes. You can edit the ConfigMap's contents however you want.



#### Warning

The Operator does not validate your configuration changes.

Before applying any changes, consult the [Patroni documentation](#) to ensure your configuration is correct. This will help you avoid problems caused by invalid settings.

It takes some time for your changes of ConfigMap to propagate to running containers. You can verify if changes are propagated by checking the mounted file in containers for a Pod. For example:

```
kubectl exec -it cluster1-instance1-24b8-0 -- cat /etc/patroni/~postgres-operator_cluster.yaml
```



Operator doesn't apply a new configuration automatically. You must run `patronictl reload <cluster_name> <pod_name>` to apply it after your changes are propagated to the container.

To find the cluster name, run:

```
kubectl exec -it cluster1-instance1-24b8-0 -- patronictl list
```



#### Expected output



### Warning

Don't forget to remove this annotation once you've finished. It's not recommended to use this feature to permanently override Patroni configuration. As long as this annotation exists, the Operator won't touch the ConfigMap and you might have problems with your cluster.

To remove the annotation, use the following command:

```
kubectl annotate cm cluster1-instance1-24b8-0 pgv2.percona.com/override-config-
```



## Override PostgreSQL parameters

Use the `patronictl show-config` command to print PostgreSQL parameters used in the cluster. For example:

```
kubectl exec cluster1-instance1-24b8-0 -- patronictl show-config
```



### Expected output



Use the `patronictl edit-config` command to change any PostgreSQL parameter.

For example, run the following command to change the `restore_command` parameter:

```
kubectl exec -it cluster1-instance1-24b8-0 -- patronictl edit-config --pg
restore_command=/bin/true
```



### Expected output



This command changes the `shared_preload_libraries` parameter:

```
kubectl exec -it cluster1-instance1-24b8-0 -- patronictl edit-config --pg
shared_preload_libraries=""
```



### Expected output



### Warning

If you update any object controlled by the Operator, it'll reconcile the cluster and your configuration changes will be reverted. You can [put the cluster in an unmanaged mode](#) to prevent this.

## Override `pg_hba` entries

You may want to append entries to `pg_hba`. You can use the `spec.patroni.postgresql.pg_hba` field to add your rules.

```
patroni:
 dynamicConfiguration:
 postgresql:
 pg_hba:
 - local all all trust
 - reject all all all
```



The order of parameters matters in `pg_hba.conf`, so consider overriding the list completely. For this, you can use the `patronictl edit-config` command:

```
kubectl exec -it cluster1-instance1-24b8-0 -- patronictl edit-config --set
postgresql.pg_hba='[
 "local all all trust",
 "reject all all all"
]'
```



### Warning

If you update any object controlled by the Operator, it'll reconcile the cluster and your configuration changes will be reverted. You can [put the cluster in an unmanaged mode](#) to prevent this.

# Reinitialize replicas

When you create a new Percona PostgreSQL cluster, the Operator uses the `basebackup` method to create replicas for it. After the database instances are ready, the Operator automatically creates a full backup. Once this backup finishes successfully, the Operator updates the Patroni configuration and **prepends** (puts as the first method) `pgBackRest` in the `create_replica_methods` list so that new replicas are created using it.

## Warning

The Operator doesn't run `patronictl reload` in old replicas even if Patroni instance configurations are updated to put `pgBackRest` as the first method in the `create_replica_methods` list. For this configuration to run into force, you need to either restart the Pods or manually run `patronictl reload <cluster_name>` on all old replicas.

You may need to reinitialize cluster replicas. For example, if the data on the replica becomes corrupted or inconsistent with the primary node. Reinitialization ensures the replica is rebuilt with the correct data. Or, if the replica falls significantly behind the primary or encounters issues that prevent successful synchronization, reinitialization can reset the replica to match the current state of the primary.

You can do this:

- Manually by operating the database cluster. This is the recommended approach as you have the full control over the data state in your database.
- Automatically via Patroni. When Patroni notices the timelines between the primary and a replica diverged and the replica cannot stream from a primary, it automatically removes the data directory and recreates the replica to ensure it rejoins the cluster. Note that in this flow the Operator cannot ensure all transactions are replicated somewhere and it might potentially result in data loss.

This document provides the steps how to do it both ways.

## Reinitialize by deleting replica Pod and its PersistentVolumeClaim

You can force reinitialization by deleting the Pod and its PersistentVolumeClaim:

```
kubectl delete pvc/cluster1-instance1-24b8-pgdata pod/cluster1-instance1-24b8-0
```



## Expected output



The Operator will reinitialize a replica using the method configured in this instance's Patroni configuration. This configuration is stored within the ConfigMap for the instance. Use the following command to find it:

```
kubectl get cm cluster1-instance1-24b8-config
```



 Expected output



## Reinitialize with `patronictl reinit`

You can reinitialize a replica using the `patronictl reinit` command. Note that configuration in ConfigMap might not have been applied to a running Patroni instance. The recommended approach is to first run `patronictl reload <cluster_name>` and then run `patronictl reinit`.

For example:

1. List and verify Patroni configuration:

```
kubectl exec -it cluster1-instance1-24b8-0 -- cat /etc/patroni/~postgres-operator_instance.yaml
```



2. Find the cluster name:

```
kubectl exec -it cluster1-instance1-24b8-0 -- patronictl list
```



 Expected output



3. Reload the configuration:

```
kubectl exec -it cluster1-instance1-24b8-0 -- patronictl reload cluster1-ha cluster1-instance1-24b8-0
```



 Expected output



4. Reinitialize the replica:

```
kubectl exec -it cluster1-instance1-24b8-0 -- patronictl reinit cluster1-ha cluster1-instance1-24b8-0
```



 Expected output



```
+ Cluster: cluster1-ha (7487948770079264836) -----+-----+
-----+-----+
| Member | Host | Role |
State | TL | Lag in MB |
+-----+-----+-----+-----+-----+-----+
| cluster1-instance1-24b8-0 | cluster1-instance1-24b8-0.cluster1-pods | Replica |
streaming | 1 | 0 |
| cluster1-instance1-84xm-0 | cluster1-instance1-84xm-0.cluster1-pods | Leader |
running | 1 | |
| cluster1-instance1-nv28-0 | cluster1-instance1-nv28-0.cluster1-pods | Replica |
streaming | 1 | 0 |
+-----+-----+-----+-----+-----+-----+
-----+-----+
Are you sure you want to reinitialize members cluster1-instance1-24b8-0? [y/N]: y
Success: reinitialize for member cluster1-instance1-24b8-0
```

## Configure `create_replica_methods`

The Operator uses `basebackup` and `pgBackRest` methods to create replicas by default. These methods are defined within the `create_replica_methods` configuration block of a Patroni instance.

If you want to change `create_replica_methods` list for any reason, you can use the `spec.patroni.create_replica_methods` option in the `deploy/cr.yaml` Custom Resource manifest:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGCluster
metadata:
 name: cluster1
spec:
 patroni:
 createReplicaMethods:
 - basebackup
 - pgbackrest
 ...
```

Apply this configuration:

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

The Operator updates Patroni instances' ConfigMaps. You can check their configuration with this command:

```
kubectl get configmap cluster1-instance1-24b8-config -o yaml
```

 Expected output



After the ConfigMap is updated, it takes some time for changes to appear in mounted files in containers. You can verify the updates by manually checking the file:

```
kubectl exec -it cluster1-instance1-24b8-0 -- cat /etc/patroni/~postgres-operator_instance.yaml
```



#### Expected output



Though the Operator updates the ConfigMaps, it doesn't automatically apply the new configuration for Patroni. To make Patroni aware of the changes, reload its configuration on every instance with the `patronictl reload <cluster_name> <pod-name>` command.

## Automate replica reinitialization with Patroni

You can instruct Patroni to reinitialize the replica when it detects that the replica's timeline diverges from the primary one. Update the Custom Resource and set the

`.spec.patroni.removeDataDirectoryOnDivergedTimelines` option:

```
spec:
 patroni:
 removeDataDirectoryOnDivergedTimelines: true
```



Apply the configuration for the changes to come into force:

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



#### Warning

The `removeDataDirectoryOnDivergedTimelines` option can lead to data loss. When the Operator resyncs the replica automatically, some transactions may be lost. The risk is usually small but not zero. Use this option only if you understand and accept this trade-off.

# Reference

# Custom Resource options

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



**Note**

Some options cannot be changed after creation or have specific modification limits. See [Options with modification limits](#) for details.

## metadata

The metadata part of this file contains the following keys:

- `name` (`cluster1` by default) sets the name of your Percona Distribution for PostgreSQL Cluster; it should include only [URL-compatible characters](#) , not exceed 22 characters, start with an alphabetic character, and end with an alphanumeric character;
- `annotations.pgvg2.percona.com/custom-patroni-version` [Kubernetes annotation](#)  which allows turning off automatic Patroni version detection by the Operator. You can use this annotation to set the version manually ("3" for Patroni 3.x, "4" for Patroni 4.x).
- `finalizers.percona.com/delete-ssl` if present, activates the [Finalizer](#)  which deletes [objects, created for SSL](#) (Secret, certificate, and issuer) after the cluster deletion event (off by default).
- `finalizers.percona.com/delete-pvc` if present, activates the [Finalizer](#)  which deletes [Persistent Volume Claims](#)  for the database cluster Pods and user Secrets after the deletion event (off by default).
- `finalizers.percona.com/delete-backups` if present, activates the [Finalizer](#)  which deletes all the [backups](#) of the database cluster from all configured repos on cluster deletion event (off by default). **`delete-backups` finalizer is in tech preview state, and it is not yet recommended for production environments.**

## Top level spec elements

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

### crVersion

Version of the Operator the Custom Resource belongs to.

Value type	Example
 string	<code>3.0.0</code>

## clusterServiceDNSSuffix

A custom cluster domain to be used as a DNS suffix used when constructing internal service names. Use this when the Operator runs in a vcluster or a cluster with a custom DNS domain so it can correctly resolve services. See [Configure DNS suffix for service discovery](#) for details.

Value type	Example
 string	<code>cluster.local</code>

## metadata.annotations

The [Kubernetes annotations](#)  metadata to be set at a global level for all resources created by the Operator.

Value type	Example
 label	<code>example-annotation: value</code>

## metadata.labels

The [Kubernetes labels](#)  metadata to be set at a global level for all resources created by the Operator.

Value type	Example
 label	<code>example-label: value</code>

## tlsOnly

Enforce the Operator to use only Transport Layer Security (TLS) for both internal and external communications.

Value type	Example
 boolean	<code>false</code>

## tls.certValidityDuration

Validity duration for TLS certificates (cluster, instance, and PgBouncer). Used only when [cert-manager](#) manages certificates. Format: Go duration (e.g. 2160h). Default: 8760h (1 year).

Value type	Example
 string	2160h

**tls.caValidityDuration**

Validity duration for the root CA certificate. Used only when [cert-manager](#) manages certificates. Format: Go duration (e.g. 26280h). Default: 8760h (1 year).

Value type	Example
 string	26280h

**tls.pgBackRestCertValidityDuration**

Validity duration for the `pgBackRest` client and repository host certificates

**standby.enabled**

Enables or disables running the cluster in a standby mode (read-only copy of an existing cluster, useful for disaster recovery, etc).

Value type	Example
 boolean	false

**standby.host**

Host address of the primary cluster this standby cluster connects to.

Value type	Example
 string	"<primary-ip>"

**standby.port**

Port number used by a standby copy to connect to the primary cluster.

Value type	Example
 string	"<primary-port>"

**openshift**

Set to `true` if the cluster is being deployed on OpenShift, set to `false` otherwise, or unset it for auto-detection.

Value type	Example
 boolean	<code>true</code>

**autoCreateUserSchema**

If set to `true`, the cluster will have automatically created schemas for the [custom user](#) defined in the `spec.users` subsection for all of the databases listed for this specific user.

Value type	Example
 boolean	<code>true</code>

**standby.repoName**

Name of the pgBackRest repository in the primary cluster this standby cluster connects to.

Value type	Example
 string	<code>repo1</code>

**standby.maxAcceptableLag**

The maximum amount of WAL data that the standby cluster can be behind the primary cluster. It is measured in bytes of WAL data. When the WAL lag exceeds this value, the primary pod in the standby cluster is marked as unready, the cluster goes into the `initializing` state, and a `StandbyLagging` condition is set in the status. If unset, lag is not checked. Use Kubernetes quantity format (for example, `10Mi`, `1Gi`).

--

Value type	Example
 string	10Mi

### `secrets.customRootCATLSecret.name`

Name of the secret with the custom root CA certificate and key for secure connections to the PostgreSQL server, see [Transport Layer Security \(TLS\)](#) for details.

Value type	Example
 string	cluster1-ca-cert

### `secrets.customRootCATLSecret.items`

Key-value pairs of the `key` (a key from the `secrets.customRootCATLSecret.name` secret) and the `path` (name on the file system) for the custom root certificate and key. See [Transport Layer Security \(TLS\)](#) for details.

Value type	Example
 subdoc	- key: "tls.crt" path: "root.crt"     - key: "tls.key" path: "root.key"

### `secrets.customTLSecret.name`

A secret with TLS certificate generated for *external* communications, see [Transport Layer Security \(TLS\)](#) for details.

Value type	Example
 string	cluster1-cert

### `secrets.customReplicationTLSecret.name`

A secret with TLS certificate generated for *internal* communications, see [Transport Layer Security \(TLS\)](#) for details.

--

Value type	Example
<b>S</b> string	replication1-cert

### users.name

The name of the PostgreSQL user.

Value type	Example
<b>S</b> string	rhino

### users.databases

Databases accessible by a specific PostgreSQL user with rights to create objects in them (the option is ignored for `postgres` user; also, modifying it can't be used to revoke the already given access).

Value type	Example
<b>S</b> string	zoo

### users.password.type

The set of characters used for password generation: can be either `ASCII` (default) or `AlphaNumeric`.

Value type	Example
<b>S</b> string	ASCII

### users.options

The `ALTER ROLE` options other than password (the option is ignored for `postgres` user).

Value type	Example
<b>S</b> string	"SUPERUSER"

### users.secretName

The custom name of the user’s Secret; if not specified, the default `<clusterName>-pguser-<userName>` variant will be used.

Value type	Example
 string	<code>"rhino-credentials"</code>

**users.grantPublicSchemaAccess**

Grants access to the `public` schema to the user for all databases associated with this user.

Value type	Example
 string	<code>false</code>

**authentication.rules**

Defines additional authentication rules for PostgreSQL host-based authentication (`pg_hba.conf`). Rules are applied after mandatory Operator rules and before the default `scram-sha-256` fallback. Use this to configure [LDAP authentication](#).

Value type	Example
 array	See <a href="#">LDAP authentication</a> for examples

**authentication.rules.connection**

Connection type for the rule: `local`, `host`, `hostssl`, `hostnoss1`, `hostgssenc`, or `hostnogssenc`.

Value type	Example
 string	<code>host</code>

**authentication.rules.method**

Authentication method to use when a connection matches this rule (e.g., `ldap`, `scram-sha-256`, `md5`).

Value type	Example

**s** string

ldap

## authentication.rules.users

Users to match. An empty list matches all users.

Value type	Example
<b>a</b> array	[ "percona" ]

## authentication.rules.databases

Databases to match. An empty list matches all databases.

Value type	Example
<b>a</b> array	[ "percona" ]

## authentication.rules.options

Options for the authentication method. Supported LDAP authentication options are:

- `ldapserver` - the LDAP server hostname
- `ldapport` - the port on which the LDAP server is reachable: `389` for plain LDAP and `636` for LDAPS
- `ldaptls` - Controls the connection type for the LDAP server. `1` is for LDAP over TLS.
- `ldapscheme` - Specifies the connection type. An alternative to `ldaptls`. Must be supported by LDAP server implementations.
- `ldapprefix` - A prefix for the username when constructing the DN. Use this option for Plain LDAP.
- `ldappsuffix` - A suffix for the username when constructing the DN. Use this option for Plain LDAP.
- `ldapbasedn` - The root DN in your LDAP directory tree where to start searching for the user. Use it for search+bind mode.
- `ldapbinddn` - The bind user DN that will be used for initial bind to LDAP server and username search. Use it for search+bind mode.
- `ldapbindpasswd` - The bind user password. Use it for search+bind mode.
- `ldapsearchattribute` - The attribute to match against when searching for the user. When not specified, the `uid` attribute will be used.

To learn more about LDAP, see [LDAP authentication](#) and the [PostgreSQL auth-ldap documentation](#) [↗](#).

Value type	Example
☰ subdoc	ldapservers: openldap ldapport: "389" ldapprefix: "uid=" ldapsuffix: ",ou=users,dc=example,dc=com"

## authentication.rules.hba

The authentication rules specified as a raw `pg_hba.conf` line. When non-empty, this line is used as-is and the structured fields are ignored.

Value type	Example
📄 string	"host all all 10.0.0.0/8 md5"

## config.files.secret.name

The name of the Secret object that stores the CA certificates for [LDAP over TLS \(LDAPS\)](#). The Operator mounts this certificate under `/etc/postgres`.

Value type	Example
📄 string	ldap-ca

## config.files.secret.items.key

The CA certificates to use for [LDAP over TLS \(LDAPS\)](#).

Value type	Example
📄 string	ca.crt

## config.files.secret.items.path

The path to the CA certificates to use for [LDAP over TLS \(LDAPS\)](#).

Value type	Example
📄 string	ldap-ca.crt

## databaseInitSQL.key

Data key for the [Custom configuration options ConfigMap](#) with the init SQL file, which will be executed at cluster creation time.

Value type	Example
<code>S</code> string	<code>init.sql</code>

## databaseInitSQL.name

Name of the [ConfigMap](#) with the init SQL file, which will be executed at cluster creation time.

Value type	Example
<code>S</code> string	<code>cluster1-init-sql</code>

## pause

Setting it to `true` gracefully stops the cluster, scaling workloads to zero and suspending CronJobs; setting it to `false` after shut down starts the cluster back.

Value type	Example
<code>S</code> string	<code>false</code>

## unmanaged

Setting it to `true` stops the Operator’s activity including the rollout and reconciliation of changes made in the Custom Resource; setting it to `false` starts the Operator’s activity back.

Value type	Example
<code>S</code> string	<code>false</code>

## dataSource subsection

Contains the configuration options for restoring from a backup onto a new cluster. For usage examples, see [Restore the backup to a new cluster \(cluster clone\)](#).

**dataSource.postgresCluster.clusterName**

Name of an existing cluster to use as the data source when restoring backup to a new cluster.

Value type	Example
<b>S</b> string	cluster1

**dataSource.postgresCluster.clusterNamespace**

Namespace of an existing cluster used as a data source (is needed if the new cluster will be created in a different namespace; needs the Operator deployed [in multi-namespace/cluster-wide mode](#)).

Value type	Example
<b>S</b> string	cluster1-namespace

**dataSource.postgresCluster.repoName**

Name of the pgBackRest repository in the source cluster that contains the backup to be restored to a new cluster.

Value type	Example
<b>S</b> string	repo1

**dataSource.postgresCluster.options**

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

Value type	Example
<b>S</b> string	

**dataSource.postgresCluster.tolerations.effect**

The [Kubernetes Pod tolerations](#)  effect for data migration.

Value type	Example
 string	NoSchedule

### `dataSource.postgresCluster.tolerations.key`

The [Kubernetes Pod tolerations](#)  key for data migration.

Value type	Example
 string	role

### `dataSource.postgresCluster.tolerations.operator`

The [Kubernetes Pod tolerations](#)  operator for data migration.

Value type	Example
 string	Equal

### `dataSource.postgresCluster.tolerations.value`

The [Kubernetes Pod tolerations](#)  value for data migration.

Value type	Example
 string	connection-poolers

### `dataSource.pgbackrest.stanza`

Name of the [pgBackRest stanza](#)  to use as the data source when restoring backup to a new cluster.

Value type	Example
 string	db

## `dataSource.pgbackrest.configuration.secret.name`

Name of the [Kubernetes Secret object](#) with custom pgBackRest configuration, which will be added to the pgBackRest configuration generated by the Operator.

Value type	Example
 string	<code>pgo-s3-creds</code>

## `dataSource.pgbackrest.global`

Settings, which are to be included in the `global` section of the pgBackRest configuration generated by the Operator.

Value type	Example
 subdoc	<code>/pgbackrest/postgres-operator/hippo/repo1</code>

## `dataSource.pgbackrest.repo.name`

Name of the pgBackRest repository.

Value type	Example
 string	<code>repo1</code>

## `dataSource.pgbackrest.repo.s3.bucket`

The [Amazon S3 bucket](#) or [Google Cloud Storage bucket](#) name used for backups. Bucket name should follow [Amazon naming rules](#) or [Google naming rules](#), and additionally, it can't contain dots.

Value type	Example
 string	<code>"my-bucket"</code>

## `dataSource.pgbackrest.repo.s3.endpoint`

The endpoint URL of the S3-compatible storage to be used for backups (not needed for the original Amazon S3 cloud).

Value type	Example
 string	"s3.ca-central-1.amazonaws.com"

### dataSource.pgbackrest.repo.s3.region

The [AWS region](#)  to use for Amazon and all S3-compatible storages.

Value type	Example
 boolean	"ca-central-1"

### dataSource.pgbackrest.tolerations.effect

The [Kubernetes Pod tolerations](#)  effect for pgBackRest at data migration.

Value type	Example
 string	NoSchedule

### dataSource.pgbackrest.tolerations.key

The [Kubernetes Pod tolerations](#)  key for pgBackRest at data migration.

Value type	Example
 string	role

### dataSource.pgbackrest.tolerations.operator

The [Kubernetes Pod tolerations](#)  operator for pgBackRest at data migration.

Value type	Example
 string	Equal

### dataSource.pgbackrest.tolerations.value

The [Kubernetes Pod tolerations](#) value for pgBackRest at data migration.

Value type	Example
<b>S</b> string	connection-poolers

**dataSource.volumes.pgDataVolume.pvcName**

The PostgreSQL data volume name for the [Persistent Volume Claim](#) used for data migration.

Value type	Example
<b>S</b> string	cluster1

**dataSource.volumes.pgDataVolume.directory**

The mount point for PostgreSQL data volume used for data migration.

Value type	Example
<b>S</b> string	cluster1

**dataSource.volumes.pgDataVolume.tolerations.effect**

The [Kubernetes Pod tolerations](#) effect for PostgreSQL data volume used for data migration.

Value type	Example
<b>S</b> string	NoSchedule

**dataSource.volumes.pgDataVolume.tolerations.key**

The [Kubernetes Pod tolerations](#) key for PostgreSQL data volume used for data migration.

Value type	Example
<b>S</b> string	role

### dataSource.volumes.pgDataVolume.tolerations.operator

The [Kubernetes Pod tolerations](#)  operator for PostgreSQL data volume used for data migration.

Value type	Example
 string	Equal

### dataSource.volumes.pgDataVolume.tolerations.value

The [Kubernetes Pod tolerations](#)  value for PostgreSQL data volume used for data migration.

Value type	Example
 string	connection-poolers

### dataSource.volumes.pgDataVolume.annotations

The [Kubernetes annotations](#)  metadata for PostgreSQL data volume used for data migration.

Value type	Example
 label	test-annotation: value

### dataSource.volumes.pgDataVolume.labels

The [Kubernetes labels](#)  for PostgreSQL data volume used for data migration.

Value type	Example
 label	test-label: value

### dataSource.volumes.pgWALVolume.pvcName

The PostgreSQL write-ahead logs volume name for the [Persistent Volume Claim](#)  used for data migration.

Value type	Example
 string	cluster1

**dataSource.volumes.pgWALVolume.directory**

The mount point for PostgreSQL write-ahead logs volume used for data migration.

Value type	Example
 string	cluster1

**dataSource.volumes.pgWALVolume.tolerations.effect**

The [Kubernetes Pod tolerations](#)  effect for PostgreSQL write-ahead logs volume used for data migration.

Value type	Example
 string	NoSchedule

**dataSource.volumes.pgWALVolume.tolerations.key**

The [Kubernetes Pod tolerations](#)  key for PostgreSQL write-ahead logs volume used for data migration.

Value type	Example
 string	role

**dataSource.volumes.pgWALVolume.tolerations.operator**

The [Kubernetes Pod tolerations](#)  operator for PostgreSQL write-ahead logs volume used for data migration.

Value type	Example
 string	Equal

**dataSource.volumes.pgWALVolume.tolerations.value**

The [Kubernetes Pod tolerations](#)  value for PostgreSQL write-ahead logs volume used for data migration.

Value type	Example

**s** string

connection-poolers

### **dataSource.volumes.pgWALVolume.annotations**

The [Kubernetes annotations](#) metadata for PostgreSQL write-ahead logs volume used for data migration.

Value type	Example
label	test-annotation: value

### **dataSource.volumes.pgWALVolume.labels**

The [Kubernetes labels](#) for PostgreSQL write-ahead logs volume used for data migration.

Value type	Example
label	test-label: value

### **dataSource.volumes.pgBackRestVolume.pvcName**

The pgBackRest volume name for the [Persistent Volume Claim](#) used for data migration.

Value type	Example
<b>s</b> string	cluster1

### **dataSource.volumes.pgBackRestVolume.directory**

The mount point for pgBackRest volume used for data migration.

Value type	Example
<b>s</b> string	cluster1

### **dataSource.volumes.pgBackRestVolume.tolerations.effect**

The [Kubernetes Pod tolerations](#) effect pgBackRest volume used for data migration.

Value type	Example
<b>S</b> string	NoSchedule

### `dataSource.volumes.pgBackRestVolume.tolerations.key`

The [Kubernetes Pod tolerations](#)  key for pgBackRest volume used for data migration.

Value type	Example
<b>S</b> string	role

### `dataSource.volumes.pgBackRestVolume.tolerations.operator`

The [Kubernetes Pod tolerations](#)  operator for pgBackRest volume used for data migration.

Value type	Example
<b>S</b> string	Equal

### `dataSource.volumes.pgBackRestVolume.tolerations.value`

The [Kubernetes Pod tolerations](#)  value for pgBackRest volume used for data migration.

Value type	Example
<b>S</b> string	connection-poolers

### `dataSource.volumes.pgBackRestVolume.annotations`

The [Kubernetes annotations](#)  metadata for pgBackRest volume used for data migration.

Value type	Example
<b>D</b> label	test-annotation: value

### `dataSource.volumes.pgBackRestVolume.labels`

The [Kubernetes labels](#)  for pgBackRest volume used for data migration.

Value type	Example
 label	<code>test-label: value</code>

**dataSource.apiGroup**

The name of the VolumeSnapshot API. It is required for bootstrapping a new cluster from a PVC snapshot.

Value type	Example
 string	<code>snapshot.storage.k8s.io</code>

**dataSource.kind**

Specifies what kind of resources serves as the data source

Value type	Example
 string	<code>VolumeSnapshot</code>

**dataSource.name**

Specifies what name of the PVC snapshot backup will be used as a data source for the restore.

Value type	Example
 string	<code>my-snapshot-backup-data</code>

**image**

The PostgreSQL Docker image to use.

Value type	Example
 string	<code>perconalab/percona-postgresql-operator:3.0.0-ppg18.4-1-postgres</code>

## imagePullPolicy

This option is used to set the [policy](#) for updating PostgreSQL images.

Value type	Example
 string	Always

## postgresVersion

The major version of PostgreSQL to use.

Value type	Example
 int	16

## port

The port number for PostgreSQL.

Value type	Example
 int	5432

## expose.annotations

The [Kubernetes annotations](#) metadata for PostgreSQL primary.

Value type	Example
 label	my-annotation: value1

## expose.labels

Set [labels](#) for the PostgreSQL primary.

Value type	Example
 label	my-label: value2

## expose.type

Specifies the type of [Kubernetes Service](#) for PostgreSQL primary.

Value type	Example
 string	LoadBalancer

## expose.loadBalancerClass

Define the implementation of the load balancer you want to use. This setting enables you to select a custom or specific load balancer class instead of the default one provided by the cloud provider.

Value type	Example
 string	eks.amazonaws.com/nlb

## expose.loadBalancerSourceRanges

The range of client IP addresses from which the load balancer should be reachable (if not set, there is no limitations).

Value type	Example
 string	"10.0.0.0/8"

## exposeReplicas.annotations

The [Kubernetes annotations](#) metadata for PostgreSQL replicas.

Value type	Example
 label	my-annotation: value1

## exposeReplicas.labels

Set [labels](#) for the PostgreSQL replicas.

--

Value type	Example
📄 label	<code>my-label: value2</code>

## exposeReplicas.type

Specifies the type of [Kubernetes Service](#)  for PostgreSQL replicas.

Value type	Example
📄 string	<code>LoadBalancer</code>

## exposeReplicas.loadBalancerClass

Define the implementation of the load balancer you want to use. This setting enables you to select a custom or specific load balancer class instead of the default one provided by the cloud provider.

Value type	Example
📄 string	<code>eks.amazonaws.com/nlb</code>

## exposeReplicas.loadBalancerSourceRanges

The range of client IP addresses from which the load balancer should be reachable (if not set, there is no limitations).

Value type	Example
📄 string	<code>"10.0.0.0/8"</code>

# Instances section

The `instances` section in the [deploy/cr.yaml](#)  file contains configuration options for PostgreSQL instances. This section contains at least one *cluster instance* with a number of *PostgreSQL instances* in it (cluster instances are groups of PostgreSQL instances used for fine-grained resources assignment).

## instances.metadata.labels

Set [labels](#)  for PostgreSQL Pods.

Value type	Example
🏷 label	<code>pg-cluster-label: cluster1</code>

### `instances.name`

The name of the PostgreSQL instance.

Value type	Example
📄 string	<code>rs 0</code>

### `instances.replicas`

The number of Replicas to create for the PostgreSQL instance.

Value type	Example
🔢 int	<code>3</code>

### `instances.env.name`

Name of an environment variable for PostgreSQL Pods. Read more about defining environment variables in [Kubernetes documentation](#) [↗](#).

Value type	Example
📄 string	<code>MY_ENV</code>

### `instances.env.value`

The value for an environment variable.

Value type	Example
📄 string	<code>1000</code>

## `instances.envFrom.secretRefName`

Name of a Secret or a ConfigMap, key/values of which are used as environment variables for PostgreSQL Pods.

Value type	Example
<code>S</code> string	<code>instance-env-secret</code>

## `instances.initContainer.image`

Defines an image for an init container to run before the main container in the Pod. The init container is typically used for setup tasks such as initializing filesystems, setting permissions, or preparing configuration.

Value type	Example
<code>S</code> string	<code>perconalab/percona-postgresql-operator:3.0.0</code>

## `instances.initContainer.resources.limits.cpu`

[Kubernetes CPU limits](#)  for an init container.

Value type	Example
<code>S</code> string	<code>2.0</code>

## `instances.initContainer.resources.limits.memory`

The [Kubernetes memory limits](#)  for an init container.

Value type	Example
<code>S</code> string	<code>4Gi</code>

## `instances.initContainer.securityContext`

Security settings for the init container. These settings control privileges, user/group IDs, and other security-related options. For more details, see the [Kubernetes documentation on SecurityContext](#) 

--

Value type	Example
 subdoc	<pre>runAsUser: 1001 runAsGroup: 1001 runAsNonRoot: true privileged: false allowPrivilegeEscalation: false readOnlyRootFilesystem: true</pre>

**instances.resources.requests.cpu**

[Kubernetes CPU requests](#)  for a PostgreSQL instance. It must not exceed the limit.

If you specify a limit and don't specify a request, Kubernetes uses the specified limit as the requested value for a resource.

Value type	Example
 string	1.0

**instances.resources.requests.memory**

[Kubernetes memory requests](#)  for a PostgreSQL instance. It must not exceed the limit.

If you specify a limit and don't specify a request, Kubernetes uses the specified limit as the requested value for a resource.

Value type	Example
 string	3Gi

**instances.resources.limits.cpu**

[Kubernetes CPU limits](#)  for a PostgreSQL instance.

Value type	Example
 string	2.0

**instances.resources.limits.memory**

The [Kubernetes memory limits](#) for a PostgreSQL instance.

Value type	Example
<span>S</span> string	4Gi

**instances.containers.replicaCertCopy.resources.requests.cpu**

[Kubernetes CPU requests](#) for a replica-cert-copy sidecar container. It must not exceed the limit.

If you specify a limit and don't specify a request, Kubernetes uses the specified limit as the requested value for a resource.

Value type	Example
<span>S</span> string	100m

**instances.containers.replicaCertCopy.resources.requests.memory**

[Kubernetes memory requests](#) for a replica-cert-copy sidecar container. It must not exceed the limit.

If you specify a limit and don't specify a request, Kubernetes uses the specified limit as the requested value for a resource.

Value type	Example
<span>S</span> string	120Mi

**instances.containers.replicaCertCopy.resources.limits.cpu**

[Kubernetes CPU limits](#) for replica-cert-copy sidecar container.

Value type	Example
<span>S</span> string	200m

**instances.containers.replicaCertCopy.resources.limits.memory**

The [Kubernetes memory limits](#) for replica-cert-copy sidecar container.

Value type	Example
<b>S</b> string	128Mi

**instances.topologySpreadConstraints.maxSkew**

The degree to which Pods may be unevenly distributed under the [Kubernetes Pod Topology Spread Constraints](#) .

Value type	Example
<b>I</b> int	1

**instances.topologySpreadConstraints.topologyKey**

The key of node labels for the [Kubernetes Pod Topology Spread Constraints](#) .

Value type	Example
<b>S</b> string	my-node-label

**instances.topologySpreadConstraints.whenUnsatisfiable**

What to do with a Pod if it doesn't satisfy the [Kubernetes Pod Topology Spread Constraints](#) .

Value type	Example
<b>S</b> string	DoNotSchedule

**instances.topologySpreadConstraints.labelSelector.matchLabels**

The Label selector for the [Kubernetes Pod Topology Spread Constraints](#) .

Value type	Example
 label	postgres-operator.crunchydata.com/instance-set: instance1

**instances.tolerations.effect**

The [Kubernetes Pod tolerations](#) effect for the PostgreSQL instance.

Value type	Example
<span>S</span> string	NoSchedule

**instances.tolerations.key**

The [Kubernetes Pod tolerations](#) key for the PostgreSQL instance.

Value type	Example
<span>S</span> string	role

**instances.tolerations.operator**

The [Kubernetes Pod tolerations](#) operator for the PostgreSQL instance.

Value type	Example
<span>S</span> string	Equal

**instances.tolerations.value**

The [Kubernetes Pod tolerations](#) value for the PostgreSQL instance.

Value type	Example
<span>S</span> string	connection-poolers

**instances.priorityClassName**

The [Kubernetes Pod priority class](#) for PostgreSQL instance Pods.

Value type	Example
<span>S</span> string	high-priority

## instances.securityContext

A custom [Kubernetes Security Context for a Pod](#) to be used instead of the default one.

Value type	Example
≡ subdoc	<pre>fsGroup: 1001 runAsUser: 1001 runAsNonRoot: true fsGroupChangePolicy: "OnRootMismatch" runAsGroup: 1001 selinuxOptions:   type: spc_t   level: s0:c123,c456 seccompProfile:   type: Localhost   localhostProfile: localhost/profile.json supplementalGroups: - 1001 sysctls: - name: net.ipv4.tcp_keepalive_time   value: "600" - name: net.ipv4.tcp_keepalive_intvl   value: "60"</pre>

## instances.walVolumeClaimSpec.accessModes

The [Kubernetes PersistentVolumeClaim](#) access modes for the PostgreSQL Write-ahead Log storage.

Value type	Example
 string	ReadWriteOnce

## instances.walVolumeClaimSpec.storageClassName

Set the [Kubernetes storage class](#) to use with the PostgreSQL Write-ahead Log storage [PersistentVolumeClaim](#).

Value type	Example
 string	standard

## instances.walVolumeClaimSpec.resources.requests.storage

The [Kubernetes storage requests](#) for the storage the PostgreSQL instance will use.

Value type	Example
<b>S</b> string	1Gi

**instances.dataVolumeClaimSpec.accessModes**

The [Kubernetes PersistentVolumeClaim](#) access modes for the PostgreSQL storage.

Value type	Example
<b>S</b> string	ReadWriteOnce

**instances.dataVolumeClaimSpec.storageClassName**

Set the [Kubernetes storage class](#) to use with PostgreSQL Cluster [PersistentVolumeClaim](#) for the PostgreSQL storage.

Value type	Example
<b>S</b> string	standard

**instances.dataVolumeClaimSpec.resources.requests.storage**

The [Kubernetes storage requests](#) for the storage the PostgreSQL instance will use.

Value type	Example
<b>S</b> string	1Gi

**instances.dataVolumeClaimSpec.resources.limits.storage**

The [Kubernetes storage limits](#) for the storage the PostgreSQL instance will use.

Value type	Example
<b>S</b> string	5Gi

## instances.tablespaceVolumes.name

Name for the custom [tablespace volume](#).

Value type	Example
<b>S</b> string	user

## instances.tablespaceVolumes.dataVolumeClaimSpec.accessModes

The [Kubernetes PersistentVolumeClaim](#) [↗](#) access modes for the tablespace volume.

Value type	Example
<b>S</b> string	ReadWriteOnce

## instances.tablespaceVolumes.dataVolumeClaimSpec.resources.requests.storage

The [Kubernetes storage requests](#) [↗](#) for the tablespace volume.

Value type	Example
<b>S</b> string	1Gi

## instances.sidecars subsection

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

### instances.sidecars.image

Image for the [custom sidecar container](#) for PostgreSQL Pods.

Value type	Example
<b>S</b> string	busybox:latest

## instances.sidecars.name

Name of the [custom sidecar container](#) for PostgreSQL Pods.

Value type	Example
 string	testcontainer

## instances.sidecars.imagePullPolicy

This option is used to set the [policy](#)  for the PostgreSQL Pod sidecar container.

Value type	Example
 string	Always

## instances.sidecars.env

The [environment variables set as key-value pairs](#)  for the [custom sidecar container](#) for PostgreSQL Pods.

Value type	Example
 subdoc	

## instances.sidecars.envFrom

The [environment variables set as key-value pairs in ConfigMaps](#)  for the [custom sidecar container](#) for PostgreSQL Pods.

Value type	Example
 subdoc	

## instances.sidecars.command

Command for the [custom sidecar container](#) for PostgreSQL Pods.

Value type	Example

 array

`["/bin/sh"]`

## **instances.sidecars.args**

Command arguments for the [custom sidecar container](#) for PostgreSQL Pods.

Value type	Example
 array	<code>["-c", "while true; do trap 'exit 0' SIGINT SIGTERM SIGQUIT SIGKILL; done;"]</code>

## **instances.sidecarVolumes.name**

The name of the [volume](#)  to attach to PostgreSQL instance Pods for use by [custom sidecar containers](#). Supports secret, configMap, NFS and other volume types.

Value type	Example
 subdoc	sidecar-secret

## **instances.sidecarVolumes.secret.secretName**

The name of the [volume Secret](#)  to attach to PostgreSQL instance Pods for use by [custom sidecar containers](#).

Value type	Example
:material-text-string: string	mysecret

## **instances.sidecarVolumes.configMap.name**

The name of the configMap object to attach to PostgreSQL instance Pods for use by [custom sidecar containers](#).

Value type	Example
:material-text-string: string	sidecar-config

## **instances.sidecarVolumes.nfs.server**

The hostname of the NFS server that will provide remote filesystem to the [custom sidecar containers](#) in PostgreSQL instance Pods.

Value type	Example
:material-text-string: string	"nfs-service.storage.svc.cluster.local"

**instances.sidecarVolumes.nfs.path**

The path on the NFS server that will be provided as a remote filesystem to the [custom sidecar containers](#) in PostgreSQL instance Pods.

Value type	Example
:material-text-string: string	"nfs-service.storage.svc.cluster.local"

**instances.sidecarPVCs**

[PersistentVolumeClaims](#) [↗](#) that the operator creates and mounts for [custom sidecar containers](#) in PostgreSQL instance Pods.

You can use PVCs with sidecar containers only when you deploy a new cluster. Updates to running cluster are not supported.

Value type	Example
≡ subdoc	<pre>- name: sidecar-volume-claim   spec:     resources:       requests:         storage: 1Gi     volumeMode: Filesystem     accessModes:       - ReadWriteOnce</pre>

## Backup section

The `backup` section in the [deploy/cr.yaml](#) [↗](#) file contains the following configuration options for the regular Percona Distribution for PostgreSQL backups.

**backups.enabled**

Enables to turn on/off backups for the cluster. Use this option with caution. Read more in [Disable backups](#).

Value type	Example
 string	true

### backups.trackLatestRestorableTime

Enables or disables [tracking the latest restorable time](#) for latest successful backup (on by default). It can be turned off to reduce the S3 API usage.

Value type	Example
 boolean	true

### backups.volumeSnapshots.className

Name of the [VolumeSnapshotClass](#)  to use when creating [PVC snapshots](#). When set, the Operator creates a volume snapshot in coordination with each backup. Snapshots enable much faster restores when provisioning new clusters. Requires the `BackupSnapshots=true` feature gate.

Value type	Example
 string	csi-gce-pd-snapshot-class

### backups.volumeSnapshots.mode

Specifies the type of PVC snapshot-based backups.

Value type	Example
 string	offline

### backups.volumeSnapshots.schedule

Specifies the schedule in Cron format to run PVC snapshot-based backups automatically.

Value type	Example

**S** string

"0 3 \* \* \*"

## backups.pgbackrest.metadata.labels

Set [labels](#)  for pgBackRest Pods.

Value type	Example
 label	pg-cluster-label: cluster1

## backups.pgbackrest.image

The Docker image for [pgBackRest](#).

Value type	Example
<b>S</b> string	docker.io/percona/percona-pgbackrest:2.58.0-2

## backups.pgbackrest.env.name

Name of an environment variable for pgBackRest Pods. Read more about defining environment variables in [Kubernetes documentation](#) .

Value type	Example
<b>S</b> string	MY_ENV

## backups.pgbackrest.env.value

The value for an environment variable.

Value type	Example
<b>S</b> string	1000

## backups.pgbackrest.envFrom.secretRefName

Name of a Secret or a ConfigMap, key/values of which are used as environment variables for pgBouncer Pods.

Value type	Example
<b>S</b> string	repo-host-env-secret

## backups.pgbackrest.containers.pgbackrest.resources.requests.cpu

[Kubernetes CPU requests](#)  for a pgBackRest container. It must not exceed the limit.

If you specify a limit and don't specify a request, Kubernetes uses the specified limit as the requested value for a resource.

Value type	Example
<b>S</b> string	150m

## backups.pgbackrest.containers.pgbackrest.resources.requests.memory

[Kubernetes memory requests](#)  for a pgBackRest container. It must not exceed the limit.

If you specify a limit and don't specify a request, Kubernetes uses the specified limit as the requested value for a resource.

Value type	Example
<b>S</b> string	120Mi

## backups.pgbackrest.containers.pgbackrest.resources.limits.cpu

[Kubernetes CPU limits](#)  for a pgBackRest container.

Value type	Example
<b>S</b> string	1.0

## backups.pgbackrest.containers.pgbackrest.resources.limits.memory

The [Kubernetes memory limits](#)  for a pgBackRest container.

Value type	Example
------------	---------

**S** string

1Gi

## **backups.pgbackrest.containers.pgbackrestConfig.resources.limits.cpu**

[Kubernetes CPU limits](#)  for pgbackrest-config sidecar container.

Value type	Example
<b>S</b> string	1.0

## **backups.pgbackrest.containers.pgbackrestConfig.resources.limits.memory**

The [Kubernetes memory limits](#)  for pgbackrest-config sidecar container.

Value type	Example
<b>S</b> string	1Gi

## **backups.pgbackrest.containers.pgbackrestConfig.resources.requests.cpu**

[Kubernetes CPU requests](#)  for a pgbackrest-config sidecar container. It must not exceed the limit.

If you specify a limit and don't specify a request, Kubernetes uses the specified limit as the requested value for a resource.

Value type	Example
<b>S</b> string	150m

## **backups.pgbackrest.containers.pgbackrestConfig.resources.requests.memory**

[Kubernetes memory requests](#)  for a pgbackrest-config sidecar container. It must not exceed the limit.

If you specify a limit and don't specify a request, Kubernetes uses the specified limit as the requested value for a resource.

Value type	Example
<b>S</b> string	120Mi

## backups.pgbackrest.configuration.secret.name

Name of the [Kubernetes Secret object](#) with custom pgBackRest configuration, which will be added to the pgBackRest configuration generated by the Operator.

Value type	Example
 string	cluster1-pgbackrest-secrets

## backups.pgbackrest.jobs.backoffLimit

The number of retries to make a backup with incremental pauses of 10 seconds, 20 seconds, etc. between retries. By default it's 0, which means that pgBackRest job Pod fails after first unsuccessful attempt (causing creation of a new Pod on failure).

Value type	Example
 int	2

## backups.pgbackrest.jobs.restartPolicy

The [Kubernetes Pod restart policy](#) for pgBackRest jobs.

Value type	Example
 string	OnFailure

## backups.pgbackrest.jobs.priorityClassName

The [Kubernetes Pod priority class](#) for pgBackRest jobs.

Value type	Example
 string	high-priority

## backups.pgbackrest.jobs.resources.limits.cpu

[Kubernetes CPU limits](#) for a pgBackRest job.

--

Value type	Example
<b>I</b> int	200

## backups.pgbackrest.jobs.resources.limits.memory

The [Kubernetes memory limits](#) for a pgBackRest job.

Value type	Example
<b>S</b> string	128Mi

## backups.pgbackrest.jobs.resources.requests.cpu

[Kubernetes CPU requests](#) for a pgBackRest job. It must not exceed the limit.

If you specify a limit and don't specify a request, Kubernetes uses the specified limit as the requested value for a resource.

Value type	Example
<b>S</b> string	150m

## backups.pgbackrest.jobs.resources.requests.memory

[Kubernetes memory requests](#) for pgBackRest job. It must not exceed the limit.

If you specify a limit and don't specify a request, Kubernetes uses the specified limit as the requested value for a resource.

Value type	Example
<b>S</b> string	120Mi

## backups.pgbackrest.jobs.tolerations.effect

The [Kubernetes Pod tolerations](#) effect for a backup job.

Value type	Example

**S** string

NoSchedule

## backups.pgbackrest.jobs.tolerations.key

The [Kubernetes Pod tolerations](#)  key for a backup job.

Value type	Example
<b>S</b> string	role

## backups.pgbackrest.jobs.tolerations.operator

The [Kubernetes Pod tolerations](#)  operator for a backup job.

Value type	Example
<b>S</b> string	Equal

## backups.pgbackrest.jobs.tolerations.value

The [Kubernetes Pod tolerations](#)  value for a backup job.

Value type	Example
<b>S</b> string	connection-poolers

## backups.pgbackrest.jobs.securityContext

A custom [Kubernetes Security Context for a Pod](#)  to be used instead of the default one.

Value type	Example
≡ subdoc	

```

fsGroup: 1001
runAsUser: 1001
runAsNonRoot: true
fsGroupChangePolicy: "OnRootMismatch"
runAsGroup: 1001
selinuxOptions:
 type: spc_t
 level: s0:c123,c456
seccompProfile:
 type: Localhost
 localhostProfile: localhost/profile.json
supplementalGroups:
- 1001
sysctls:
- name: net.ipv4.tcp_keepalive_time
 value: "600"
- name: net.ipv4.tcp_keepalive_intvl
 value: "60"

```

## backups.pgbackrest.global

Settings, which are to be included in the `global` section of the pgBackRest configuration generated by the Operator.

Value type	Example
≡ subdoc	repo1-retention-full: "14" repo1-retention-full-type: time repo1-path: /pgbackrest/postgres-operator/cluster1/repo1 repo1-cipher-type: aes-256-cbc repo1-s3-uri-style: path repo2-path: /pgbackrest/postgres-operator/cluster1-multi-repo/repo2 repo3-path: /pgbackrest/postgres-operator/cluster1-multi-repo/repo3 repo4-path: /pgbackrest/postgres-operator/cluster1-multi-repo/repo4

## backups.pgbackrest.repoHost.sidecars.name

The name of a [custom sidecar container](#) for pgBackRest Pods.

Value type	Example
📄 string	testcontainer

## backups.pgbackrest.repoHost.sidecars.image

The image used to deploy a [custom sidecar container](#) for pgBackRest Pods.

Value type	Example
 string	<code>busybox:latest</code>

**backups.pgbackrest.repoHost.sidecars.command**

The command to use inside a custom sidecar container for pgBackRest Pods

Value type	Example
 string	<code>[ "sleep", "30d" ]</code>

**backups.pgbackrest.repoHost.sidecars.securityContext**

Security settings for the sidecar container. These settings control privileges, user/group IDs, and other security-related options. For more details, see the [Kubernetes documentation on SecurityContext](#)

Value type	Example
 string	<code>{}</code>

**backups.pgbackrest.repoHost.sidecarVolumes.name**

The name of the [volume](#) to attach to pgBackRest repo host Pods for use by [custom sidecar containers](#). Supports secret, configMap, NFS and other volume types.

Value type	Example
 subdoc	<code>sidecar-secret</code>

**backups.pgbackrest.repoHost.sidecarVolumes.secret.secretName**

The name of the [volume Secret](#) to attach to pgBackRest repo host Pods for use by [custom sidecar containers](#).

Value type	Example
<code>:material-text-string: string</code>	<code>mysecret</code>

**backups.pgbackrest.repoHost.sidecarVolumes.configMap.name**

The name of the configMap object to attach to pgBackRest repo host Pods for use by [custom sidecar containers](#).

Value type	Example
:material-text-string: string	sidecar-config

**backups.pgbackrest.repoHost.sidecarVolumes.nfs.server**

The hostname of the NFS server that will provide remote filesystem to the [custom sidecar containers](#) in pgBackRest repo host Pods Pods.

Value type	Example
:material-text-string: string	"nfs-service.storage.svc.cluster.local"

**backups.pgbackrest.repoHost.sidecarVolumes.nfs.path**

The path on the NFS server that will be provided as a remote filesystem to the [custom sidecar containers](#) in pgBackRest repo host Pods` Pods.

Value type	Example
:material-text-string: string	"nfs-service.storage.svc.cluster.local"

**backups.pgbackrest.repoHost.sidecarPVCs**

[PersistentVolumeClaims](#)  that the Operator creates and mounts for [custom sidecar containers](#) in pgBackRest repo host Pods.

You can use PVCs with sidecar containers only when you deploy a new cluster. Updates to running cluster are not supported.

Value type	Example
 subdoc	

```
- name: sidecar-volume-claim
 spec:
 resources:
 requests:
 storage: 1Gi
 volumeMode: Filesystem
 accessModes:
 - ReadWriteOnce
```

### **backups.pgbackrest.repoHost.resources.requests.cpu**

[Kubernetes CPU requests](#)  for a pgBackRest repo. It must not exceed the limit.

If you specify a limit and don't specify a request, Kubernetes uses the specified limit as the requested value for a resource.

Value type	Example
 string	150m

### **backups.pgbackrest.repoHost.resources.requests.memory**

[Kubernetes memory requests](#)  for pgBackRest repo. It must not exceed the limit.

If you specify a limit and don't specify a request, Kubernetes uses the specified limit as the requested value for a resource.

Value type	Example
 string	120Mi

### **backups.pgbackrest.repoHost.resources.limits.cpu**

[Kubernetes CPU limits](#)  for a pgBackRest repo.

Value type	Example
 int	200

### **backups.pgbackrest.repoHost.resources.limits.memory**

The [Kubernetes memory limits](#)  for a pgBackRest repo.

Value type	Example
<b>S</b> string	128Mi

## backups.pgbackrest.repoHost.priorityClassName

The [Kubernetes Pod priority class](#)  for pgBackRest repo.

Value type	Example
<b>S</b> string	high-priority

## backups.pgbackrest.repoHost.topologySpreadConstraints.maxSkew

The degree to which Pods may be unevenly distributed under the [Kubernetes Pod Topology Spread Constraints](#) .

Value type	Example
<b>I</b> int	1

## backups.pgbackrest.repoHost.topologySpreadConstraints.topologyKey

The key of node labels for the [Kubernetes Pod Topology Spread Constraints](#) .

Value type	Example
<b>S</b> string	my-node-label

## backups.pgbackrest.repoHost.topologySpreadConstraints.whenUnsatisfiable

What to do with a Pod if it doesn't satisfy the [Kubernetes Pod Topology Spread Constraints](#) .

Value type	Example
<b>S</b> string	ScheduleAnyway

**backups.pgbackrest.repoHost.topologySpreadConstraints.labelSelector.matchLabels**

The Label selector for the [Kubernetes Pod Topology Spread Constraints](#).

Value type	Example
label	postgres-operator.crunchydata.com/pgbackrest: ""

**backups.pgbackrest.repoHost.affinity.podAntiAffinity**

[Pod anti-affinity](#), allows setting the standard Kubernetes affinity constraints of any complexity.

Value type	Example
subdoc	

**backups.pgbackrest.repoHost.tolerations.effect**

The [Kubernetes Pod tolerations](#) effect for pgBackRest repo.

Value type	Example
string	NoSchedule

**backups.pgbackrest.repoHost.tolerations.key**

The [Kubernetes Pod tolerations](#) key for pgBackRest repo.

Value type	Example
string	role

**backups.pgbackrest.repoHost.tolerations.operator**

The [Kubernetes Pod tolerations](#) operator for pgBackRest repo.

Value type	Example

**S** string

Equal

## backups.pgbackrest.repoHost.tolerations.value

The [Kubernetes Pod tolerations](#) value for pgBackRest repo.

Value type	Example
<b>S</b> string	connection-poolers

## 'backups.pgbackrest.repoHost.securityContext'

A custom [Kubernetes Security Context for a Pod](#) to be used instead of the default one.

Value type	Example
 subdoc	<pre>fsGroup: 1001 runAsUser: 1001 runAsNonRoot: true fsGroupChangePolicy: "OnRootMismatch" runAsGroup: 1001 seLinuxOptions:   type: spc_t   level: s0:c123,c456 seccompProfile:   type: Localhost   localhostProfile: localhost/profile.json supplementalGroups: - 1001 sysctls: - name: net.ipv4.tcp_keepalive_time   value: "600" - name: net.ipv4.tcp_keepalive_intvl   value: "60"</pre>

## backups.pgbackrest.manual.repoName

Name of the pgBackRest repository for on-demand backups.

Value type	Example
<b>S</b> string	repo1

## backups.pgbackrest.manual.options

The on-demand backup command-line options which will be passed to pgBackRest for on-demand backups.

Value type	Example
<b>S</b> string	<code>--type=full</code>

**backups.pgbackrest.manual.initialDelaySeconds**

The time to delay a backup start after the backup Pod is scheduled. The backup process wait for the defined time before it connectsto the API server to start a backup.

Value type	Example
<b>I</b> int	<code>120</code>

**backups.pgbackrest.repos.name**

Name of the pgBackRest repository for backups.

Value type	Example
<b>S</b> string	<code>repo1</code>

**backups.pgbackrest.repos.schedules.full**

Scheduled time to make a full backup specified in the [crontab format](#) .

Value type	Example
<b>S</b> string	<code>0 0 \* \* 6</code>

**backups.pgbackrest.repos.schedules.differential**

Scheduled time to make a differential backup specified in the [crontab format](#) .

Value type	Example
<b>S</b> string	<code>0 0 \* \* 6</code>

## backups.pgbackrest.repos.volume.volumeClaimSpec.accessModes

The [Kubernetes PersistentVolumeClaim](#) access modes for the pgBackRest Storage.

Value type	Example
<span>S</span> string	ReadWriteOnce

## backups.pgbackrest.repos.volume.volumeClaimSpec.storageClassName

Set the [Kubernetes Storage Class](#) to use with the Percona Operator for PostgreSQL backups stored on [Persistent Volume](#).

Value type	Example
<span>S</span> string	standard

## backups.pgbackrest.repos.volume.volumeClaimSpec.resources.requests.storage

The [Kubernetes storage requests](#) for the pgBackRest storage.

Value type	Example
<span>S</span> string	1Gi

## backups.pgbackrest.repos.s3.bucket

The [Amazon S3 bucket](#) name used for backups

Value type	Example
<span>S</span> string	"my-bucket"
.	

## backups.pgbackrest.repos.s3.endpoint

The endpoint URL of the S3-compatible storage to be used for backups (not needed for the original Amazon S3 cloud).

Value type	Example
 string	"s3.ca-central-1.amazonaws.com"

### backups.pgbackrest.repos.s3.region

The [AWS region](#)  to use for Amazon and all S3-compatible storages.

Value type	Example
 string	"ca-central-1"

### backups.pgbackrest.repos.gcs.bucket

The [Google Cloud Storage bucket](#)  name used for backups.

Value type	Example
 string	"my-bucket"

### backups.pgbackrest.repos.azure.container

Name of the [Azure Blob Storage container](#)  for backups.

Value type	Example
 string	my-container

### backups.restore.tolerations.effect

The [Kubernetes Pod tolerations](#)  effect for the backup restore job.

Value type	Example
 string	NoSchedule

## backups.restore.tolerations.key

The [Kubernetes Pod tolerations](#) key for the backup restore job.

Value type	Example
 string	role

## backups.restore.tolerations.operator

The [Kubernetes Pod tolerations](#) operator for the backup restore job.

Value type	Example
 string	Equal

## backups.restore.tolerations.value

The [Kubernetes Pod tolerations](#) value for the backup restore job.

Value type	Example
 string	connection-poolers

# PMM section

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

## pmm.enabled

Enables or disables [monitoring Percona Distribution for PostgreSQL cluster with PMM](#).

Value type	Example
 boolean	false

## pmm.image

[Percona Monitoring and Management \(PMM\) Client](#)  Docker image.

Value type	Example
 string	<code>percona/pmm-client:3.7.1</code>

### `pmm.imagePullPolicy`

This option is used to set the [policy](#)  for updating PMM Client images.

Value type	Example
 string	<code>IfNotPresent</code>

### `pmm.secret`

Name of the [Kubernetes Secret object](#)  for the PMM Server password.

Value type	Example
 string	<code>cluster1-pmm-secret</code>

### `pmm.serverHost`

Address of the PMM Server to collect data from the cluster.

Value type	Example
 string	<code>monitoring-service</code>

### `pmm.customClusterName`

A custom name to define for a cluster. PMM Server uses this name to properly parse the metrics and display them on dashboards. Using a custom name is useful for clusters deployed in different data centers - PMM Server connects them and monitors them as one deployment. Another use case is for clusters deployed with the same name in different namespaces - PMM treats each cluster separately.

Value type	Example

**s** string

postgresql-cluster

### **pmm.resources.requests.cpu**

[Kubernetes CPU requests](#)  for a PMM Client container. It must not exceed the limit.

If you specify a limit and don't specify a request, Kubernetes uses the specified limit as the requested value for a resource.

Value type	Example
<b>s</b> string	150m

### **pmm.resources.requests.memory**

[Kubernetes memory requests](#)  for PMM Client container. It must not exceed the limit.

If you specify a limit and don't specify a request, Kubernetes uses the specified limit as the requested value for a resource.

Value type	Example
<b>s</b> string	120Mi

### **pmm.resources.limits.cpu**

[Kubernetes CPU limits](#)  for a PMM Client container.

Value type	Example
<b>i</b> int	200

### **pmm.resources.limits.memory**

The [Kubernetes memory limits](#)  for a PMM Client container.

Value type	Example
<b>s</b> string	128Mi

## pmm.querySource

Query source to track PostgreSQL statistics. Either `pg_stat_monitor` (`pgstatmonitor`, the default value) or `pg_stat_statements` (`pgstatstatements`) can be used.

Value type	Example
 string	<code>pgstatmonitor</code>

## pmm.postgresParams

Additional parameters which will be passed to the `pmm-admin add postgresql` command for PostgreSQL Pods.

Value type	Example
 string	

# Proxy section

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

## proxy.pgBouncer.metadata.labels

Set [labels](#)  for pgBouncer Pods.

Value type	Example
 label	<code>pg-cluster-label: cluster1</code>

## proxy.pgBouncer.replicas

The number of the pgBouncer Pods to provide connection pooling.

Value type	Example
 int	<code>3</code>

### proxy.pgBouncer.image

Docker image for the [pgBouncer](#) connection pooler.

Value type	Example
 string	docker.io/percona/percona-pgbouncer:1.25.2-1

### proxy.pgBouncer.env.name

Name of an environment variable for pgBouncer Pods. Read more about defining environment variables in [Kubernetes documentation](#).

Value type	Example
 string	MY_ENV

### proxy.pgBouncer.env.value

The value for an environment variable.

Value type	Example
 string	1000

### proxy.pgBouncer.envFrom.secretRefName

Name of a Secret or a ConfigMap, key/values of which are used as environment variables for pgBouncer Pods.

Value type	Example
 string	pgbouncer-env-secret

### proxy.pgBouncer.exposeSuperusers

Enables or disables [exposing superuser user through pgBouncer](#).

Value type	Example

 boolean

false

## proxy.pgBouncer.resources.requests.cpu

[Kubernetes CPU requests](#)  for a pgBouncer container. It must not exceed the limit.

If you specify a limit and don't specify a request, Kubernetes uses the specified limit as the requested value for a resource.

Value type	Example
 string	150m

## proxy.pgBouncer.resources.requests.memory

[Kubernetes memory requests](#)  for a pgBouncer container. It must not exceed the limit.

If you specify a limit and don't specify a request, Kubernetes uses the specified limit as the requested value for a resource.

Value type	Example
 string	120Mi

## proxy.pgBouncer.resources.limits.cpu

[Kubernetes CPU limits](#)  for a pgBouncer container.

Value type	Example
 string	200m

## proxy.pgBouncer.resources.limits.memory

The [Kubernetes memory limits](#)  for a pgBouncer container.

Value type	Example
 string	128Mi

**proxy.pgBouncer.containers.pgouncerConfig.resources.limits.cpu**

[Kubernetes CPU limits](#)  for pgbouncer-config sidecar container.

Value type	Example
 string	1.0

**proxy.pgBouncer.containers.pgouncerConfig.resources.limits.memory**

The [Kubernetes memory limits](#)  for pgbouncer-config sidecar container.

Value type	Example
 string	1Gi

**proxy.pgBouncer.containers.pgouncerConfig.resources.requests.cpu**

[Kubernetes CPU requests](#)  for a pgbouncer-config sidecar container. It must not exceed the limit.

If you specify a limit and don't specify a request, Kubernetes uses the specified limit as the requested value for a resource.

Value type	Example
 string	150m

**proxy.pgBouncer.containers.pgouncerConfig.resources.requests.memory**

[Kubernetes memory requests](#)  for a pgbouncer-config sidecar container. It must not exceed the limit.

If you specify a limit and don't specify a request, Kubernetes uses the specified limit as the requested value for a resource.

Value type	Example
 string	120Mi

**proxy.pgBouncer.expose.type**

Specifies the type of [Kubernetes Service](#) for pgBouncer.

Value type	Example
<span>S</span> string	ClusterIP

**proxy.pgBouncer.expose.annotations**

The [Kubernetes annotations](#) metadata for pgBouncer.

Value type	Example
<span>D</span> label	my-annotation: value1

**proxy.pgBouncer.expose.labels**

Set [labels](#) for the pgBouncer Service.

Value type	Example
<span>D</span> label	pg-cluster-label: cluster1

**proxy.pgBouncer.expose.loadBalancerClass**

Define the implementation of the load balancer you want to use. This setting enables you to select a custom or specific load balancer class instead of the default one provided by the cloud provider.

Value type	Example
<span>S</span> string	eks.amazonaws.com/nlb

**proxy.pgBouncer.expose.loadBalancerSourceRanges**

The range of client IP addresses from which the load balancer should be reachable (if not set, there is no limitations).

Value type	Example
<span>S</span> string	"10.0.0.0/8"

proxy.pgBouncer.affinity.podAntiAffinity

[Pod anti-affinity](#), allows setting the standard Kubernetes affinity constraints of any complexity.

Value type	Example
 subdoc	

‘proxy.pgBouncer.securityContext’

A custom [Kubernetes Security Context for a Pod](#)  to be used instead of the default one.

Value type	Example
 subdoc	<pre>fsGroup: 1001 runAsUser: 1001 runAsNonRoot: true fsGroupChangePolicy: "OnRootMismatch" runAsGroup: 1001 seLinuxOptions:   type: spc_t   level: s0:c123,c456 seccompProfile:   type: Localhost   localhostProfile: localhost/profile.json supplementalGroups: - 1001 sysctls: - name: net.ipv4.tcp_keepalive_time   value: "600" - name: net.ipv4.tcp_keepalive_intvl   value: "60"</pre>

proxy.pgBouncer.config

Custom configuration options for pgBouncer. Please note that configuration changes are automatically applied to the running instances without validation, so having an invalid config can make the cluster unavailable.

Value type	Example
 subdoc	<pre>global: pool_mode: transaction</pre>

# proxy.pgBouncer.sidecars subsection

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

## proxy.pgBouncer.sidecars.image

Image for the [custom sidecar container](#) for pgBouncer Pods.

Value type	Example
 string	<code>mycontainer1:latest</code>

## proxy.pgBouncer.sidecars.name

Name of the [custom sidecar container](#) for pgBouncer Pods.

Value type	Example
 string	<code>testcontainer</code>

## proxy.pgBouncer.sidecars.imagePullPolicy

This option is used to set the [policy](#) for the pgBouncer Pod sidecar container.

Value type	Example
 string	<code>Always</code>

## proxy.pgBouncer.sidecars.env

The [environment variables set as key-value pairs](#) for the [custom sidecar container](#) for pgBouncer Pods.

Value type	Example
 subdoc	

## proxy.pgBouncer.sidecars.envFrom

The [environment variables](#) set as [key-value pairs](#) in [ConfigMaps](#) [↗](#) for the [custom sidecar container](#) for pgBouncer Pods.

Value type	Example
 subdoc	

**proxy.pgBouncer.sidecars.command**

Command for the [custom sidecar container](#) for pgBouncer Pods.

Value type	Example
 array	<code>["/bin/sh"]</code>

**proxy.pgBouncer.sidecars.args**

Command arguments for the [custom sidecar container](#) for pgBouncer Pods.

Value type	Example
 array	<code>["-c", "while true; do trap 'exit 0' SIGINT SIGTERM SIGQUIT SIGKILL; done;"]</code>

**proxy.pgBouncer.sidecarVolumes.name**

The name of the [volumes](#) [↗](#) to attach to `pgBouncer` Pods for use by [custom sidecar containers](#). Supports secret, configMap, NFS, and other volume types.

Value type	Example
:material-text-string: string	sidecar-secret

**proxy.pgBouncer.sidecarVolumes.secret.secretName**

The name of the [volume Secret](#) [↗](#) to attach to `pgBouncer` Pods for use by [custom sidecar containers](#).

Value type	Example
:material-text-string: string	mysecret

**proxy.pgBouncer.sidecarVolumes.configMap.name**

The name of the configMap object to attach to pgBouncer Pods for use by [custom sidecar containers](#).

Value type	Example
:material-text-string: string	sidecar-config

**proxy.pgBouncer.sidecarVolumes.nfs.server**

The hostname of the NFS server that will provide remote filesystem to the [custom sidecar containers](#) in pgBouncer Pods.

Value type	Example
:material-text-string: string	"nfs-service.storage.svc.cluster.local"

**proxy.pgBouncer.sidecarVolumes.nfs.path**

The path on the NFS server that will be provided as a remote filesystem to the [custom sidecar containers](#) in pgBouncer Pods.

Value type	Example
:material-text-string: string	"nfs-service.storage.svc.cluster.local"

**proxy.pgBouncer.sidecarPVCs**

[PersistentVolumeClaims](#)  that the Operator creates and mounts for [custom sidecar containers](#) in pgBouncer Pods.

You can use PVCs with sidecar containers only when you deploy a new cluster. Updates to running cluster are not supported.

Value type	Example
≡ subdoc	

```

- name: sidecar-volume-claim
 spec:
 resources:
 requests:
 storage: 1Gi
 volumeMode: Filesystem
 accessModes:
 - ReadWriteOnce

```

## Patroni Section

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

Value type	Example
 int	3

### `patroni.syncPeriodSeconds`

How often to perform [liveness/readiness probes](#) for the patroni container (in seconds).

Value type	Example
 int	3

### `patroni.leaderLeaseDurationSeconds`

Initial delay for [liveness/readiness probes](#) for the patroni container (in seconds).

### `patroni.dynamicConfiguration`

Custom PostgreSQL configuration options. Please note that configuration changes are automatically applied to the running instances without validation, so having an invalid config can make the cluster unavailable.

Value type	Example
 subdoc	<pre> postgresql:   parameters:     max_parallel_workers: 2     max_worker_processes: 2     shared_buffers: 1GB     work_mem: 2MB </pre>

## patroni.switchover.enabled

Enables or disables [manual change of the cluster primary instance](#).

Value type	Example
<b>S</b> string	true

## patroni.switchover.targetInstance

The name of the Pod that should be [set as the new primary](#). When not specified, the new primary will be selected randomly.

Value type	Example
<b>S</b> string	

## patroni.createReplicaMethods

Defines available replica creation methods and the order of executing them during a cluster start or reinitialisation. Patroni will stop on the first one that returns 0.

By default, `pg_basebackup` is used to create replicas during a new cluster deployment. After the Operator makes an initial backup, it updates the Patroni ConfigMap assign the `pgBackRest` as the first item in the list. This configuration is not propagated to Patroni itself until you restart the database instance Pods or manually reload Patroni configuration.

In the same way, after you define the replica set methods and apply the configuration, the Operator updates the Patroni ConfigMap. You must manually reload Patroni configuration of every database instance to make Patroni aware of the changes. Read more about setting replica methods in the [Configure create\\_replica\\_methods](#) section.

Value type	Example
<b>S</b> string	- pgbackrest - basebackup

# Custom extensions Section

The `extensions` section in the [deploy/cr.yaml](#)  file contains configuration options to [manage PostgreSQL extensions](#).

**extensions.image**

Image for the custom PostgreSQL extension loader sidecar container.

Value type	Example
 string	<code>docker.io/percona/percona-postgresql-operator:3.0.0</code>

**extensions.imagePullPolicy**

[Policy](#)  for the custom extension sidecar container.

Value type	Example
 string	<code>Always</code>

**extensions.storage.type**

The cloud storage type used for backups. Only `s3` type is currently supported.

Value type	Example
 string	<code>s3</code>

**extensions.storage.bucket**

The [Amazon S3 bucket](#)  name for prepackaged PostgreSQL custom extensions.

Value type	Example
 string	<code>pg-extensions</code>

**extensions.storage.region**

The [AWS region](#)  to use.

Value type	Example
<b>S</b> string	eu-central-1

## extensions.storage.endpoint

The [S3 endpoint](#)  to use.

Value type	Example
<b>S</b> string	s3.eu-central-1.amazonaws.com

## extensions.storage.forcePathStyle

When set to `true`, enforces path-style access method of constructing S3 URLs, where the bucket name appears in the path portion of the URL. Default `false` value means the Operator uses the virtual-hosted-style for accessing S3 storage, where the bucket name is part of the domain name.

Value type	Example
 boolean	false

## extensions.storage.disableSSL

When set to `true`, instructs the Operator to skip TLS verification when accessing the storage. Can be used if your storage endpoint uses self-signed certificates or doesn't support TLS to allow successful downloads.

Value type	Example
 boolean	false

## extensions.storage.secret.name

The [Kubernetes secret](#)  for the custom extensions storage. It should contain `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` keys.

Value type	Example
<b>S</b> string	cluster1-extensions-secret

## extensions.builtin.pg\_stat\_monitor

Enable or disable [pg\\_stat\\_monitor](#) PostgreSQL extension. Disabled by default starting with version 2.9.0.

Value type	Example
<input type="checkbox"/> boolean	false

## extensions.builtin.pg\_stat\_statements

Enable or disable [pg\\_stat\\_statements](#) PostgreSQL extension.

Value type	Example
<input type="checkbox"/> boolean	false

## extensions.builtin.pg\_audit

Enable or disable [PGAudit](#) PostgreSQL extension.

Value type	Example
<input type="checkbox"/> boolean	true

## extensions.builtin.pgvector

Enable or disable [pgvector](#) PostgreSQL extension. **This extension is not compatible with PostgreSQL 12!**

Value type	Example
<input type="checkbox"/> boolean	false

## extensions.builtin.pg\_repack

Enable or disable [pg\\_repack](#) PostgreSQL extension.

Value type	Example

 boolean

false

## extensions.custom.name

Name of the PostgreSQL custom extension.

Value type	Example
 string	pg_cron

## extensions.custom.version

Version of the PostgreSQL custom extension.

Value type	Example
 string	1.6.1

# Backup Resource Options

A Backup resource is a Kubernetes object that tells the Operator how to create and manage your database backups. The `deploy/backup.yaml` file is a template for creating backup resources when you make an on-demand backup. It defines the `PerconaPGBBackup` resource.

This document describes all available options that you can use to customize your backups.

## apiVersion

Specifies the API version of the Custom Resource. `pgv2.percona.com` indicates the group, and `v2` is the version of the API.

## kind

Defines the type of resource being created: `PerconaPGBBackup`.

## metadata

The metadata part of the `deploy/backup.yaml` contains metadata about the resource, such as its name and other attributes. It includes the following keys:

- `name` - The name of the backup resource used to identify it in your deployment. You also use the backup name for the restore operation.

## spec

This subsection includes the configuration of a backup resource.

### pgCluster

Specifies the name of the PostgreSQL cluster to back up.

Value type	Example
<code>s</code> string	<code>cluster1</code>

### repoName

Specifies the name of the `pgBackRest` repository where to save a backup. It must match the name you specified in the `spec.backups.pgBackRest.repos` subsection of the `deploy/cr.yaml` file.

Value type	Example
 string	<code>repo1</code>

**method**

Specifies what method to use for the backup. When undefined, uses `pgBackRest` by default. Supported values are `volumeSnapshot` and `pgBackRest`. See [PVC snapshot support](#) to learn more.

Value type	Example
 string	<code>volumeSnapshot</code>

**options**

You can customize the backup by specifying different [command line options supported by pgBackRest](#) [:octicons-external-link-16:](#).

Value type	Example
 string	<code>--type=full</code>

**containerOptions.env.name**

Specifies the name of a custom environment variable that you pass to backup containers for manual backups.

Value type	Example
 string	<code>MY_ENV</code>

**containerOptions.env.value**

Specifies the value for a custom environment variable that you pass to backup containers for manual backups.

Value type	Example
------------	---------

<b>S</b> string	1000
-----------------	------

**containerOptions.envFrom.secretRef.name**

Name of a Secret, key/values of which are used as environment variables for manual backups.

Value type	Example
<b>S</b> string	backup-env-secret

# Restore Resource Options

A Restore resource is a Kubernetes object that tells the Operator how to restore your database from a specific backup. The `deploy/restore.yaml` file is a template for creating restore resources. It defines the `PerconaPGRestore` resource.

This document describes all available options that you can use to customize a restore.

## apiVersion

Specifies the API version of the Custom Resource. `pgv2.percona.com` indicates the group, and `v2` is the version of the API.

## kind

Defines the type of resource being created: `PerconaPGRestore`.

## metadata

The metadata part of the `deploy/restore.yaml` contains metadata about the resource, such as its name and other attributes. It includes the following keys:

- `name` - The name of the restore resource used to identify it in your deployment. You use this name to track the restore operation status and view information about it.

## spec

This subsection includes the configuration of a restore resource.

### pgCluster

Specifies the name of the PostgreSQL cluster to restore.

Value type	Example
<code>s</code> string	<code>restore1</code>

### repoName

Specifies the name of one of the 4 pgBackRest repositories, already configured in the `backups.pgbackrest.repos` subsection of the `deploy/cr.yaml` file.

Value type	Example
<b>S</b> string	repo1

**volumeSnapshotBackupName**

Specifies the name of a PVC snapshot-based backup to restore from. See [Configure and use PVC snapshots](#) to learn more.

Value type	Example
<b>S</b> string	backup1

**options**

Specify the [command line options supported by pgBackRest](#) [:octicons-external-link-16:](#). For example, to make a point-in-time restore or to restore from a specific backup.

Value type	Example
<b>S</b> string	<code>--type=time</code> <code>--target=YYYY-MM-DD HH:MM:DD +00</code> <code>--set=20240628-074416F (backup name)</code>

To restore from a specific backup, use the `--set` option with the backup label. You can find the backup label in the `status.backupName` field of the `PerconaPGBBackup` resource. For the full restore workflow, see [Restore to the same cluster](#) or [Restore to a new cluster](#).

**containerOptions.env.name**

Specifies the name of a custom environment variable that you pass to backup containers for restore Pods.

Value type	Example
<b>S</b> string	MY_ENV

**containerOptions.env.value**

Specifies the value for a custom environment variable that you pass to backup containers for restore Pods.

Value type	Example
<b>S</b> string	1000

**containerOptions.envFrom.secretRef.name**

Name of a Secret, key/values of which are used as environment variables for restore Pods.

Value type	Example
<b>S</b> string	restore-env-secret

# Secrets Resource options

A Kubernetes Secret is an object used to store sensitive data, such as passwords, tokens, or keys in a secure and manageable way. Unlike ConfigMaps, Secrets are specifically designed to hold confidential information and can be mounted as volumes or injected into environment variables within Pods.

## apiVersion

Specifies the API version of the Custom Resource.

## kind

Defines the type of resource being created: `Secret`.

## metadata.name

Contains the metadata about the resource, such as its name.

## type

Defines the type of data stored within the Secret resource. `Opaque` type signals to Kubernetes and to the Operator that the content of the secret is custom and unstructured.

## stringData

The data that you pass to the Operator within the Secret.

Value type	Example
 string	<code>PMM_SERVER_TOKEN</code>

# Upgrade Resource options

When you upgrade from one major version of Percona Distribution for PostgreSQL to another one (for example, from 17 to 18), the Operator creates a special Upgrade resource, `PerconaPGUpgrade`.

This document describes all available options that you can use to configure the major upgrade flow.

## `apiVersion`

Specifies the API version of the Custom Resource. `pgv2.percona.com` indicates the group, and `v2` is the version of the API.

## `kind`

Defines the type of resource being created: `PerconaPGUpgrade`.

## `metadata`

The metadata part of the `deploy/upgrade.yaml` contains metadata about the upgrade, such as the name and other attributes. It includes the following keys:

- `name` - The name of the upgrade resource used to identify it in your deployment. You also use the upgrade name to track the upgrade progress.

## `spec`

This subsection includes the configuration of an upgrade resource.

### `postgresClusterName`

Specifies the name of the PostgreSQL cluster to upgrade.

Value type	Example
<code>s</code> string	<code>cluster1</code>

## `image`

Specifies the special upgrade image used for the major upgrade.

---

Value type	Example
<b>S</b> string	<code>docker.io/percona/percona-distribution-postgresql-upgrade:18.4-17.10-16.14-15.18-14.23-1</code>

## fromPostgresVersion

Specifies the current major version of PostgreSQL from which you wish to upgrade.

Value type	Example
<b>S</b> integer	<code>17</code>

## toPostgresVersion

Specifies the target major version of PostgreSQL to which you wish to upgrade. You can only upgrade to the next major version (for example, from 16 to 17, or from 17 to 18). Skipping major versions (such as upgrading directly from 15 to 17) is not supported.

Value type	Example
<b>S</b> integer	<code>18</code>

## toPostgresImage

Specifies the image for the target major version of PostgreSQL.

Value type	Example
<b>S</b> string	<code>docker.io/percona/percona-distribution-postgresql:18.4-1</code>

## toPgBouncerImage

Specifies the image for the `pgBouncer` connection pooler that is compatible with the target major version of PostgreSQL.

Value type	Example
<b>S</b> string	<code>docker.io/percona/percona-pgbouncer:1.25.2-1</code>

### toPgBackRestImage

Specifies the image for the `pgBackRest` backup and restore solution that is compatible with the target major version of PostgreSQL.

Value type	Example
<code>S</code> string	<code>docker.io/percona/percona-pgbackrest:2.58.0-2</code>

# Immutable options

Percona Operator for PostgreSQL manages certain options internally to ensure cluster health, backup integrity, and security. You cannot change these options through the Custom Resource or other configuration methods. If you try to modify them, the Operator reconciles the cluster and reverts your changes.

There are some options that you can override and some of them have modification limits.

This document lists all of them:

- [PostgreSQL parameters that you cannot override](#)
- [PostgreSQL parameters that you can override](#)
- [Custom Resource options with modification limits](#).

## PostgreSQL parameters that cannot be overridden

The Operator sets and enforces the following PostgreSQL parameters. You cannot override them via `patroni.dynamicConfiguration.postgresql.parameters` or any other method. The Operator reconciles the Patroni configuration and restores these values.

### TLS and security

Parameter	Value	Purpose
<code>ssl</code>	<code>on</code>	Always set to <code>on</code> for encrypted connections.
<code>ssl_cert_file</code>	<code>/pgconf/tls/tls.crt</code>	Path to the TLS certificate. The Operator manages certificate paths.
<code>ssl_key_file</code>	<code>/pgconf/tls/tls.key</code>	Path to the TLS private key.
<code>ssl_ca_file</code>	<code>/pgconf/tls/ca.crt</code>	Path to the CA certificate.

### Cluster internals

Parameter	Value	Purpose
<code>unix_socket_directories</code>	<code>/tmp/postgres</code>	Socket path for local connections. The Operator uses a fixed path for Pod communication.
<code>log_file_mode</code>	<code>0660</code>	File permissions for log files. The Operator sets this for Pod security context compatibility.

## WAL archiving and recovery (pgBackRest parameters)

Parameter	Value	Purpose
<code>archive_mode</code>	<code>on</code>	Must be <code>on</code> for pgBackRest to archive WAL files. The Operator sets this to enable backups.
<code>archive_command</code>	<code>pgbackrest - stanza=db archive-push "%p"</code>	Command that archives WAL segments to pgBackRest. The Operator configures this for the backup repository.
<code>archive_timeout</code>	<code>60s</code>	Forces a WAL switch after the specified interval. The Operator manages this for backup consistency.
<code>track_commit_timestamp</code>	<code>true</code>	Enables commit timestamps for point-in-time recovery when <a href="#">backups.trackLatestRestorableTime</a> is enabled and the <code>crVersion</code> is 2.8.0 or higher

## Extension parameters

When you enable built-in extensions, the Operator appends or sets the following parameters. You cannot override these values while the extension is enabled.

When `spec.extensions.builtin.pg_stat_statements` is `true`:

Parameter	Value
<code>shared_preload_libraries</code>	Appended with <code>pg_stat_statements</code>
<code>pg_stat_statements.track</code>	<code>all</code>

When `spec.extensions.builtin.pg_stat_monitor` is `true`:

Parameter	Value
<code>shared_preload_libraries</code>	Appended with <code>pg_stat_monitor</code>
<code>pg_stat_monitor.pgsm_query_max_len</code>	<code>2048</code>

When `spec.extensions.builtin.pg_audit` is `true`:

Parameter	Value
<code>shared_preload_libraries</code>	Appended with <code>pgaudit</code>

 **Note**

To view the full list of parameters the Operator sets, run `patronictl show-config` inside a PostgreSQL Pod. See [Manage a database manually](#) for details. Any changes you make via `patronictl edit-config` will be reverted when the Operator reconciles, unless the cluster is in [unmanaged mode](#).

## PostgreSQL parameters that can be overridden

The following parameters are set by the Operator but you can override them via `patroni.dynamicConfiguration.postgresql.parameters`:

Parameter	Default value
<code>wal_level</code>	<code>logical</code>
<code>jit</code>	<code>off</code>
<code>password_encryption</code>	<code>scram-sha-256</code>
<code>archive_timeout</code>	<code>60s</code>
<code>huge_pages</code>	<code>try</code> or <code>off</code> (computed from resource limits)
<code>restore_command</code>	<code>pgbackrest --stanza=db archive-get %f "%p"</code>

## Custom Resource options with modification limits

### `metadata.name`

The cluster name is set at creation time and cannot be changed. Kubernetes Custom Resource names are immutable. To use a different name, create a new cluster and migrate data.

### `users.databases`

You can add databases to a user’s access list, but you cannot revoke access to a database once it has been granted. Removing a database from the list does not revoke existing privileges.

## **users.options**

The `ALTER ROLE` options (such as `SUPERUSER`) are ignored for the `postgres` user. The Operator manages the superuser role.

## **databaseInitSQL**

Initialization SQL runs only at cluster creation time. You cannot add or change init SQL for an existing cluster. The Operator executes the script once during bootstrap.

## **dataSource**

The `dataSource` subsection configures restore-from-backup for a *new* cluster. It applies only during initial cluster creation. You cannot change the data source of an existing cluster.

# **Backup options**

## **Backup encryption**

You cannot change encryption settings after backups are established. To enable encryption or change the encryption key, create a new repository. See [Backup encryption](#).

## **Storage size**

You cannot shrink the size of an existing Persistent Volume Claim (PVC). Kubernetes allows only volume expansion. See [Scale storage](#).

# **Patroni dynamic configuration**

Only the `parameters` and `pg_hba` subsections under `patroni.dynamicConfiguration.postgresql` are applied. All other Patroni dynamic configuration options (such as `use_slots`, `use_pg_rewind`, or `loop_wait`) are ignored. See [Changing PostgreSQL options](#).

# Percona certified images

This page lists Percona's certified Docker images that you can use with Percona Operator for PostgreSQL 3.0.0.

To find images for a specific Operator version, see [Retrieve Percona certified images](#)

## Images released with the Operator version 3.0.0:

Image	Digest
percona/percona-postgresql-operator:3.0.0 (x86_64)	3bcbaec261b2e67c81b3812a4b220c859434b6791c53ee1fb7ecb66bd179de56
percona/percona-postgresql-operator:3.0.0 (ARM64)	5aac965c3d7bc89a23e1292ae47c2ef89d590e77677d793e15f56b11e5d67f7
percona/percona-distribution-postgresql-upgrade:18.4-17.10-16.14-15.18-14.23-1 (x86_64)	31a9612320d3b08cb74d5e98f86b054c4562e4b900140152b28476be62f086f5
percona/percona-distribution-postgresql-upgrade:18.4-17.10-16.14-15.18-14.23-1 (ARM64)	81a831086fc4aaee83af8d86b1422b64ce77e200a8f27627c27282c9af6ad9b8
percona/percona-distribution-postgresql:18.4-1 (x86_64)	ca25cc5e291cf2307d9ec4c29811c6f6d93171f98dd2cb4c69d2036f07517a7d
percona/percona-distribution-postgresql:18.4-1 (ARM64)	fae3368e04b80f4eb887621724074db4db279ca59743aacb47de51b468d514f5
percona/percona-distribution-postgresql:17.10-1 (x86_64)	720bf87ef8cda340f833981a674d5dc71283cb39af3fd4ef97eff4ccf910b87a
percona/percona-distribution-postgresql:17.10-1 (ARM64)	6416424e58765b9434f18905f0338d466cb79bafc42e6a5570030c71a1323eee
percona/percona-distribution-postgresql:16.14-1 (x86_64)	ad84dc4fa537a5ba03c7c1f51f9a1ee959132155b6943d02cb0d614fc57e4271
percona/percona-distribution-postgresql:16.14-1 (ARM64)	f377d7fd5e5e3eae56ef5886aa77691c436269bae5dec706130061286080465
percona/percona-distribution-postgresql:15.18-1 (x86_64)	60728795a6e954d9255526a208e8e9e8e93278d4c1e89004f2999a6ccb196eea

percona/percona-distribution-postgresql:15.18-1 (ARM64)	78b1b7488a897ffaf609c8d1edc6c5a27b76a3c697bbb24db5f2dec3fd66ee34
percona/percona-distribution-postgresql:14.23-1 (x86_64)	d04d3e1c41fd0c9fe438ab04081c555e1edafaed9a33f2979c7b0080045ff5fd
percona/percona-distribution-postgresql:14.23-1 (ARM64)	88925bdfc04d7dec629230c3b0f31daf7f9aed8757b2456a4439475a6b3175f5
percona/percona-distribution-postgresql-with-postgis:18.4-2 (x86_64)	763d037b012a8856ff5ede045631d2f9b9a792b5e8281b756cfb3e017b0395e9
percona/percona-distribution-postgresql-with-postgis:18.4-2 (ARM64)	5f7561ebfc4f7d3e237953b1aae2b41d6bb01d7e7c55e04e2666dd5775cc671a
percona/percona-distribution-postgresql-with-postgis:17.10-2 (x86_64)	fcaccb00ea6937e43c7b4b27a36191faed94dae1cf7ab5ccdc73bb0444984ada
percona/percona-distribution-postgresql-with-postgis:17.10-2 (ARM64)	7e28bb3effd1492057005f8c6c9bd6c3c3aba4ed45f8a01a771adb6e7458537e
percona/percona-distribution-postgresql-with-postgis:16.14-2 (x86_64)	97222e27f34ee5151ed86bf124598004c13ba1c4034eb591641a6eafddd17df1
percona/percona-distribution-postgresql-with-postgis:16.14-2 (ARM64)	251e8fe4fae00ca7562f3ea66620dcfdb1c0bd7989bb501922ac16326da0dc53
percona/percona-distribution-postgresql-with-postgis:15.18-2 (x86_64)	36651cb8711644763676e6ff6b7756647af55402fbae9a1bcf6336392e78bfc5
percona/percona-distribution-postgresql-with-postgis:15.18-2 (ARM64)	a4a557f1490bdb1611d3d1026de227608112e6e2b752e98eab95b660193d53e8
percona/percona-distribution-postgresql-with-postgis:14.23-2 (x86_64)	5d418d47c8442620ddf4063c9d147fda932153a7d9b7e0098ba58d9ecd640b9d
percona/percona-distribution-postgresql-with-postgis:14.23-2	947af4d80224fa0b7031448beec3fb965b5d321e4fa5d742753dba9079667799

(ARM64)	
percona/percona-pgbackrest:2.58.0-2 (x86_64)	0b792f3d0bcfdd7a72c8d74f1c905469486f106db8f9299b5216d23dacf80501
percona/percona-pgbackrest:2.58.0-2 (ARM64)	97361155cdce8642ce47ebe83766a0c7b825930adb6c055d78476756db02fa9f
percona/percona-pgbouncer:1.25.2-1 (x86_64)	25881754364b7a2aaad716bfc77d292a7a0f145c200162cc392874e09bae7918
percona/percona-pgbouncer:1.25.2-1 (ARM64)	8f1bbf2159b6089c235dd512aa397ee53cd84e7e34554238e7981c45bf33767f
percona/pmm-client:3.7.1 (x86_64)	8b98629a469bf6360b14eb3ea121687737870261296f64e4587f6a6723a6845b
percona/pmm-client:3.7.1 (ARM64)	9951a74522a6bd70531457628daca758ffa3363941938539d164e592188e23e3
percona/pmm-client:2.44.1-1 (x86_64)	52a8fb5e8f912eef1ff8a117ea323c401e278908ce29928dafc23fac1db4f1e3
percona/pmm-client:2.44.1-1 (ARM64)	390bfd12f981e8b3890550c4927a3ece071377065e001894458047602c744e3b

For older versions, please refer to the [old releases documentation archive](#) .

# Versions compatibility

Versions of the cluster components and platforms tested with different Operator releases are shown below. Other version combinations may also work but have not been tested.

Cluster components:

Operator	<a href="#">PostgreSQL</a> 	<a href="#">pgBackRest</a> 	<a href="#">pgBouncer</a> 
<a href="#">3.0.0</a>	14 - 18	2.58.0-2	1.25.2-1
<a href="#">2.9.0</a>	14 - 18	2.58.0-1	1.25.1-1
<a href="#">2.8.2</a>	13 - 18	2.57.0	1.25.0
<a href="#">2.8.1</a>	13 - 18	2.57.0	1.25.0
<a href="#">2.8.0</a>	13 - 17	2.56.0	1.24.1
<a href="#">2.7.0</a>	13 - 17	2.55.0	1.24.1
<a href="#">2.6.0</a>	13 - 17	2.54.2 for PostgreSQL 13-16 and 17.4, 2.54.0 for PostgreSQL 17.2	1.24.0 for PostgreSQL 13-16 and 17.2, 1.23.1 for PostgreSQL 17
<a href="#">2.5.1</a>	12 - 16	2.54.2	1.24.0
<a href="#">2.5.0</a>	12 - 16	2.53-1	1.23.1
<a href="#">2.4.1</a>	12 - 16	2.51	1.22.1
<a href="#">2.4.0</a>	12 - 16	2.51	1.22.1
<a href="#">2.3.1</a>	12 - 16	2.48	1.18.0
<a href="#">2.3.0</a>	12 - 16	2.48	1.18.0
<a href="#">2.2.0</a>	12 - 15	2.43	1.18.0
<a href="#">2.1.0</a>	12 - 15	2.43	1.18.0
<a href="#">2.0.0</a>	12 - 14	2.41	1.17.0
<a href="#">1.6.0</a>	12 - 14	2.50	1.22.0

<a href="#">1.5.1</a>	12 - 14	2.47	1.20.0
<a href="#">1.5.0</a>	12 - 14	2.47	1.20.0
<a href="#">1.4.0</a>	12 - 14	2.43	1.18.0
<a href="#">1.3.0</a>	12 - 14	2.38	1.17.0
<a href="#">1.2.0</a>	12 - 14	2.37	1.16.1
<a href="#">1.1.0</a>	12 - 14	2.34	1.16.0 for PostgreSQL 12, 1.16.1 for other versions
<a href="#">1.0.0</a>	12 - 13	2.33	1.13.0

## Platforms:

Operator	<a href="#">GKE</a> 	<a href="#">EKS</a> 	<a href="#">Openshift</a> 	<a href="#">Azure Kubernetes Service (AKS)</a> 	<a href="#">Minikube</a> 
<a href="#">3.0.0</a>	1.33 - 1.35	1.33 - 1.35	4.18 - 4.21	1.33 - 1.35	1.38.1
<a href="#">2.9.0</a>	1.32 - 1.34	1.33 - 1.35	4.17 - 4.21	1.33 - 1.35	1.38.1
<a href="#">2.8.2</a>	1.31 - 1.33	1.31 - 1.34	4.16 - 4.20	1.32 - 1.34	1.37.0
<a href="#">2.8.1</a>	1.31 - 1.33	1.31 - 1.34	4.16 - 4.20	1.32 - 1.34	1.37.0
<a href="#">2.8.0</a>	1.31 - 1.33	1.31 - 1.34	4.16 - 4.20	1.32 - 1.34	1.37.0
<a href="#">2.7.0</a>	1.30 - 1.32	1.30 - 1.33	4.15 - 4.19	1.30 - 1.33	1.36.0
<a href="#">2.6.0</a>	1.29 - 1.31	1.29 - 1.32	4.14 - 4.18	1.29 - 1.31	1.35.0
<a href="#">2.5.1</a>	1.28 - 1.30	1.28 - 1.30	4.13.46 - 4.16.7	1.28 - 1.30	1.33.1
<a href="#">2.5.0</a>	1.28 - 1.30	1.28 - 1.30	4.13.46 - 4.16.7	1.28 - 1.30	1.33.1
<a href="#">2.4.1</a>	1.27 - 1.29	1.27 - 1.30	4.12.59 - 4.15.18	-	1.33.1
<a href="#">2.4.0</a>	1.27 - 1.29	1.27 - 1.30	4.12.59 - 4.15.18	-	1.33.1

<a href="#">2.3.1</a>	1.24 - 1.28	1.24 - 1.28	4.11.55 - 4.14.6	-	1.32
<a href="#">2.3.0</a>	1.24 - 1.28	1.24 - 1.28	4.11.55 - 4.14.6	-	1.32
<a href="#">2.2.0</a>	1.23 - 1.26	1.23 - 1.27	-	-	1.30.1
<a href="#">2.1.0</a>	1.23 - 1.25	1.23 - 1.25	-	-	-
<a href="#">2.0.0</a>	1.22 - 1.25	-	-	-	-
<a href="#">1.6.0</a>	1.26 - 1.29	1.26 - 1.29	4.12.57 - 4.15.13	-	1.33
<a href="#">1.5.1</a>	1.24 - 1.28	1.24 - 1.28	4.11 - 4.14	-	1.32
<a href="#">1.5.0</a>	1.24 - 1.28	1.24 - 1.28	4.11 - 4.14	-	1.32
<a href="#">1.4.0</a>	1.22 - 1.25	1.22 - 1.25	4.10 - 4.12	-	1.28
<a href="#">1.3.0</a>	1.21 - 1.24	1.20 - 1.22	4.7 - 4.10	-	-
<a href="#">1.2.0</a>	1.19 - 1.22	1.19 - 1.21	4.7 - 4.10	-	-
<a href="#">1.1.0</a>	1.19 - 1.22	1.18 - 1.21	4.7 - 4.9	-	-
<a href="#">1.0.0</a>	1.17 - 1.21	1.21	4.6 - 4.8	-	-

# About documentation

This document describes how to deploy, configure, and operate Percona Operator for PostgreSQL on Kubernetes. It also includes reference materials for Custom Resource options, backup and restore settings, and related topics.

## Documentation versions

We recommend that you run the latest version of Percona Operator for PostgreSQL to receive bug fixes and new features. For that reason, the default view of this site reflects the latest documentation.

Starting with **Percona Operator for PostgreSQL 2.7.0**, the documentation site includes a **version switcher** in the header. You can use it to open the documentation that matches a specific Operator release line.

The version switcher displays the **three most recent minor versions** of the documentation (for example, 2.7.0, 2.8.0, and 2.9.0). Patch releases (such as 2.8.1 or 2.8.2), if present, are also listed in the switcher because they belong to their respective minor version. When a new minor version is released (for example, 2.10.0), the oldest minor version and all its associated patch versions (like 2.8.0, 2.8.1, and 2.8.2) are removed from the switcher.

Where a feature was added or behavior changed, the text often notes the Operator version. See the [Release notes](#) for detailed changes in each release.

## Older documentation

Documentation versions that are no longer supported is **not** available in the version switcher. Those releases are published as **PDF** files in the [Percona legacy documentation archive](#) .

## Report an issue or request a change

To report a problem with the documentation, use the feedback widget at the bottom of the page or open a ticket in the [Percona Operator for PostgreSQL project](#)  on Jira.

## How to contribute

Thank you for your interest in improving this documentation. See the [Contributing guide](#)  in the documentation repository for instructions.

# Legal

# Copyright and licensing information

## Documentation licensing

Percona Operator for PostgreSQL documentation is (C)2009-2023 Percona LLC and/or its affiliates and is distributed under the [Creative Commons Attribution 4.0 International License](#) .

# Trademark policy

This [Trademark Policy](#)  is to ensure that users of Percona-branded products or services know that what they receive has really been developed, approved, tested and maintained by Percona. Trademarks help to prevent confusion in the marketplace, by distinguishing one company's or person's products and services from another's.

Percona owns a number of marks, including but not limited to Percona, XtraDB, Percona XtraDB, XtraBackup, Percona XtraBackup, Percona Server, and Percona Live, plus the distinctive visual icons and logos associated with these marks. Both the unregistered and registered marks of Percona are protected.

Use of any Percona trademark in the name, URL, or other identifying characteristic of any product, service, website, or other use is not permitted without Percona's written permission with the following three limited exceptions.

*First*, you may use the appropriate Percona mark when making a nominative fair use reference to a bona fide Percona product.

*Second*, when Percona has released a product under a version of the GNU General Public License ("GPL"), you may use the appropriate Percona mark when distributing a verbatim copy of that product in accordance with the terms and conditions of the GPL.

*Third*, you may use the appropriate Percona mark to refer to a distribution of GPL-released Percona software that has been modified with minor changes for the sole purpose of allowing the software to operate on an operating system or hardware platform for which Percona has not yet released the software, provided that those third party changes do not affect the behavior, functionality, features, design or performance of the software. Users who acquire this Percona-branded software receive substantially exact implementations of the Percona software.

Percona reserves the right to revoke this authorization at any time in its sole discretion. For example, if Percona believes that your modification is beyond the scope of the limited license granted in this Policy or that your use of the Percona mark is detrimental to Percona, Percona will revoke this authorization. Upon revocation, you must immediately cease using the applicable Percona mark. If you do not immediately cease using the Percona mark upon revocation, Percona may take action to protect its rights and interests in the Percona mark. Percona does not grant any license to use any Percona mark for any other modified versions of Percona software; such use will require our prior written permission.

Neither trademark law nor any of the exceptions set forth in this Trademark Policy permit you to truncate, modify or otherwise use any Percona mark as part of your own brand. For example, if XYZ creates a modified version of the Percona Server, XYZ may not brand that modification as "XYZ Percona Server" or "Percona XYZ Server", even if that modification otherwise complies with the third exception noted above.

In all cases, you must comply with applicable law, the underlying license, and this Trademark Policy, as amended from time to time. For instance, any mention of Percona trademarks should include the full trademarked name, with proper spelling and capitalization, along with attribution of ownership to Percona Inc. For example, the full proper name for XtraBackup is Percona XtraBackup. However, it is acceptable to omit the word “Percona” for brevity on the second and subsequent uses, where such omission does not cause confusion.

In the event of doubt as to any of the conditions or exceptions outlined in this Trademark Policy, please contact [trademarks@percona.com](mailto:trademarks@percona.com) for assistance and we will do our very best to be helpful.

# Release Notes

# Percona Operator for PostgreSQL Release Notes

- [Percona Operator for PostgreSQL 3.0.0 \(2026-05-22\)](#)
- [Percona Operator for PostgreSQL 2.9.0 \(2026-04-01\)](#)
- [Percona Operator for PostgreSQL 2.8.2 \(2025-12-25\)](#)
- [Percona Operator for PostgreSQL 2.8.1 \(2025-12-16\)](#)
- [Percona Operator for PostgreSQL 2.8.0 \(2025-11-13\)](#)
- [Percona Operator for PostgreSQL 2.7.0 \(2025-07-18\)](#)
- [Percona Operator for PostgreSQL 2.6.0 \(2025-03-17\)](#)
- [Percona Operator for PostgreSQL 2.5.1 \(2025-03-03\)](#)
- [Percona Operator for PostgreSQL 2.5.0 \(2024-10-08\)](#)
- [Percona Operator for PostgreSQL 2.4.1 \(2024-08-06\)](#)
- [Percona Operator for PostgreSQL 2.4.0 \(2024-06-24\)](#)
- [Percona Operator for PostgreSQL 2.3.1 \(2024-01-23\)](#)
- [Percona Operator for PostgreSQL 2.3.0 \(2023-12-21\)](#)
- [Percona Operator for PostgreSQL 2.2.0 \(2023-06-30\)](#)
- [Percona Operator for PostgreSQL 2.1.0 Tech preview \(2023-05-04\)](#)
- [Percona Operator for PostgreSQL 2.0.0 Tech preview \(2022-12-30\)](#)

# Percona Operator for PostgreSQL 3.0.0 (2026-05-22)

[Get started with the Operator →](#)

We are excited to announce the release of Percona Operator for PostgreSQL 3.0.0 – a new major release that marks a new chapter for the Operator and everyone who relies on it.

As of version 3.0.0, Percona Operator for PostgreSQL becomes a **hard fork of the [Crunchy PGO project](#)**. This marks a significant change in how the Operator is built and maintained going forward.

As a hard fork, Percona Operator for PostgreSQL is now fully independent, empowering the Percona team and community to drive and rapidly deliver features and improvements tailored to user needs. This shift brings long-term sustainability, greater flexibility, and a product that evolves in direct response to community feedback and real-world requirements.

## Release highlights

### Crunchy CRD renaming and seamless resource migration

With this release, all Crunchy CRDs are renamed and moved under a new API group `upstream.pg.v2.percona.com`. All references to Crunchy CRDs in dependent objects, such as Custom Resources, Deployments, Secrets, ConfigMaps, Jobs, and so on, are updated accordingly.

When you update from the earlier Operator versions to 3.0.0 and newer, new API group CRDs are created alongside the legacy CRDs to keep existing workflows uninterrupted. Then the Operator automatically migrates all resources that depend on the legacy CRDs to the new API group. No manual intervention is required.

 **Important**

As part of the API group migration, the common name in the `pgBackRest` client certificate is updated. The Operator automatically updates both the `pgBackRest` configuration and the related certificate Secret. However, you may experience brief disruptions to `pgBackRest` operations while Kubernetes propagates these changes to the containers. This process typically completes within 1–2 minutes.

The migration event is recorded in cluster conditions and logs, so you have full visibility into what happened and when.

This change delivers these benefits:

- Safe coexistence of Percona Operator for PostgreSQL and Crunchy PGO

- Cleaner API boundaries and predictable Operator behavior
- Smooth migration from Crunchy Operator to Percona Operator using native PostgreSQL techniques and without re-architecting your deployments

For how to migrate to Percona Operator from Crunchy PGO, see our [documentaton](#).

## Improved namespace scoping for Operator OLM installations from Community catalogues

When you install the Operator from a Community catalogue through OpenShift Lifecycle Manager (OLM), it now honors the namespace scope as defined by its `OperatorGroup` settings:

- **All namespaces** — The Operator watches all namespaces, as intended for this mode. In earlier releases, it watched only its own installation namespace, which left PostgreSQL custom resources in other namespaces unmanaged.
- **Single namespace** — The Operator watches only the namespaces listed for the `OperatorGroup` (`olm.targetNamespaces`), so a scoped install stays scoped.

With this change, you get the same namespace coverage your subscription mode promises, so clusters behave the way OpenShift administrators expect.

After you upgrade, the Operator may begin reconciling Percona PostgreSQL custom resources that already exist in namespaces it previously ignored. That is most common when several Operators run in one OpenShift cluster, all in all-namespaces mode, and each installation now actually honors that mode.

Before you upgrade, confirm how the Operator is installed by inspecting the `OperatorGroup`: a single entry under `spec.targetNamespaces` means single-namespace mode; an empty value means all-namespaces mode.

To limit which namespaces an Operator reconciles, set `spec.targetNamespaces` explicitly:

```
spec:
 targetNamespaces:
 - "<namespace-1>"
 - "<namespace-2>"
```



If you run more than one Operator in the same cluster, switch each one to single-namespace mode so their reconciliation work does not overlap in ways you did not intend.

## Official Docker image for major version upgrades

Starting with this release, the Operator performs major upgrades using the official Percona Docker image: `percona/percona-distribution-postgresql-upgrade:<available-upgrade-versions>`.

Refer to our [major upgrade documentation](#) for detailed instructions, including the updated image path.

## Deprecation, Change, Rename and Removal

- The support of version 2.7.0 is dropped from CRDs.

## Changelog

### Improvements

- [K8SPG-1007](#) - Renamed upstream Custom Resource Definitions (CRDs) to simplify migration from other PostgreSQL operators. This change allows users to install the Percona Operator for PostgreSQL alongside existing installations without resource conflicts or API group overlapping.
- [K8SPG-1019](#) - Improved the codebase with performance optimizations, tooling, language ergonomics, and security by applying Go 1.26 using `go fix`.
- [K8SPG-1022](#) - Updated the major upgrade documentation to include critical instructions for the `pgaudit` extension. Users are now guided to drop and reinstall the extension during the upgrade process to ensure it functions correctly on the new version.

### Fixed bugs

- [K8SPG-737](#) - Fixed an issue where disk usage statistics were not accessible by PMM `node_exporter` in PostgreSQL pods. The update ensures the data directory is correctly exposed to the monitoring sidecar, enabling full visibility into storage metrics through PMM.
- [K8SPG-999](#) - Fixed a bug where only the first extension was removed when attempting to drop multiple custom extensions. The updated logic ensures all targeted extensions are correctly deleted, maintaining consistency between the Custom Resource specification and the database state. (Thank you Matan Haver for contributing to this issue)
- [K8SPG-1017](#) - Resolved a critical issue where upgrading to version 2.9.0 caused TLS configurations to unexpectedly switch from the internal PKI to cert-manager. This fix prevents unwanted certificate re-issuance and service disruption for clusters that rely on the Operator's internal certificate management.

### Documentation updates

- Created [Migration from Crunchy Postgres Operator to Percona Operator for PostgreSQL tutorials](#)
- Improved the [major upgrade description](#) for users of `pgAudit` extension.
- Database cluster upgrade documentation now uses PostgreSQL 18 as the default example in upgrade commands. If you are upgrading a cluster running PostgreSQL 17 or below, refer to the [certified images list](#)

to determine the correct version and image tags.

## Supported software

This Operator version is developed, tested and based on:

- PostgreSQL 14.23-1, 15.18-1, 16.14-1, 17.10-1, 18.4-1 as the database. Other versions may also work but have not been tested.
- pgBackRest 2.58.0-2 for backup and recovery
- pgBouncer 1.25.2-1 for connection pooling
- Patroni version 4.1.0 for high-availability
- PostGIS version 3.5.5
- PMM Client version 2.44.1-1 and 3.7.1

## Supported platforms

Percona Operators are designed for compatibility with all [CNCF-certified](#)  Kubernetes distributions.

Our release process includes targeted testing and validation on major cloud provider platforms and OpenShift, as detailed below:

- [Google Kubernetes Engine \(GKE\)](#)  1.33 - 1.35
- [Amazon Elastic Container Service for Kubernetes \(EKS\)](#)  1.33 - 1.35
- [OpenShift](#)  4.18 - 4.21
- [Azure Kubernetes Service \(AKS\)](#)  1.33 - 1.35
- [Minikube](#)  1.38.1 with Kubernetes v1.35.1

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

## Percona certified images

Find Percona's certified Docker images that you can use with the Percona Operator for PostgreSQL in the following table.

Image	Digest
percona/percona-postgresql-	3bcbaec261b2e67c81b3812a4b220c859434b6791c53ee1fb7ecb66bd179de56

operator:3.0.0 (x86_64)	
percona/percona-postgresql-operator:3.0.0 (ARM64)	5aacf965c3d7bc89a23e1292ae47c2ef89d590e77677d793e15f56b11e5d67f7
percona/percona-distribution-postgresql-upgrade:18.4-17.10-16.14-15.18-14.23-1 (x86_64)	31a9612320d3b08cb74d5e98f86b054c4562e4b900140152b28476be62f086f5
percona/percona-distribution-postgresql-upgrade:18.4-17.10-16.14-15.18-14.23-1 (ARM64)	81a831086fc4aaee83af8d86b1422b64ce77e200a8f27627c27282c9af6ad9b8
percona/percona-distribution-postgresql:18.4-1 (x86_64)	ca25cc5e291cf2307d9ec4c29811c6f6d93171f98dd2cb4c69d2036f07517a7d
percona/percona-distribution-postgresql:18.4-1 (ARM64)	fae3368e04b80f4eb887621724074db4db279ca59743aacb47de51b468d514f5
percona/percona-distribution-postgresql:17.10-1 (x86_64)	720bf87ef8cda340f833981a674d5dc71283cb39af3fd4ef97eff4ccf910b87a
percona/percona-distribution-postgresql:17.10-1 (ARM64)	6416424e58765b9434f18905f0338d466cb79bafc42e6a5570030c71a1323eee
percona/percona-distribution-postgresql:16.14-1 (x86_64)	ad84dc4fa537a5ba03c7c1f51f9a1ee959132155b6943d02cb0d614fc57e4271
percona/percona-distribution-postgresql:16.14-1 (ARM64)	f377d7fd5e5e3eae56ef5886aa77691c436269bae5decbb706130061286080465
percona/percona-distribution-postgresql:15.18-1 (x86_64)	60728795a6e954d9255526a208e8e9e8e93278d4c1e89004f2999a6ccb196eea
percona/percona-distribution-postgresql:15.18-1 (ARM64)	78b1b7488a897ffaf609c8d1edc6c5a27b76a3c697bbb24db5f2dec3fd66ee34
percona/percona-distribution-postgresql:14.23-1 (x86_64)	d04d3e1c41fd0c9fe438ab04081c555e1edafaed9a33f2979c7b0080045ff5fd
percona/percona-distribution-postgresql:14.23-1 (ARM64)	88925bdfc04d7dec629230c3b0f31daf7f9aed8757b2456a4439475a6b3175f5
percona/percona-distribution-postgresql-with-postgis:18.4-2 (x86_64)	763d037b012a8856ff5ede045631d2f9b9a792b5e8281b756cfb3e017b0395e9

percona/percona-distribution-postgresql-with-postgis:18.4-2 (ARM64)	5f7561ebfc4f7d3e237953b1aae2b41d6bb01d7e7c55e04e2666dd5775cc671a
percona/percona-distribution-postgresql-with-postgis:17.10-2 (x86_64)	fcacccb00ea6937e43c7b4b27a36191faed94dae1cf7ab5ccdc73bb0444984ada
percona/percona-distribution-postgresql-with-postgis:17.10-2 (ARM64)	7e28bb3effd1492057005f8c6c9bd6c3c3aba4ed45f8a01a771adb6e7458537e
percona/percona-distribution-postgresql-with-postgis:16.14-2 (x86_64)	97222e27f34ee5151ed86bf124598004c13ba1c4034eb591641a6eafddd17df1
percona/percona-distribution-postgresql-with-postgis:16.14-2 (ARM64)	251e8fe4fae00ca7562f3ea66620dcfdb1c0bd7989bb501922ac16326da0dc53
percona/percona-distribution-postgresql-with-postgis:15.18-2 (x86_64)	36651cb8711644763676e6ff6b7756647af55402fbae9a1bcf6336392e78bfc5
percona/percona-distribution-postgresql-with-postgis:15.18-2 (ARM64)	a4a557f1490bdb1611d3d1026de227608112e6e2b752e98eab95b660193d53e8
percona/percona-distribution-postgresql-with-postgis:14.23-2 (x86_64)	5d418d47c8442620ddf4063c9d147fda932153a7d9b7e0098ba58d9ecd640b9d
percona/percona-distribution-postgresql-with-postgis:14.23-2 (ARM64)	947af4d80224fa0b7031448beec3fb965b5d321e4fa5d742753dba9079667799
percona/percona-pgbackrest:2.58.0-2 (x86_64)	0b792f3d0bcfdd7a72c8d74f1c905469486f106db8f9299b5216d23dacf80501
percona/percona-pgbackrest:2.58.0-2 (ARM64)	97361155cdce8642ce47ebe83766a0c7b825930adb6c055d78476756db02fa9f
percona/percona-pgbouncer:1.25.2-1 (x86_64)	25881754364b7a2aaad716bfc77d292a7a0f145c200162cc392874e09bae7918
percona/percona-pgbouncer:1.25.2-1 (ARM64)	8f1bbf2159b6089c235dd512aa397ee53cd84e7e34554238e7981c45bf33767f

percona/pmm-client:3.7.1 (x86_64)	8b98629a469bf6360b14eb3ea121687737870261296f64e4587f6a6723a6845b
percona/pmm-client:3.7.1 (ARM64)	9951a74522a6bd70531457628daca758ffa3363941938539d164e592188e23e3
percona/pmm-client:2.44.1-1 (x86_64)	52a8fb5e8f912eef1ff8a117ea323c401e278908ce29928dafc23fac1db4f1e3
percona/pmm-client:2.44.1-1 (ARM64)	390bfd12f981e8b3890550c4927a3ece071377065e001894458047602c744e3b

# Percona Operator for PostgreSQL 2.9.0 (2026-04-01)

[Get started with the Operator →](#)

## What's new at a glance

### Backup and restore

- [Use PVC snapshots for faster backups and restores](#) (tech preview)

### Operations

- [Configure WAL lag detection for standby clusters](#)
- [Mount volumes to sidecar containers](#)
- [Configure leader election for the Operator](#)
- [Troubleshoot with pprof profiling](#)
- [Specify custom DNS suffix for vcluster or custom DNS setups](#)
- [Configure `wal\_level` for replication requirements](#)

### Database and integrations

- [PostgreSQL 18 is now the default version](#)
- [Major upgrade flow is generally available](#)
- [LDAP authentication support](#)
- [Automated TLS certificate management via cert-manager](#)
- [Official PostGIS Docker image](#)

## Release Highlights

This release provides the following features and improvements:

### PostgreSQL 18 is now the default version

Starting with this release PostgreSQL 18 becomes the default and recommended version for new cluster deployments using the Operator. This change enables you to benefit from its latest features, performance improvements, and enhanced security.

## General availability for major upgrades

With this release the major upgrade flow is generally available. This means it has undergone thorough testing and you can use it in production environments.

See our [upgrade guide](#) for the detailed step-by-step instructions.

## Boost backup and restore performance with PVC snapshot support

You can now use PVC snapshots to speed up backups and restores in your PostgreSQL clusters. A PVC snapshot is a point-in-time copy of your data volumes taken directly at the storage layer. Instead of streaming data to cloud storage, the Operator creates fast, storage level snapshots. This is especially beneficial for large data set owners, boosting their backup and restore performance.

Using PVC snapshots, you benefit from:

- **Faster backups** — Snapshots typically complete in seconds or minutes, no matter how large your database is. Traditional full backups can take hours.
- **Faster restores** — Creating a new cluster from a snapshot is significantly quicker than restoring from cloud storage.
- **Lower resource usage** — Snapshots avoid the CPU and network overhead of transferring data to remote storage.
- **Point-in-time recovery support** - By combining PVC snapshots with `pgBackRest` WAL archiving, you ensure data consistency and can make point-in-time recovery of your PostgreSQL cluster.

The Operator currently supports cold (offline) backups, where one replica is briefly taken offline to create the snapshot. We plan to introduce hot backups in future releases, allowing you to run snapshot backups without downtime.

You can find details about the workflow, requirements, and limitations in [the PVC snapshot support documentation](#).

PVC snapshot support is released as a tech preview. We don't recommend using it in production yet, but we encourage you to try it out and share your feedback. To enable the feature, turn on the `BackupSnapshots` feature gate in your Operator Deployment.

Refer to our tutorials for [setting up](#) and [using](#) PVC snapshots.

## Improve replication for standby clusters with WAL lag detection

If your primary cluster has a large volume of WAL files, the standby cluster may not be able to apply them quickly enough. When this happens, the standby can fall behind, experience replication issues, and temporarily miss the most recent data.

To improve the replication, you can now enable replication lag detection for your standby cluster. You set the maximum amount of WAL data the standby is allowed to lag behind. When the amount of WAL files exceeds this threshold, the primary Pod in the standby cluster is marked as `Unready`, the cluster enters the `Initializing` state and the `StandbyLagging` condition is recorded in the cluster status.

This enhancement gives you a clear view of your replication health, speeds up troubleshooting, and helps prevent application downtime during disaster recovery scenarios. To learn more about it, read our [documentation](#).

## Centralize user identity management with LDAP authentication support

LDAP authentication is now available in Percona Operator for PostgreSQL, giving you a straightforward way to centralize database access around your existing corporate identity systems. Instead of verifying user passwords locally, PostgreSQL delegates it to an LDAP server.

You have the flexibility to make a *simple LDAP bind* where the user Distinguished Name is constructed from the prefix and suffix you provide, or make a *bind and search* for a user using a specific attribute. See our [documentation](#) to learn more.

This improvement lets users log in to the database with their existing organizational credentials and not have to remember multiple logins. You benefit from a deployment that is easier to secure, audit and operate at scale.

## Automated TLS certificate lifecycle management via the cert-manager

You can now install and use the `cert-manager` to generate TLS certificates and manage their lifecycle. The Operator automatically detects if the cert-manager is installed in your Kubernetes environment and requests certificates from it when it deploys Percona Distribution for PostgreSQL cluster. You can additionally configure the certificate duration via the Custom Resource to follow your security policies.

With this improvement, you benefit from the following:

- **Automatic renewal** – cert-manager renews certificates before they expire (by default, 30 days before expiry)
- **Configurable validity** – you can set certificate and CA validity durations via Custom Resource options
- **Centralized management** – use cert-manager's tooling and policies for all TLS certificates in the cluster

Follow our [documentation](#) on how to install and use the cert-manager in your deployment.

## Troubleshoot Operator with pprof profiling

`pprof` is Go's built-in profiling tool for CPU and memory analysis. You can use it to investigate Operator performance issues, high CPU usage, or memory leaks when troubleshooting. Set the `PPROF_BIND_ADDRESS` environment variable in the Operator's deployment to an address that the controller should bind to for serving `pprof` metrics. An example value is `127.0.0.1:6060`. Then expose the port from inside the Operator Pod to your local machine so that you can collect CPU or memory profiles.

## Configurable DNS suffix for Operator connections

You can now specify a custom DNS suffix that the Operator uses when it generates service names. This is useful when the Operator runs in a vcluster or in a cluster with a custom DNS configuration. In those setups, the default `cluster.local` suffix can cause incorrect domain name resolution and, as a result, failed connections to external services such as PMM or `pgBackRest`. By setting the `clusterServiceDNSSuffix` in the Custom Resource to your cluster's DNS suffix, the Operator generates hostnames that match your DNS configuration. This ensures the service discoverability and correct communication between workloads.

## Mount volumes to sidecar containers

You can now attach volumes and PersistentVolumeClaims to custom sidecar containers in PostgreSQL instance Pods, pgBouncer Pods, and pgBackRest repo host Pods.

You can either claim storage via Persistent Volume Claim or mount volume Secrets and/or ConfigMaps to sidecars, all via the Custom Resource.

This improvement lets the main application container and sidecars share data without the need for complex API calls for information exchange. You can also update the configuration dynamically, without restarts, which provides additional flexibility in operating sidecars.

For more details and examples, see the [sidecar documentation](#).

## Configurable leader election for Percona Operator for PostgreSQL

You can now tune leader election settings for the Operator Deployment via environment variables. This helps when the Operator hits leader election failures, for example in high-latency or resource-constrained clusters.

- Use the `PGO_CONTROLLER_LEASE_DURATION`, `PGO_CONTROLLER_RENEW_DEADLINE`, `PGO_CONTROLLER_RETRY_PERIOD` environment variables to adjust timing for lease acquisition and renewal.
- Use the `PGO_CONTROLLER_LEADER_ELECTION_ENABLED` environment variable to turn on or off leader election for single-replica deployments
- Use the `PGO_CONTROLLER_LEASE_NAME` environment variable to use a custom Lease resource for a leader lock.

Learn more about available environment variables in our [documentation](#).

## Ability to configure wal\_level

The `wal_level` setting defines how much information to write to Write Ahead Log (WAL) in PostgreSQL. The Operator sets the default `wal_level` to `logical`. This works well if you use logical replication. However, for clusters that only need physical replication or no replication at all, it can add unnecessary overhead: more WAL data, extra I/O, and higher CPU usage.

You can now choose the right level for your use case. Set `wal_level` in the Custom Resource when you create a cluster or update it later.

```
spec:
 patroni:
 dynamicConfiguration:
 postgresql:
 parameters:
 wal_level: replica
```



If you change the `wal_level` value on an existing cluster, your PostgreSQL Pods will be restarted.

This ability to configure `wal_level` gives you control over WAL behavior and lets you avoid extra overhead when you don't need logical replication.

## Official Docker image for PostGIS

The Operator now uses the official Percona Docker images for PostGIS, with the image path `percona/percona-distribution-postgresql-with-postgis:<postgresql-version>`.

Note that there is no official Docker image for PostGIS for PostgreSQL 13 as this major version [entered end-of-life](#).

Pay attention to the image path when you [upgrade the database](#).

## Deprecation, Change, Rename and Removal

### Deprecated support for PMM2

The Operator deprecates support for PMM2 as this version entered the end-of-life stage. PMM2 remains available so you can still monitor the health of your database using this version. However, we encourage you to plan migration to PMM3 to enjoy all features and fixes that this version provides. See the [PMM upgrade documentation](#) [↗](#) for steps.

The support for PMM2 will be dropped in the Operator in two releases.

## PostgreSQL 13 is end-of-life

PostgreSQL 13 has now end-of-life and is not included in this Operator release.

## Operators in Red Hat Marketplace catalog are no longer maintained

Red Hat Marketplace was discontinued in April 2025. Percona Operator for PostgreSQL will remain listed in the Marketplace catalog, but it won't be updated beyond OpenShift 4.22.

If you use the Operator from Red Hat Marketplace, switch to the Certified Operator Catalog for future updates and support.

### `pg_stat_monitor` is disabled by default

Starting with this release `pg_stat_monitor` is disabled by default when you deploy a new cluster. If you wish to keep using this extension after upgrading to this version, re-enable it in the Custom resource explicitly.

## CRD changes

- The description of the `.spec.majorVersion` option has been updated to include PostgreSQL 18.
- New fields:
- `tls.properties.caValidityDuration`
- `tls.properties.pgBackRestCertValidityDuration`
- `tls.properties.certValidityDuration`

## Changelog

### New features

- [K8SPG-374](#) - Added the WAL lag detection for standby clusters based on user defined thresholds
- [K8SPG-771](#) - Added support of PVC snapshots, which enable storage-level snapshot capabilities and allow for faster backups and restores of large databases.

### Improvements

- [K8SPG-552](#) - Added the ability to automatically create TLS certificates via the cert-manager and configurable certificate duration via the Custom Resource
- [K8SPG-758](#) - Added support for pprof profiling tool. You can set up port-forwarding to the Operator controller to perform CPU and memory profiling for easier performance investigation

- [K8SPG-779](#) - Added the ability to configure `wal_level` according to replication requirements to avoid extra I/O overhead.
- [K8SPG-822](#) - Improved e2e tests by adding `pg_repack` built-in extension to the testsuite
- [K8SPG-837](#) - Updated the base image to RHEL 10 for PostgreSQL Operator
- [K8SPG-840](#) - The logic for determining the latest restorable time has been improved to provide more accurate point-in-time recovery options.
- [K8SPG-864](#) - Added the ability to add custom volumes, ConfigMaps, and Secrets to sidecar containers, allowing for deeper customization without code changes.
- [K8SPG-873](#) - Added `.env` and `.envFrom` fields to backup/restore resources. These new fields allow users to inject custom environment variables into backup and restore jobs for better configuration flexibility.
- [K8SPG-904](#) - Users can now initialize a new cluster from an existing data source even if automated backups are currently disabled in the configuration.
- [K8SPG-908](#) - The Operator has transitioned to using official PostGIS images to ensure better compatibility and standard support.
- [K8SPG-915](#) - Added the ability to tune leader election parameters for the Operator to prevent unnecessary failovers in unstable network environments.
- [K8SPG-956](#) - Optimized the `archive_command` to reduce CPU usage by applying resource manager filters in `pg_waldump` during WAL processing.

## Bugs fixed

- [K8SPG-647](#) - Resolved a race condition that occasionally prevented secondary nodes from reaching a ready state following a major version upgrade or restore.
- [K8SPG-665](#) - Resolved the issue with excessive log messages about superusers being exposed through PGBouncer by changing the default value for the `exposeSuperusers` option to `false`.
- [K8SPG-694](#) - Fixed an issue where the Operator incorrectly identified the internal cluster domain when running inside a vcluster environment by adding the ability to specify custom DNS suffix.
- [K8SPG-740](#) - Fixed the issue with the Operator failing to clean up outdated backups during the minor upgrade by checking the repo-host pod status before attempting backup cleanup. The cleanup is skipped until the repo-host pod is ready.
- [K8SPG-821](#) - Enhanced automated testing for database updates to ensure stability across different OS base images and PostgreSQL versions.
- [K8SPG-901](#) - Fixed a crash that occurred when the proxy configuration block was missing from the Custom Resource file.
- [K8SPG-903](#) - Improved error reporting for upstream controllers to include stacktraces, making support cases and troubleshooting significantly faster.

- [K8SPG-907](#) - Fixed a bug where point-in-time recovery operations would fail with a panic during internal JSON data processing.
- [K8SPG-923](#) - Improved documentation on how to fine-tune backup performance via asynchronous replication
- [K8SPG-933](#) - Fixed an issue that prevented the successful creation of standby clusters when enabled in the Custom Resource.
- [K8SPG-938](#) - Fixed a synchronization issue where cluster status conditions were not correctly reflecting updates from the underlying controllers.
- [K8SPG-939](#) - Fixed an issue with Patroni not considering custom labels. Now all Patroni-managed objects correctly inherit and apply all Custom Resource labels for better tracking and reporting
- [K8SPG-943](#) - Resolved a nil pointer dereference that could cause the Operator to crash during cluster creation under specific conditions.
- [K8SPG-957](#) - Fixed a crash in Amazon EKS environments where the Operator would fail if VolumeSnapshot APIs were not pre-installed.
- [K8SPG-982](#) - Fixed the issue with Pods being killed with out-of-memory error when pg\_stat\_monitor is enabled by default by disabling it by default.
- [K8SPG-983](#) - Fixed the issue with PMM QAN exporter not respecting pg\_stat\_statements set as the query source by adding the value mapping to correctly parse the query-source value.

## Documentation improvements

- Updated OpenShift documentation to better show available installation options
- Updated PostGIS documentation now features simplified deployment steps and clarifies the how to connect to PostgreSQL and enable the extension.
- Improved documentation about cluster-wide setup with a clear explanation of the WATCH\_NAMESPACE environment variable
- Added documentation of available environment variables
- Added documentation about immutable options in the Operator
- Added instructions how to override resource names when installing the Operator via Helm

## Supported software

This Operator version is developed, tested and based on:

- PostgreSQL 14.22-1, 15.17-1, 16.13-1, 17.9-1, 18.3-1 as the database. Other versions may also work but have not been tested.
- pgBackRest 2.58.0-1 for backup and recovery

- pgBouncer 1.25.1-1 for connection pooling
- Patroni version 4.1.0 for high-availability
- PostGIS version 3.5.5
- PMM Client versions 2.44.1-1 and 3.6.0

## Supported platforms

Percona Operators are designed for compatibility with all [CNCF-certified](#)  Kubernetes distributions.

Our release process includes targeted testing and validation on major cloud provider platforms and OpenShift, as detailed below:

- [Google Kubernetes Engine \(GKE\)](#)  1.32 - 1.34
- [Amazon Elastic Container Service for Kubernetes \(EKS\)](#)  1.33 - 1.35
- [OpenShift](#)  4.17 - 4.21
- [Azure Kubernetes Service \(AKS\)](#)  1.33 - 1.35
- [Minikube](#)  1.38.1 with Kubernetes v1.35.1

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

## Percona certified images

Find Percona's certified Docker images that you can use with the Percona Operator for PostgreSQL in the following table.

Image	Digest
percona/percona-postgresql-operator:2.9.0	1990ab3568a25fbe4fbb85bc0a524c72458b6d4419f2d96a6ef61874da83ea96
percona/percona-postgresql-operator:2.9.0 (ARM64)	470f0a141973c91474b9337c92773aa467a2145ff5ad74fc4731a11beb446083
percona/percona-distribution-postgresql:18.3-1 (x86_64)	f7f2af7cd155162fcffbd2a09e28918795db4ca1d1119c60b61a0d7c2f146ee7
percona/percona-distribution-postgresql:18.3-1 (ARM64)	97531c11ffaf33f677f7e8062783e9ce13d1cd2618cb88c56d6387bf92720dcb

percona/percona-distribution-postgresql:17.9-1 (x86_64)	deca076dc5b837d9f7712de4ed007e019900d09c629fcba53d35b7ec47f4b308
percona/percona-distribution-postgresql:17.9-1 (ARM64)	921279b3b85c6595ba3cbd67856c456f8f4b711b270f8473ff5acbd82781a43d
percona/percona-distribution-postgresql:16.13-1 (x86_64)	36ae43818f7e1414332549ef5361ed3874e3f3ad2c430e07dcea7552d8c8b362
percona/percona-distribution-postgresql:16.13-1 (ARM64)	b4771737ee43d576437fa301bd0f15f7477b0058f3d8d58f5c7e8349412c0c94
percona/percona-distribution-postgresql:15.17-1 (x86_64)	0b3faf1329c018f155aa9eb182f99b4a008f8f25b549f4cef98581002ca57d01
percona/percona-distribution-postgresql:15.17-1 (ARM64)	64c9c06271eb24552fba4f766992b9228cfd99fbaafc93313ebba10d91bcda25
percona/percona-distribution-postgresql:14.22-1 (x86_64)	2e854233f37877edf5a1920de5749a96eb0d81022b2270e00446889a6a3d6140
percona/percona-distribution-postgresql:14.22-1 (ARM64)	93034300269680d1f024be3f500590f39a3eae91868ec6ec32c5689d76b2e999
percona/percona-distribution-postgresql-with-postgis:18.3-1 (x86_64)	a2cdf2fa7b76d6f02fb249ce56efda51db476d695ae1b5e276ab89d99ab1d0a5
percona/percona-distribution-postgresql-with-postgis:18.3-1 (ARM64)	5058d7a615bf647ff629598e1feae0a9ffcde14dce70f35814d631d90bf57e93
percona/percona-distribution-postgresql-with-postgis:17.9-1 (x86_64)	964a1a3116db7cd7fed0452376f43b07a9e3b45bf1ba2377307837745d285101
percona/percona-distribution-postgresql-with-postgis:17.9-1 (ARM64)	ecbabb4b2296fd1964b46cbdb71dae9d21157ac59f64ff776aff7d39aac66d1c
percona/percona-distribution-postgresql-with-postgis:16.13-1 (x86_64)	30a64dc854caf5770906e17fc4e32e4a7de3f545478c94719a8c6d7ab41b88d3
percona/percona-distribution-postgresql-with-postgis:16.13-1 (ARM64)	6936f74de4e6f5206e5367581bcfad49860d1572a30e9387a0479d988065778

percona/percona-distribution-postgresql-with-postgis:15.17-1 (x86_64)	1d9a94124bddd3939e8ad0beb6ef3ffd8db0858ba97ef1822e08f6c891ae2719
percona/percona-distribution-postgresql-with-postgis:15.17-1 (ARM64)	f2b21836b0e0d995b8187e0c770e31f9113bf6770f51d5eae92aa608b88d4d72
percona/percona-distribution-postgresql-with-postgis:14.22-1 (x86_64)	46cf19acc553c84d643201c4ecd83a69a9d98c7432596a6907fadb093a0cd4df
percona/percona-distribution-postgresql-with-postgis:14.22-1 (ARM64)	9342ff19350446e83041e7775f8f134e0d464233fe3076e0a172a50dfc41b66c
percona/percona-pgbouncer:1.25.1-1 (x86_64)	183f1cad97f7064745aedba96c169287ce54f2945073c28797a65bb9dc64cf8d
percona/percona-pgbouncer:1.25.1-1 (ARM64)	6f4d7e68678a040516f729dc9a9fdf0a1e20ed3f5e5328a7b4fba23b4084c72a
percona/percona-pgbackrest:2.58.0-1 (x86_64)	56542b3615f742a1ff4dec4eff7f53e87228085e50ebb66e3468d943e5a0f02e
percona/percona-pgbackrest:2.58.0-1 (ARM64)	d0b86dc1b725483999828cbf44b5dbad9616767da70cc1b33d2fef2841cd3f05
percona/pmm-client:2.44.1-1 (x86_64)	52a8fb5e8f912eef1ff8a117ea323c401e278908ce29928dafc23fac1db4f1e3
percona/pmm-client:2.44.1-1 (ARM64)	390bfd12f981e8b3890550c4927a3ece071377065e001894458047602c744e3b
percona/pmm-client:3.6.0 (x86_64)	174fa4675d3ea4d95fd7b45d11f2bcc98b98b703662e6b2614dfe886a7187b23
percona/pmm-client:3.6.0 (ARM64)	435a9af2083adb68ddab6a97e6d02bd6d31c54562e919ebc09618e886d58d1ae

# Percona Operator for PostgreSQL 2.8.2 (2025-12-25)

[Get started with the Operator →](#)

## Release Highlights

This release provides PostgreSQL images for the updated releases of Percona Distribution for PostgreSQL rebuilt with disabled debug assertions.

You can find the latest available images in the [images list](#).

## Supported software

This version of the Operator is developed, tested and based on:

- PostgreSQL 18.1-3, 17.7-2, 16.11-2, 15.15-2, 14.20-2, 13.23-2 as the database. Other versions may also work but have not been tested.
- pgBackRest 2.57.0-1 for backup and recovery
- pgBouncer 1.25.0-1 for connection pooling
- Patroni version 4.1.0 for high availability
- PMM Client versions 2.44.1-1 and 3.5.0
- PostGIS:
  - version 3.5.4 for PostgreSQL 18
  - version 3.3.8 for PostgreSQL 17, 16, 15, 14, and 13

## Supported platforms

Percona Operators are designed for compatibility with all [CNCF-certified](#)  Kubernetes distributions.

Our release process includes targeted testing and validation on major cloud provider platforms and OpenShift, as detailed below for the current Operator version:

- [Google Kubernetes Engine \(GKE\)](#)  1.31 - 1.33
- [Amazon Elastic Kubernetes Service \(EKS\)](#)  1.31 - 1.34
- [Azure Kubernetes Service \(AKS\)](#)  1.32 - 1.34

- [OpenShift](#)  4.16 - 4.20
- [Minikube](#)  1.37.0 with Kubernetes v1.34.0

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

## Percona certified images

Find Percona’s certified Docker images that you can use with the Percona Operator for PostgreSQL in the following table.

Image	Digest
percona/percona-postgresql-operator:2.8.2 (x86_64)	018b0063352fff83d7850d732c80ba6a938c425ac2d9ac7e9a0a270361ff3fc0
percona/percona-postgresql-operator:2.8.2 (ARM64)	ce7e6f612d4cef4ef86f06521549f3e3c4e1fe8ecf794feff6f3205667863792
percona/percona-distribution-postgresql:18.1-3	940859b7c45d1217ba852e8c5e5500832daf61c6914d5af33808251cb23f0102
percona/percona-distribution-postgresql:17.7-2	7beb6a0d9fd4c4e8125c5cb43c48e8b189a4dca7b4f1ec1e433b49956d4203d6
percona/percona-distribution-postgresql:16.11-2	80882a55997c58b7a4dd5defc6482d99dc31c11fbd206f788e540a74ffab4823
percona/percona-distribution-postgresql:15.15-2	100ef9920d70d0ff53a0466f9b077ba3f1828c31b330a08a6e6e11802b870178
percona/percona-distribution-postgresql:14.20-2	011bae00abeb266352aa6c8e127396b0a3fb5877f0112f5291172624b5556120
percona/percona-distribution-postgresql:13.23-2	39ef40f72704e8e782ed43fb5a5072ee557fd410b85b9d2177b332ce39b888f5
percona/percona-postgresql-operator:2.8.2-ppg18.1-postgres-gis3.5.4	a9c58611e1660a1377a579370c0684a5bc12a37df69031bdcf7dac5332d016ad
percona/percona-postgresql-operator:2.8.2-ppg17.7-postgres-gis3.3.8	64246fa0fe2213f2321bdd565181578fd84e8d745b044553b60d45c41dbf3e5c

percona/percona-postgresql-operator:2.8.2-ppg16.11-postgres-gis3.3.8	d4467fe931e6403538a1f9831c7c97f84177027bfb226a9a7e021026506bb4d7
percona/percona-postgresql-operator:2.8.2-ppg15.15-postgres-gis3.3.8	d9680938ae31bc0c2d1fd89cda306b1bdb9b4aef424d706896460de71a2311e2
percona/percona-postgresql-operator:2.8.2-ppg14.20-postgres-gis3.3.8	71947a799ea4e957bf6a9dd6a4ff055037a335d9d71186452b64e05d1ce68c38
percona/percona-postgresql-operator:2.8.2-ppg13.23-postgres-gis3.3.8	9b75f33ee9d7e95a0d9ea9e619533878d92814183a0ed9ad59bc69813b11a84d
percona/percona-pgbouncer:1.25.0-1 (x86_64)	bf2f325cc733b96dc360c2386c8931ed9e3513f55cb425e59033e1e56737134f
percona/percona-pgbouncer:1.25.0-1 (ARM64)	902feac78cf98fbd6a7aece1761371dd1a43faaed88b63be0e0d54dd524b8286
percona/percona-pgbackrest:2.57.0-1 (x86_64)	2bf7265f84210671bc5c0928cc772202c0e6054d426eb6ecf86279d69e831b96
percona/percona-pgbackrest:2.57.0-1 (ARM64)	59245b25fd5d0c1a2540b465e846b69f4c91b6cd183c3bfd96aa856d3e7ffbf3
percona/pmm-client:2.44.1-1	52a8fb5e8f912eef1ff8a117ea323c401e278908ce29928dafc23fac1db4f1e3
percona/pmm-client:3.5.0 (x86_64)	352aee74f25b3c1c4cd9dff1f378a0c3940b315e551d170c09953bf168531e4a
percona/pmm-client:3.5.0 (ARM64)	cbbb074d51d90a5f2d6f1d98a05024f6de2ffdcdb5acab632324cea4349a820bd

# Percona Operator for PostgreSQL 2.8.1 (2025-12-16)

[Get started with the Operator →](#)

## Release Highlights

This release provides the following features and improvements:

### PostgreSQL 18 support

You can now deploy PostgreSQL 18 on Kubernetes with the Operator. This latest major version of PostgreSQL delivers major improvements in performance, usability, and security, empowering you to make large-scale, mission-critical deployments more reliable and efficient.

Key improvements of PostgreSQL 18 are:

- Asynchronous I/O (AIO) boosts throughput and reduces latency for sequential scans, vacuums, and other heavy operations. This means faster queries and smoother performance under load.
- Queries can now use multicolumn B-tree indexes more effectively. Users benefit from faster lookups without needing redundant indexes.
- Upgrades made via `pg_upgrade` no longer discard optimizer statistics. This reduces downtime and ensures consistent query performance after migrations.
- You can now enforce PRIMARY KEY, UNIQUE, and FOREIGN KEY constraints over ranges of time. This is especially valuable for applications managing time-series or historical data.
- Generated columns are now computed at read time by default. This reduces storage overhead and makes schema design more flexible.

Read more about PostgreSQL 18 in:

- [Percona Blog: Planning Ahead for PostgreSQL 18: What Matters for Your Organization](#) 
- [PostgreSQL 18.1 release notes](#) 

Find PostgreSQL 18 images in the list of [Percona-certified images](#).

## Supported software

The Operator 2.8.1 is developed, tested and based on:

- PostgreSQL 18.1-1, 17.7-1, 16.11-1, 15.15-1, 14.20-1, 13.23-1 as the database. Other versions may also work but have not been tested.
- pgBouncer 1.25.0-1 for connection pooling
- Patroni version 4.1.0 for high-availability
- PMM Client 3.5.0
- PostGIS:
  - version 3.5.4 for PostgreSQL 18,
  - version 3.3.8 for PostgreSQL 17, 16, 15, 14, and 13



#### PostgreSQL RPMs rebuilt to disable debug assertions

The Percona Server for PostgreSQL (PSP) and Percona Distribution for PostgreSQL (PPG) RPM packages for **PostgreSQL versions 13 through 18 released as part of the Q4 quarterly release** were built with debug assertions enabled (`--enable-cassert`).

If you installed or updated PostgreSQL RPMs within the last four months, you may suffer performance degradation for the following versions: 18.1, 17.6, 17.7, 16.10, 16.11, 15.14, 15.15, 14.19, 14.20, 13.22, 13.23.

These packages have been rebuilt, and all users running RPM-based installations of the affected releases are **strongly advised** to update to the latest available packages.

To verify, run `pg_config --configure`. If the output includes `--enable-cassert`, then your installation is affected.

**We do not recommend using the affected PostgreSQL versions listed above in production.** We are working on a fix to fully address this issue. Find more information in [Percona Distribution for PostgreSQL 18.1.1](#) release notes.

## Supported platforms

Percona Operators are designed for compatibility with all [CNCF-certified](#)  Kubernetes distributions.

Our release process includes targeted testing and validation on major cloud provider platforms and OpenShift, as detailed below for Operator version 2.8.1:

- [Google Kubernetes Engine \(GKE\)](#)  1.31 - 1.33
- [Amazon Elastic Kubernetes Service \(EKS\)](#)  1.31 - 1.34
- [Azure Kubernetes Service \(AKS\)](#)  1.32 - 1.34
- [OpenShift](#)  4.16 - 4.20
- [Minikube](#)  1.37.0 with Kubernetes v1.34.0

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

# Percona certified images

Find Percona’s certified Docker images that you can use with the Percona Operator for PostgreSQL in the following table.

Image	Digest
percona/percona-postgresql-operator:2.8.1 (x86_64)	018b0063352fff83d7850d732c80ba6a938c425ac2d9ac7e9a0a270361ff3fc0
percona/percona-postgresql-operator:2.8.1 (ARM64)	ce7e6f612d4cef4ef86f06521549f3e3c4e1fe8ecf794feff6f3205667863792
percona/percona-distribution-postgresql:18.1-1	23522ee9c1abda0b9cbb40c4b414328dafc10596506731954a5754e8b6994e76
percona/percona-distribution-postgresql:17.7-1	c4eec3a4fc8a5d7ba6a631a19f7387f5e34ca9ddcc8ba34bdc6709159be2c3ac
percona/percona-distribution-postgresql:16.11-1	4cd7092284bf323893c75349a5f6c0f4948d8e602fa213210e3efca28cfc2f1d
percona/percona-distribution-postgresql:15.15-1	9ace25f15a319ec741ab32502d4818874a981c38dbb22625e8f2f67bf42bb558
percona/percona-distribution-postgresql:14.20-1	e926d10167ba73da8e2c75218256cd99c68ca9072fd1b9b1ed4b00822a165ab0
percona/percona-distribution-postgresql:13.23-1	03d8d76d844495d07d1eae3fe5767a1c8130fba0a29c6c1353e872601380f9da
percona/percona-postgresql-operator:2.8.1-ppg18.1-postgres-gis3.5.4	706c0aa7c45692d108fbb172cbcd6bf990ee95ff5a77c5e5f79638e45dccd0a9
percona/percona-postgresql-operator:2.8.1-ppg17.7-postgres-gis3.3.8	6ceb2c24a279ddc6914ec762cf96f4b89cbd4869aa8a875c4a55fc82685cb3e7
percona/percona-postgresql-operator:2.8.1-ppg16.11-postgres-gis3.3.8	dca87ac8ddf79ed600f8b7243d7a351ec058a0a65aedbd360ac77fcb061e441c
percona/percona-postgresql-operator:2.8.1-ppg15.15-postgres-gis3.3.8	c82dc9203cbe24b5dbf3cc540d5c13484c543be8b225ad9fdec92fc4c8b6f2ff

percona/percona-postgresql-operator:2.8.1-ppg14.20-postgres-gis3.3.8	14ec320d529d7c941b23da58d4d0da6249e6eba3b6d10b5d11b36e4d2aaeb929
percona/percona-postgresql-operator:2.8.1-ppg13.23-postgres-gis3.3.8	9df1dd41a1369d672b6f8a9653dd358f2cf85f363ceff1ca3389641094494b07
percona/percona-pgbouncer:1.25.0-1 (x86_64)	bf2f325cc733b96dc360c2386c8931ed9e3513f55cb425e59033e1e56737134f
percona/percona-pgbouncer:1.25.0-1 (ARM64)	902feac78cf98fbd6a7aece1761371dd1a43faaed88b63be0e0d54dd524b8286
percona/percona-pgbackrest:2.57.0-1 (x86_64)	2bf7265f84210671bc5c0928cc772202c0e6054d426eb6ecf86279d69e831b96
percona/percona-pgbackrest:2.57.0-1 (ARM64)	59245b25fd5d0c1a2540b465e846b69f4c91b6cd183c3bfd96aa856d3e7ffbf3
percona/pmm-client:2.44.1-1	52a8fb5e8f912eef1ff8a117ea323c401e278908ce29928dafc23fac1db4f1e3
percona/pmm-client:3.5.0 (x86_64)	352aee74f25b3c1c4cd9dff1f378a0c3940b315e551d170c09953bf168531e4a
percona/pmm-client:3.5.0 (ARM64)	cbbb074d51d90a5f2d6f1d98a05024f6de2ffdcdb5acab632324cea4349a820bd

# Percona Operator for PostgreSQL 2.8.0 (2025-11-13)

[Get started with the Operator →](#)

## Release Highlights

This release provides the following features and improvements:

### Custom PostgreSQL user credentials are now fully respected by the Operator

You no longer have to define full login and connection information within a Secret to have the Operator use it. Now you can set only the password. The Operator generates the missing details that it needs automatically using the values from the Custom Resource. Also, if you name your Secret in the format that the Operator expects such as `<clusterName>-pguser-<userName>` – the Operator will automatically detect and use it without needing an explicit reference in the Custom Resource.

However, if you choose a custom name for the Secret, you must still reference it explicitly in the Custom Resource under the `users[ ].secretName` field. This ensures the Operator can locate and apply it correctly.

Read more about managing user passwords in the [documentation](#).

This enhancement makes the management of user credentials more straightforward.

### Ability to use huge pages

PostgreSQL can now use huge pages if they are enabled for your Kubernetes cluster. Instruct the Operator to use huge pages when deploying a PostgreSQL cluster with this configuration:

```
spec:
 instances:
 - name: instance1
 resources:
 limits:
 hugepages-2Mi: 16Mi
 memory: 4Gi
```



This improvement leads to a more efficient memory utilization and improved performance. Learn more about huge pages and their use in the [Huge pages](#) chapter.

### Expanded S3 compatibility for custom extensions

Some S3-compatible services (like MinIO or Ceph) require path-style access instead of virtual-hosted style. Or they may use self-signed certificates or not support TLS.

To address these issues, you can now fine-tune the Operator with these new options:

- `forcePathStyle` enforces path-style access instead of virtual-hosted style
- `disableSSL` disables SSL verification to allow successful downloads.

```
extensions:
 image: docker.io/perconalab/percona-postgresql-operator:main
 storage:

 forcePathStyle: false
 disableSSL: false
```



This improvement enables you to use a wider range of S3-compatible storage services with the Operator for storing custom extensions.

## Changed Patroni version management

The Operator no longer runs a temporary Pod `cluster_name-patroni-version-check` to identify the Patroni version during cluster initialization.

Instead, it uses the `patronictl` CLI tool to connect to a database Pod and detect the Patroni version. The detected version is recorded in the `pgv2.percona.com/patroni-version` annotation on the cluster resource and is added to the resource status.

The Operator standardizes on Patroni 4 as the only supported version and no longer honors Patroni version overrides via the `pgv2.percona.com/custom-patroni-version` annotation.

However, if your Custom Resource is still at version 2.7.0, the Operator 2.8.0 will continue to run a temporary Pod to check Patroni version and use Patroni 3 if specified via the annotation for backward compatibility. But after you upgrade the Custom Resource to version 2.8.0, the `pgv2.percona.com/custom-patroni-version` annotation is ignored, and Patroni 4 is always used.

This change eliminates ambiguity and ensures your cluster is deployed with a modern high-availability implementation.

## Official Docker image for PostgreSQL images

The Operator now uses the official Percona Docker images for Percona Distribution for PostgreSQL, with the image path `percona/percona-distribution-postgresql:<postgresql-version>`.

Because of this transition, the Operator is compatible with and supports only the following specific PostgreSQL versions:

- Percona Distribution for PostgreSQL 17.6
- Percona Distribution for PostgreSQL 16.10
- Percona Distribution for PostgreSQL 15.14
- Percona Distribution for PostgreSQL 14.19
- Percona Distribution for PostgreSQL 13.22

Attempting to use the Operator with other PostgreSQL versions or custom images is not supported.

## Changelog

### New features

- [K8SPG-730](#) - Added the `status.observedGeneration` field to the Custom Resource Definition to improve observability and ensure the controller successfully reconciled the latest changes to the cluster.
- [K8SPG-752](#) - Allowed setting `loadBalancerClass` service type and use a custom implementation of a load balancer rather than the cloud provider default one.
- [K8SPG-768](#) - Introduced a mechanism to prevent excessive logging caused by continuous pod annotation updates for suggested volume sizing. The Operator now skips updating the Pod annotation with the suggested volume size unless the auto-growable disk feature is explicitly configured. This significantly reduces redundant logs and unnecessary load on both the Kubernetes API and the logging pipeline.
- [K8SPG-832](#) - Users can now specify custom sidecar containers for the `repo-host` Pod, enabling seamless integration with external tools, storage systems, or observability agents. This enhances flexibility in backup workflows without modifying the Operator's core logic.
- [K8SPG-833](#) - Added the ability to define custom environment variables across all components. This enables tighter integration with external systems, secrets, or runtime configurations.

### Improvements

- [K8SPG-460](#) - The Operator now correctly enables and used Huge pages functionality if they are enabled on the OS level.
- [K8SPG-570](#) - The Operator now correctly respects custom user passwords defined in secrets when creating new users, and automatically adds any missing credentials.
- [K8SPG-611](#) - The operator now uses official Percona PostgreSQL docker images, which are compatible only with specific latest PostgreSQL versions.
- [K8SPG-624](#), [K8SPG-728](#) - Added the ability to configure the Operator to use path-style access to S3 storage or skip TLS verification to ensure broader compatibility with S3 storage services.
- [K8SPG-718](#) - Improved Patroni observability by sending Patroni metrics to PMM.

- [K8SPG-748](#) - The PerconaPGCluster status now provides more comprehensive details, including persistent volume resizing and pgBackRest backup conditions.
- [K8SPG-757](#): The Percona PostgreSQL Operator now successfully deploys in environments where `readOnlyRootFilesystem` is enforced.
- [K8SPG-874](#) - Improved logging to no longer contain backup-related information when backups are disabled.
- [K8SPG-882](#) - The operator no longer deploys a temporary Patroni version check pod, as it now detects the version directly from running database instances.

## Fixed bugs

- [K8SPG-724](#) - Fixed the issue with upgrading custom extension versions. The Operator now correctly uninstalls old versions and installs new ones automatically.
- [K8SPG-777](#) - Custom Resource `crVersion` is now automatically assigned if not explicitly defined.
- [K8SPG-778](#) - Backup restores no longer fail due to empty repository name errors during the finalization process.
- [K8SPG-781](#) - Error messages for primary pod issues now reveal the specific underlying problem instead of a generic message.
- [K8SPG-803](#) - Outdated backups are now correctly cleaned up, even when pgBackRest debug logging is enabled.
- [K8SPG-826](#) - Fixed the issue with cluster monitoring on OpenShift by using the correct folder for PMM3 .
- [K8SPG-835](#) - Improved affinity behavior for `patroni-version-check` pod
- [K8SPG-844](#) - Fixed the issue with the Operator overriding user configuration with archive commands when the latest restorable time tracking disabled by fully respecting user configuration.
- [K8SPG-869](#) - A backup repository is no longer required when configuring a cluster with disabled backups.
- [K8SPG-872](#) - Updated DNS records used in certificates to no longer include a trailing period to comply with updated validation standards.
- [K8SPG-876](#): Fixed an issue where PostgreSQL clusters remained in an “Initialized” state after restoring a backup from S3 storage.
- [K8SPG-879](#) - Clusters can now be created successfully on Kubernetes version 1.34.
- [K8SPG-883](#): Patroni version information is now displayed in the `status.patroni.version` field instead of `status.patroniVersion`.
- [K8SPG-884](#) - Clusters deployed with PostgreSQL 13 now correctly support the `pg_stat_statements` extension.

## Documentation improvements

- Refined the Upgrade guide structure, moving instructions for updating built-in extensions under the Database upgrade section for better clarity.
- Improved documentation for generating custom TLS certificates used by your cluster and added steps how to safely renew or replace your certificate authorities and secrets.
- Enhanced the Adding custom extensions documentation by including a sample configuration for a custom extension, illustrating the overall workflow as a practical reference.
- Improved the Upgrade document with the steps to change collation version if there is a collation mismatch.
- PostGIS image documentation now accurately reflects the available versions.

## Deprecation, Change, Rename and Removal

- New repository for `postgresql` image.

Now the Operator uses the official Percona Docker images for PostgreSQL. Pay attention to the new image path when you [upgrade the Operator and the database](#). Check the [Percona certified images](#) for exact image names.

- The `patroni.patroniVersion` field in Custom Resource Definition is deprecated and will be removed in future releases. Starting with version 2.8.0, the Operator uses the `patroni.version` field in Custom Resource Definition to populate Patroni version.

```
patroni:
 status:
 systemIdentifier: "7569216022115639385"
 version: 4.0.6
```



Adjust your applications or scripts accordingly to this change if they rely on Patroni version information.

- New fields in the Custom Resource Definition:
- `status.observedGeneration` to track whether the controller has successfully applied the latest changes to the custom resource
- `patroni` subsection contains these fields for Patroni state:
  - `patroni.version`
  - `patroni.systemIdentifier`
  - `patroni.switchover`
  - `patroni.switchoverTimeline`
- `pgBackRest` subsection contains these fields to track the status of backup repository and backup jobs:
  - `pgBackRest.manualBackup`

- `pgBackRest.repoHost`
- `pgBackRest.repos`

## Supported software

The Operator 2.8.0 is developed, tested and based on:

- PostgreSQL 13.22-1, 14.19-1, 15.14-1, 16.10-1, 17.6-1 as the database. Other versions may also work but have not been tested.
- pgBouncer 1.24.1-1 for connection pooling
- Patroni version 4.0.6 for high-availability
- PostGIS version 3.3.8

## Supported platforms

Percona Operators are designed for compatibility with all [CNCF-certified](#)  Kubernetes distributions.

Our release process includes targeted testing and validation on major cloud provider platforms and OpenShift, as detailed below for Operator version 2.8.0:

- [Google Kubernetes Engine \(GKE\)](#)  1.31 - 1.33
- [Amazon Elastic Container Service for Kubernetes \(EKS\)](#)  1.31 - 1.34
- [OpenShift](#)  4.16 - 4.20
- [Azure Kubernetes Service \(AKS\)](#)  1.32 - 1.34
- [Minikube](#)  1.37.0 with Kubernetes v1.34.0

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

## Percona certified images

Find Percona's certified Docker images that you can use with the Percona Operator for PostgreSQL in the following table.

Image	Digest
percona/percona-postgresql-operator:2.8.0 (x86_64)	e34a185e1b295ff627facd3cfbdfc31f32bab714eac550de5e6da00abd9053e2

percona/percona-postgresql-operator:2.8.0 (ARM64)	18445bd761ac3f77901f0e9eddd79b295d28b779779a29bb2d69eb51c32e3815
percona/percona-distribution-postgresql:17.6-1	ce91a339a511d91d9f1946708d7ca326572796b642d2a022a1d52a2adff8a08b
percona/percona-distribution-postgresql:16.10-1	ba1aede456a938f85c9614bb70c50ce264ec68b659917a3a0847112e42bc9259
percona/percona-distribution-postgresql:15.14-1	8280ba2410235e8266761004a2f180fe3999203e69772eb822959cf1849bd967
percona/percona-distribution-postgresql:14.19-1	052e7fd765b790ad2321675e8f2b273fe705512afda5004c4d2a4da78489bfb0
percona/percona-distribution-postgresql:13.22-1	2989dcc4919c8381dc970b2286dadec45c8a53067b48f2bcfff7c7c042b3a654
percona/percona-postgresql-operator:2.8.0-ppg17.6-postgres-gis3.3.8	3322136e6e54214255601586be8f610677fe51a494d3a002cabfacd233258fab
percona/percona-postgresql-operator:2.8.0-ppg16.10-postgres-gis3.3.8	2d5f9ac5a84129e81b9ab8df25abce712223c358847afed3637fb7063a3e4a8f
percona/percona-postgresql-operator:2.8.0-ppg15.14-postgres-gis3.3.8	e7f5fda3cf7d2fab028b3fb70636c9b3b11fe6b89a9f31970d2792bd8f48d8ca
percona/percona-postgresql-operator:2.8.0-ppg14.19-postgres-gis3.3.8	3f69534a0df0b608d68808df04618222e4a20c1d1567462e4482f07b86349806
percona/percona-postgresql-operator:2.8.0-ppg13.22-postgres-gis3.3.8	cd5a2a1057708fac5dda28d0ce47006cdbf865e6ffef1ac2df74065b95258fd3
percona/percona-pgbouncer:1.24.1-1 (x86_64)	39bd093ec83ca4eaeb93b43b286d39daae4cc4b3b32956d627d242d30a5ad6f5
percona/percona-pgbouncer:1.24.1-1 (ARM64)	84d34843180d852182790ce6175f1407a0438b3a415a21741212701706808ac0
percona/percona-pgbackrest:2.56.0-1 (x86_64)	387469090be8e009e17cc07903aa28aa1c748ce1cc385bd69e88de3762657877

percona/percona-pgbackrest:2.56.0-1 (ARM64)	29290808bdeb17a49c90f2ce3ccc75f3bfab43e96e160320baf16cb557d165ee
percona/pmm-client:2.44.1-1	52a8fb5e8f912eef1ff8a117ea323c401e278908ce29928dafc23fac1db4f1e3
percona/pmm-client:3.4.1 (x86_64)	1c59d7188f8404e0294f4bfb3d2c3600107f808a023668a170a6b8036c56619b
percona/pmm-client:3.4.1 (ARM64)	2d23ba3e6f0ae88201be15272c5038d7c38f382ad8222cd93f094b5a20b854a5

# Percona Operator for PostgreSQL 2.7.0 (2025-07-18)

[Get started with the Operator →](#)

## Release Highlights

This release provides the following features and improvements:

### PMM3 support

The Operator is natively integrated with PMM 3, enabling you to monitor the health and performance of your Percona Distribution for PostgreSQL deployment and at the same time enjoy enhanced performance, new features, and improved security that PMM 3 provides.

Note that the Operator supports both PMM2 and PMM3. The decision on what PMM version is used depends on the authentication method you provide in the Operator configuration: PMM2 uses API keys while PMM3 uses service account token. If the Operator configuration contains both authentication methods with non-empty values, PMM3 takes the priority.

To use PMM, ensure that the PMM client image is compatible with the PMM Server version. Check [Percona certified images](#) for the correct client image.

For how to configure monitoring with PMM see the [documentation](#).

### Improved monitoring for clusters in multi-region or multi-namespace deployments in PMM

Now you can define a custom name for your clusters deployed in different data centers. This name helps Percona Management and Monitoring (PMM) Server to correctly recognize clusters as connected and monitor them as one deployment. Similarly, PMM Server identifies clusters deployed with the same names in different namespaces as separate ones and correctly displays performance metrics for you on dashboards.

To assign a custom name, define this configuration in the Custom Resource manifest for your cluster:

```
spec:
 pmm:
 customClusterName: postgresql-cluster
```



### Added labels to identify the version of the Operator

Custom Resource Definition (CRD) is compatible with the last three Operator versions. To know which Operator version is attached to it, we've added labels to all Custom Resource Definitions. The labels help you identify the current Operator version and decide if you need to update the CRD. To view the labels, run:

```
kubectl get crd perconaapgcclusters.pgv2.percona.com --show-labels.
```

## Grant users access to a public schema

Starting with PostgreSQL 15, a non-database owner cannot access the default `public` schema and cannot create tables in it. We have improved this behavior so that the Operator creates a user and a schema with the name matching the username for all databases listed for this user. This custom schema is set by default enabling you to work in the database right away.

You can explicitly grant access to a `public` schema for a non-superuser setting the `grantPublicSchemaAccess` option to `true`. This grants the user permission to create tables and update in the `public` schema of every database they own. If multiple users are granted access to the `public` schema in the same database, each user can only access the tables they have created themselves. If you want one user to access tables created by another user in the `public` schema, the owner of those tables must connect to PostgreSQL and explicitly grant the necessary privileges to the other user.

Superusers have access to the `public` schema for their databases by default.

## Improved troubleshooting with the ability to override Patroni configuration

You can now override Patroni configuration for the whole cluster as well as for an individual Pod. This gives you more control over the database and simplifies troubleshooting.

Also, you can redefine what method the Operator will use when it creates replica instances in your PostgreSQL cluster. For example, to force the Operator to use `pgbasebackup`, edit the `deploy/cr.yaml` manifest:

```
patroni:
 createReplicaMethods:
 - basebackup
 - pgbackrest
```



Note that after you apply this configuration, the Operator updates the Patroni ConfigMap, but it doesn't apply this configuration to Patroni. You must manually reload the Patroni configuration of every database instance for it to come into force.

Read more about these troubleshooting methods in the [documentation](#).

## Changelog

## New features

- [K8SPG-615](#) - Introduced a custom delay on the entrypoint of the backup pod. The backup process waits the defined time before connecting to the API server
- [K8SPG-708](#), [K8SPG-663](#) - Added the sleep-forever feature to keep a database container running.
- [K8SPG-712](#) - Added the ability to control every parameter supported by Patroni configuration.
- [K8SPG-725](#) - Added the ability to configure resources for the repo-host container
- [K8SPG-719](#) - Added support for PMM v3

## Improvements

- [K8SPG-571](#) - Added the ability to access to a public schema for a non-superuser custom user for every database listed for them.
- [K8SPG-612](#) - Updated the `pgBouncer` image to use the official `percona-pgbouncer` Docker image
- [K8SPG-613](#) - Updated the `pgBackRest` image to use the official `percona-pgbackrest` Docker image
- [K8SPG-654](#) - Added the ability to add custom parameters in the Custom Resource and pass them to PMM.
- [K8SPG-675](#) - Added the ability to define resource requests for CPU and memory
- [K8SPG-704](#) - Added the ability to configure `create_replica_methods` for Patroni
- [K8SPG-710](#) - Added the ability to disable backups
- [K8SPG-715](#) - Improved custom-extensions e2e test by adding `pgvector`
- [K8SPG-726](#) - Added ability to define security context for all sidecar containers
- [K8SPG-729](#) - Added Labels for Custom Resource Definitions (CRD) to identify the Operator version attached to them
- [K8SPG-732](#) - Enhanced readability of `pgbackrest debug logs` by printing log messages on separate lines
- [K8SPG-738](#) - Added startup log to the Operator Pod to print commit hash, branch and build time
- [K8SPG-743](#) - Disabled client-side rate limiting in the Kubernetes Go client to avoid throttling errors when managing multiple clusters with a single operator. This change leverages Kubernetes' server-side Priority and Fairness mechanisms introduced in v1.20 and later. (Thank you Joshua Sierles for contributing to this issue)
- [K8SPG-744](#) - Improved Contributing guide with the steps how to build the Operator for development purposes
- [K8SPG-717](#), [K8SPG-750](#) - Added the ability to define a custom cluster name for PMM for filtering
- [K8SPG-753](#) - Added the ability to enable `pg_stat_statements` instead of `pg_stat_monitor`
- [K8SPG-761](#) - Added the ability to add concurrent reconciliation workers
- [K8SPG-828](#) - Added registry name to images due to Openshift 4.19 changes

## Bugs Fixed

- [K8SPG-532](#) - Improved log visibility to include logs about missing data source to INFO logs
- [K8SPG-574](#) - Added `pg_repack` to the list of built-in extensions in the Custom Resource
- [K8SPG-661](#) - Added documentation about replica reinitialization in the Operator
- [K8SPG-677](#) - Made the `imagePullPolicy` in `pg-db` Helm chart configurable
- [K8SPG-680](#) - Prevent scheduled backups to start until the volume expansion is completed with success.
- [K8SPG-698](#) - Fixed the issue with `pgbackrest` service account not being created and reconciliation failing by creating the StatefulSet for this service account first
- [K8SPG-703](#) - Fixed the issue with the backup Pod being stuck in a running state due to running jobs being deleted because of the TTL expiration by adding an internal finalizer to keep the job running until it finishes
- [K8SPG-722](#) - Documented the replica reinitialization behavior.
- [K8SPG-772](#) - Fixed the issue with WAL watcher panicking if some backups have no `CompletedAt` status field by using `CreationTimestamp` as fallback.
- [K8SPG-782](#) - Fixed the issue with crashing WALWatcher by assigning Patroni version to status when Patroni label is configured through the Custom resource option
- [K8SPG-785](#) - Fixed PMM template in Helm chart (Thank you user Nik for reporting this issue)
- [K8SPG-792](#) - Add the ability to configure and use images defined in environment variables when starting a cluster (Thank you Jakub Jaruszewski for reporting this issue)
- [K8SPG-799](#) - Fixed the issue with the cluster being blocked due to inability to pull the image for the Patroni Version Detector Pod if `imagePullSecrets` is configured. The issue is fixed by respecting the configuration for the patroni version check pod. (Thank you Baptiste Balmon for reporting this issue)
- [K8SPG-804](#) - Fixed an issue where outdated cluster state could cause a duplicate backup job to be created, blocking new backups. The issue was fixed by ensuring `reconcileManualBackup` fetches the latest postgrescluster state.
- [K8SPG-812](#) - Fixed image in PerconaPGUpgrade example

## Deprecation, Change, Rename and Removal

- New repositories for `pgBouncer` and `pgBackRest`

Now the Operator uses the official Percona Docker images for `pgBouncer` and `pgBackRest` components. Pay attention to the new image repositories when you [upgrade the Operator and the database](#). Check the [Percona certified images](#) for exact image names.

- Changes in image pulling on OpenShift

Starting with OpenShift version 4.19, the way Operator images are pulled has changed. Now the registry name must be specified for image paths to ensure the images are pulled successfully from DockerHub.

All Custom Resource manifests now include the registry name in image paths. This enables you to successfully install the Operator using the default manifests from Git repositories. If you upgrade the Operator and the database cluster via the command line interface, add the `docker.io` registry name to image paths for all components in the format:

```
"docker.io/percona/percona-postgresql-operator:2.7.0-ppg18.4-1-postgres"
```



Follow our [upgrade documentation](#) for update guidelines.

## Supported software

This version of the Operator is developed, tested and based on:

- PostgreSQL 13.21, 14.18, 15.13, 16.9, 17.5.2 as the database. Other versions may also work but have not been tested.
- pgBouncer 1.24.1 for connection pooling
- Patroni version 4.0.5 for high-availability
- PostGIS version 3.3.8

## Supported platforms

Percona Operators are designed for compatibility with all [CNCF-certified](#)  Kubernetes distributions.

Our release process includes targeted testing and validation on major cloud provider platforms and OpenShift, as detailed below for Operator version 2.7.0:

- [Google Kubernetes Engine \(GKE\)](#)  1.30 - 1.32
- [Amazon Elastic Container Service for Kubernetes \(EKS\)](#)  1.30 - 1.33
- [OpenShift](#)  4.15 - 4.19
- [Azure Kubernetes Service \(AKS\)](#)  1.30 - 1.33
- [Minikube](#)  1.36.0 with Kubernetes v1.33.1

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

## Percona certified images

Find Percona's certified Docker images that you can use with the Percona Operator for PostgreSQL in the following table.

Image	Digest
percona/percona-postgresql-operator:2.7.0 (x86_64)	96e4e3d7e4bcbcd4880adebc5ccb958c0f4385298f0becdef2eb14b81fab407e5
percona/percona-postgresql-operator:2.7.0 (ARM64)	055da3233a7765f22b318c97223909c20ecbbc9f34c6a8f7845d04ade51364ca
percona/percona-postgresql-operator:2.7.0-ppg17.5.2-postgres	cfb99ebeec00ab6efb4fca4a8da2b8c3b489dd792bd2f907848197ba09bc9553
percona/percona-postgresql-operator:2.7.0-ppg16.9-postgres	0787088575b4e4fec368acbcf4dd7aea49620ec4524451e3b44ed424fb0eeebb
percona/percona-postgresql-operator:2.7.0-ppg15.13-postgres	c93f52ea1d6ec955a368c4539b843a9c57ee4a5acc907f0dfb59ae3018560d1b
percona/percona-postgresql-operator:2.7.0-ppg14.18-postgres	a24059edd9864f7dc9607c3e2964844f417718a5b9f471ceb98c0a0d774a4bca
percona/percona-postgresql-operator:2.7.0-ppg13.21-postgres	2c9a05399b34cfe79698bdaab66db8fdaece0db7b1fa34441124cccdbe375255
percona/percona-postgresql-operator:2.7.0-ppg17.5.2-postgres-gis3.3.8	860ccc180c1ac6be3c34c354d6ba9148b00330e183ba5913954e34d49c95d22f
percona/percona-postgresql-operator:2.7.0-ppg16.9-postgres-gis3.3.8	ca50f560bc7b3e18ec3360dc1a6b8c860e0346472af051cb0d2aec2a7a45d8b3
percona/percona-postgresql-operator:2.7.0-ppg15.13-postgres-gis3.3.8	bb6707fd12ea430708e2eb22f6c7dadf3ab4258fcfd31e86f1f78c66ba211742
percona/percona-postgresql-operator:2.7.0-ppg14.18-postgres-gis3.3.8	c3b55d1394d8f0a476cea29340442313c9c08dcd8c83f31ccfc66afdbde42488
percona/percona-postgresql-operator:2.7.0-ppg13.21-postgres-gis3.3.8	3df44c1089563b42198ef929e27b9797ef2b04d92736293952163fa7541c0068

percona/percona-pgbackrest:1.24.1	451431afa3cd288ecda92b6446bec8833fbf376fbd1b7c7e314fc42f3355255f
percona/percona-pgbackrest:1.24.1 (ARM64)	479aa893e55c5afe8b97852c90d7551dc55d3fc526773a5a7d992876bbf54cb0
percona/percona-pgbackrest:2.55.0	b0d2defbc7a07cf395b1fa6c6e13d9d3267c3a2d3c52362ac440db26ea4a4bad
percona/percona-pgbackrest:2.55.0 (ARM64)	bc15d058e7820499bf67ccec2fe51c583fe67a6e3ed55ec28adf3e252828924a
percona/pmm-client:2.44.1	8b2eaddffd626f02a2d5318ffebc0c277fe8457da6083b8cfcada9b6e6168616
percona/pmm-client:2.44.1 (ARM64)	337fec4afdb3f6daf2caa2b341b9fe41d0418a0e4ec76980c7f29be9d08b5ea
percona/pmm-client:3.3.0	0f4ef6a814946f83ef1ed26cf3526ff606fc7815007f84995492d3e4eaa15a0e
percona/pmm-client:3.3.0 (ARM64)	c03aa678d26faf783c3598b3a139a8f3154e5bf1bc9f5a3c9abf0533922f79d6

# Percona Operator for PostgreSQL 2.6.0 (2025-03-17)

## Installation

## Release Highlights

This release provides the following features and improvements:

### Backup improvements

This release implemented several improvements to the backup/restore process:

- A new [delete-backups](#) finalizer was implemented to automatically remove all backups when deleting the cluster. This finalizer is off by default. It's experimental and, therefore, is not recommended for production environments.
- Backup logic was improved and now allows retrying a failed backup in the same backup Pod for a specified number of times before deleting this Pod and creating a new one. This should be beneficial in case of short connectivity issues or timeouts. This behavior is controlled by the new [backups.pgbackrest.jobs.backoffLimit](#) and [backups.pgbackrest.jobs.restartPolicy](#) Custom Resource options.
- You can now [overwrite](#) the default restore command for `pgBackRest` via the [patroni.dynamicConfiguration](#) Custom Resource option. Particularly, this allows to control and filter files restored to `pg_wal` directory without editing these files in the backup repository storage.

### PostgreSQL 17 support

PostgreSQL 17 is now supported by the Operator in addition to versions 13 - 16. The appropriate images are now included in the [list of Percona-certified images](#). See these blogposts for details about the latest PostgreSQL 17 features with the added security and functionality improvements:

- [Encrypt PostgreSQL Data at Rest on Kubernetes](#) [↗](#) by Ege Gunes
- [The Powerful Features Released in PostgreSQL 17 Beta 2](#) [↗](#) by Shivam Dhapatkar
- [PostgreSQL 17: Two Small Improvements That Will Have a Major Impact](#) [↗](#) by David Stokes.

PostgreSQL 17 is currently not recommended for production environments due to the [known limitation](#).

*Update from April 1, 2025:* We have added PostgreSQL 17.4 image and database cluster components based on this image. It is now production ready and we recommend updating the database cluster from PostgreSQL 17.2 to 17.4. Check the [upgrade instructions](#) for steps

## pgvector is added to the PostgreSQL image

To support you with your AI journey, we've added the `pgvector` extension to the PostgreSQL images shipped with our Operator. Now, you can easily use Percona Distribution for PostgreSQL as a vector database by simply enabling it in your [Custom Resource options](#). No more [custom extension installations](#)  needed.

## New features

- [K8SPG-628](#): The custom `restore_command` [can be now passed](#) to pgBackRest via the [patroni.dynamicConfiguration](#) Custom Resource option
- [K8SPG-619](#): New `backups.pgbackrest.jobs.backoffLimit` and `backups.pgbackrest.jobs.restartPolicy` Custom Resource options allow to retry backup in the backup Pod for a specified number of times before abandoning the Pod and creating the new one
- [K8SPG-648](#): PostgreSQL 17 is now supported by the Operator

## Improvements

- [K8SPG-487](#): New `spec.metadata.labels` and `spec.metadata.annotations` Custom Resource options allow setting labels and annotation globally for all Kubernetes objects created by the Operator
- [K8SPG-554](#): New `tlsOnly` Custom Resource option allows the user to enforce TLS connections for the database cluster
- [K8SPG-586](#): The new experimental `finalizers.delete-backups` finalizer (off by default) removes all backups of the cluster at cluster deletion event
- [K8SPG-634](#): The new `autoCreateUserSchema` Custom Resource option enhances the declarative user management by automatically creating per-user schemas
- [K8SPG-652](#): Improve security and meet compliance requirements by using PostgreSQL images built based on Red Hat Universal Base Image (UBI) 9 instead of UBI 8
- [K8SPG-692](#): Patroni versions 4.x are now supported by the Operator in addition to versions 3.x
- [K8SPG-699](#): The `pgvector` extension is now included within the PostgreSQL image used by the Operator
- [K8SPG-701](#): The `extensions.image` Custom Resource option is now optional, and can be omitted for builtin PostgreSQL extensions
- [K8SPG-702](#): A retry logic was implemented to fix intermittent Pod exec failures caused by timeouts (Thanks to dcaputo-harmoni for contribution)

- [K8SPG-711](#): The new [README.md](#)  explains how to build your own images for the PostgreSQL cluster components used by the Operator

## Bugs Fixed

- [K8SPG-594](#): Fix a bug where extension was still appearing in pg\_extension table after being removed from Custom Resource and physically deleted by the Operator
- [K8SPG-637](#): Fix a bug where restore was failing with “waiting for another restore to finish” if the pg-restore object of a previous unfinished restore was manually deleted
- [K8SPG-638](#): Fix a bug that caused flooding the logs with no completed backups found error at cluster initialization.
- [K8SPG-645](#): Fix a bug where creating sidecar containers for pgBouncer did not work
- [K8SPG-681](#): Fixed a bug where the “Last Recoverable Time” information field was missing from the output of the `kubectl get pg-backup` command due to misdetection cases
- [K8SPG-713](#): Fix a bug where The cluster not found errors were appearing in the Operator logs on cluster deletion

## Deprecation, Change, Rename and Removal

- The new versions of Percona distribution for PostgreSQL used by the Operator come with Patroni 4.x, which introduces breaking changes compared to previously used 3.x versions.

To maintain backward compatibility, the Operator detects the Patroni version used in the image. It is also possible to disable this auto-detection feature by manually setting the Patroni version via the [following annotation set in the metadata part](../annotations.md#customizing-patroni-version-for-the-operator-version-260–270 of the Custom Resource:

```
pgv2.percona.com/custom-patroni-version: "4"
```



- PostgreSQL 12 is no longer supported by the Operator 2.6.0 and newer versions.

## Known limitations

- PostgreSQL 17.2 image and images for other database cluster components based on PostgreSQL 17 contain the known [CVE-2025-1094](#)  - a vulnerability in the libpq PostgreSQL client library, which makes images used by the Operator vulnerable to SQL injection within the PostgreSQL interactive terminal due to the lack of neutralizing quoting. Images for PostgreSQL 17 will be available soon, while images for other PostgreSQL versions have already been fixed.

- PostgreSQL 17.4 image includes the fix for [CVE-2025-1094](#), which closed a vulnerability in the `libpq` PostgreSQL client library but introduced a regression related to string handling for non-null terminated strings. The error would be visible based on how a PostgreSQL client implemented this behavior.

## Supported platforms

The Operator 2.6.0 is developed, tested and based on:

- PostgreSQL 13.20, 14.17, 15.12, 16.8, 17.2 and 17.4 as the database. Other versions may also work but have not been tested.
- pgBouncer for connection pooling:
  - version 1.23.1 - for PostgreSQL 17.2
  - version 1.24.0 - for PostgreSQL 13.20, 14.17, 15.12, 16.8, 17.4
- Patroni for high-availability:
  - version 4.0.5 - for PostgreSQL 17.4
  - version 4.0.3 - for PostgreSQL 17.2
  - version 4.0.4 - for PostgreSQL 13.20, 14.17, 15.12, 16.8

Percona Operators are designed for compatibility with all [CNCF-certified](#) Kubernetes distributions.

Our release process includes targeted testing and validation on major cloud provider platforms and OpenShift, as detailed below for Operator version 2.6.0:

- [Google Kubernetes Engine \(GKE\)](#) 1.29 - 1.31
- [Amazon Elastic Container Service for Kubernetes \(EKS\)](#) 1.29 - 1.32
- [OpenShift](#) 4.14 - 4.18
- [Azure Kubernetes Service \(AKS\)](#) 1.29 - 1.31
- [Minikube](#) 1.35.0 with Kubernetes 1.32.0

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

# Percona Operator for PostgreSQL 2.5.1

- **Date**

March 03, 2025

- **Installation**

[Installing Percona Operator for PostgreSQL](#)

## Release highlights

This release fixes the [CVE-2025-1094](#) , vulnerability in the libpq PostgreSQL client library, which made images used by the Operator vulnerable to SQL injection within the PostgreSQL interactive terminal due to the lack of neutralizing quoting. For now, the fix includes the image of PostgreSQL 16.8 and other database cluster images based on PostgreSQL 16.8. Fixed images for other PostgreSQL versions are to follow in the upcoming days.

*Update from March 04, 2025:* images of PostgreSQL 15.12 and other database cluster components based on PostgreSQL 15.12 were added.

*Update from March 06, 2025:* images of PostgreSQL 14.17 and other database cluster components based on PostgreSQL 14.17 were added.

*Update from March 07, 2025:* images of PostgreSQL 13.20 and other database cluster components based on PostgreSQL 13.20 were added.

## Supported platforms

The Operator was developed and tested with PostgreSQL versions 12.20, 13.20, 14.17, 15.12, and 16.8. Other options may also work but have not been tested. The Operator 2.5.1 provides connection pooling based on pgBouncer 1.24.0 and high-availability implementation based on Patroni 3.3.2.

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

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

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

# Percona Operator for PostgreSQL 2.5.0

- **Date**

October 08, 2024

- **Installation**

[Installing Percona Operator for PostgreSQL](#)

## Release Highlights

### Automated storage scaling

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

```
- name: PGO_FEATURE_GATES
 value: "AutoGrowVolumes=true"
```



When the support for auto-growable disks is turned on, the

`spec.instances[].dataVolumeClaimSpec.resources.limits.storage` Custom Resource option sets the maximum value available for the Operator to scale up.

See [official documentation](#) for more details and limitations of the feature.

### Major versions upgrade improvements

Major version upgrade, introduced in the Operator version 2.4.0 as a tech preview, had undergone some improvements. Now it is possible to upgrade from one PostgreSQL major version to another with custom images for the database cluster components (PostgreSQL, pgBouncer, and pgBackRest). The upgrade is still triggered by applying the YAML manifest with the information about the existing and desired major versions, which now includes image names. The resulting manifest may look as follows:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGUpgrade
metadata:
 name: cluster1-15-to-16
spec:
 postgresClusterName: cluster1
 image: percona/percona-postgresql-operator:2.4.1-upgrade
 fromPostgresVersion: 15
 toPostgresVersion: 16
 toPostgresImage: percona/percona-postgresql-operator:2.5.0-ppg16.4-postgres
 toPgBouncerImage: percona/percona-postgresql-operator:2.5.0-ppg16.4-pgbouncer1.23.1
 toPgBackRestImage: percona/percona-postgresql-operator:2.5.0-ppg16.4-
pgbackrest2.53-1
```



## Azure Kubernetes Service and Azure Blob Storage support

[Azure Kubernetes Service \(AKS\)](#) is now an officially supported platform, so developers and vendors of the solutions based on the Azure platform can take advantage of the official support from Percona or just use officially certified Percona Operator for PostgreSQL images; also, [Azure Blob Storage can now be used for backups](#).

## New features

- [K8SPG-227](#) and [K8SPG-157](#): Add support for the [Azure Kubernetes Service \(AKS\)](#) platform and allow [using Azure Blob Storage](#) for backups
- [K8SPG-244](#): [Automated storage scaling](#) is now supported

## Improvements

- [K8SPG-630](#): A new `backups.trackLatestRestorableTime` Custom Resource option allows to disable latest restorable time tracking for users who need reducing S3 API calls usage
- [K8SPG-605](#) and [K8SPG-593](#): Documentation now includes information about [upgrading the Operator via Helm](#) and [using databaseInitSQL commands](#)
- [K8SPG-598](#): Database major version upgrade now [supports custom images](#)
- [K8SPG-560](#): A `pg-restore` Custom Resource is now automatically created at [bootstrapping a new cluster from an existing backup](#)
- [K8SPG-555](#): The Operator now creates separate Secret with CA certificate for each cluster
- [K8SPG-553](#): Users can provide the Operator with their own [root CA certificate](#)
- [K8SPG-454](#): Cluster status obtained with `kubectl get pg` command is now “ready” not only when all Pods are ready, but also takes into account if all StatefulSets are up to date

- [K8SPG-577](#): A new `pmm.querySource` Custom Resource option allows to set PMM query source

## Bugs Fixed

- [K8SPG-629](#): Fix a bug where the Operator was not deleting backup Pods when cleaning outdated backups according to the retention policy
- [K8SPG-499](#): Fix a bug where cluster was getting stuck in the init state if pgBackRest secret didn't exist
- [K8SPG-588](#): Fix a bug where the Operator didn't stop WAL watcher if the namespace and/or cluster were deleted
- [K8SPG-644](#): Fix a bug in the `pg-db` Helm chart which prevented from setting more than one Toleration

## Deprecation, Change, Rename and Removal

With the Operator versions prior to 2.5.0, [autogenerated TLS certificates](#) for all database clusters were based on the same generated root CA. Starting from 2.5.0, the Operator creates root CA on a per-cluster basis.

## Supported platforms

The Operator was developed and tested with PostgreSQL versions 12.20, 13.16, 14.13, 15.8, and 16.4. Other options may also work but have not been tested. The Operator 2.5.0 provides connection pooling based on pgBouncer 1.23.1 and high-availability implementation based on Patroni 3.3.2.

The following platforms were tested and are officially supported by the Operator 2.5.0:

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

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

# Percona Operator for PostgreSQL 2.4.1

- **Date**

August 6, 2024

- **Installation**

[Installing Percona Operator for PostgreSQL](#)

## Bugs Fixed

- [K8SPG-616](#): Fix a bug where it was not possible to create a new cluster after deleting the previous one with the `kubectl delete pg` command

## Supported platforms

The Operator was developed and tested with PostgreSQL versions 12.19, 13.15, 14.12, 15.7, and 16.3. Other options may also work but have not been tested. The Operator 2.4.1 provides connection pooling based on pgBouncer 1.22.1 and high-availability implementation based on Patroni 3.3.0.

The following platforms were tested and are officially supported by the Operator 2.4.1:

- [Google Kubernetes Engine \(GKE\)](#)  1.27 - 1.29
- [Amazon Elastic Container Service for Kubernetes \(EKS\)](#)  1.27 - 1.30
- [OpenShift](#)  4.12.59 - 4.15.18
- [Minikube](#)  1.33.1

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

# Percona Operator for PostgreSQL 2.4.0

- **Date**

June 26, 2024

- **Installation**

[Installing Percona Operator for PostgreSQL](#)

## Release Highlights

### Major versions upgrade (tech preview)

Starting from this release Operator users can automatically upgrade from one PostgreSQL major version to another. Upgrade is triggered by applying the yaml file with the information about the existing and desired major versions, with an example present in `deploy/upgrade.yaml`:

```
apiVersion: pgv2.percona.com/v2
kind: PerconaPGUpgrade
metadata:
 name: cluster1-15-to-16
spec:
 postgresClusterName: cluster1
 image: perconalab/percona-postgresql-operator:main-upgrade
 fromPostgresVersion: 15
 toPostgresVersion: 16
```



After applying it as usual, by running `kubectl apply -f deploy/upgrade.yaml` command, the actual upgrade takes place as follows:

1. The cluster is paused for a while,
2. The cluster is specially annotated with `pgv2.percona.com/allow-upgrade:<PerconaPGUpgrade.Name>` annotation,
3. Jobs are created to migrate the data,
4. The cluster starts up after the upgrade finishes.

Check official documentation for [more details](#), including ones about tracking the upgrade process and side effects for users with custom extensions.

## Supporting PostgreSQL tablespaces

Tablespaces allow DBAs to store a database on multiple file systems within the same server and to control where (on which file systems) specific parts of the database are stored. You can think about it as if you were giving names to your disk mounts and then using those names as additional parameters when creating database objects.

PostgreSQL supports this feature, allowing you to store data outside of the primary data directory. Tablespaces support was present in Percona Operator for PostgreSQL 1.x, and starting from this version, Percona Operator for PostgreSQL 2.x [can also bring](#) this feature to your Kubernetes environment, when needed.

## Using cloud roles to authenticate on the object storage for backups

Percona Operator for PostgreSQL has introduced a new feature that allows users to authenticate to AWS S3 buckets via [IAM roles](#) . [This enhancement](#) significantly improves security by eliminating the need to manage S3 access keys directly, while also streamlining the configuration process for easier backup and restore operations.

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

```
spec:
 crVersion: 2.4.0
 metadata:
 annotations:
 eks.amazonaws.com/role-arn: arn:aws:iam::1191:role/role-pgbackrest-access-s3-
bucket
 ...
 backups:
 pgbackrest:
 image: percona/percona-postgresql-operator:2.4.0-ppg16-pgbackrest
 global:
 repo1-s3-key-type: web-id
 ...
```



## New features

- [K8SPG-138](#): Users are now able to use AWS [IAM role](#)  to provide access to the S3 bucket used for backups
- [K8SPG-254](#): Now the Operator [automates](#) upgrading PostgreSQL major versions
- [K8SPG-459](#): PostgreSQL tablespaces [are now supported](#) by the Operator
- [K8SPG-479](#) and [K8SPG-492](#): It is now possible to specify tolerations for the [backup restore jobs](#) as well as for the [data move jobs](#) created when the Operator 1.x is upgraded to 2.x; this is useful in environments with

dedicated Kubernetes worker nodes protected by taints

- [K8SPG-503](#) and [K8SPG-513](#): It is now possible to specify [resources for the sidecar containers](#) of database instance Pods

## Improvements

- [K8SPG-259](#): Users can now change the default level for log messages for pgBackRest to simplify fixing backup and restore issues
- [K8SPG-542](#): Documentation now includes HowTo on [creating a disaster recovery cluster using streaming replication](#)
- [K8SPG-506](#): The `pg-backup` objects now have a new `backupName` status field, which allows users to [obtain the backup](#) name for restore simpler
- [K8SPG-514](#): The new `securityContext` Custom Resource subsections allow to configure `securityContext` for PostgreSQL instances, pgBouncer, and pgBackRest Pods
- [K8SPG-518](#): The `kubectl get pg-backup` command now shows the latest restorable time to make it easier to pick a point-in-time recovery target
- [K8SPG-519](#): The new `extensions.storage.endpoint` Custom Resource option allows specifying a custom S3 object storage endpoint for installing custom extensions
- [K8SPG-549](#): It is now possible to expose replica nodes through a separate Service, useful if you want to balance the load and separate reads and writes traffic
- [K8SPG-550](#): The default size for `/tmp` mount point in PMM container was increased from 1.5G to 2G
- [K8SPG-585](#): The namespace field was added to the Operator and database Helm chart templates

## Bugs Fixed

- [K8SPG-462](#): Fixed a bug where backups could not start if a previous backup had the same name
- [K8SPG-470](#): Liveness and Readiness probes timeouts [are now configurable](#) through Custom Resource
- [K8SPG-559](#): Fix a bug where the first full backup was incorrectly marked as incremental in the status field
- [K8SPG-490](#): Fixed broken replication that occurred after the network loss of the primary Pod with PostgreSQL 14 and older versions
- [K8SPG-502](#): Fix a bug where backup jobs were not cleaned up after completion
- [K8SPG-510](#): Fix a bug where pausing the cluster immediately set its state to “paused” instead of “stopping” while Pods were still running
- [K8SPG-531](#): Fix a bug where scheduled backups did not work for a second database with the same name in cluster-wide mode
- [K8SPG-535](#): Fix a bug where the Operator crashed when attempting to run a backup with a non-existent

repository

- [K8SPG-540](#): Fix a bug in the pg-db Helm chart readme where the key to set the backup secret was incorrectly specified (Thanks to Abhay Tiwari for contribution)
- [K8SPG-543](#): Fix a bug where applying a cr.yaml file with an empty `spec.proxy` field caused the Operator to crash
- [K8SPG-547](#): Fix dependency issue that made pgbackrest-repo container incompatible with pgBackRest 2.50, resulting in the older 2.48 version being used instead

## Deprecation and removal

- The `plpythonu` extension was removed from the list of built-in PostgreSQL extensions; users who still need it can enable it for their databases via [custom extensions functionality](#)

## Supported platforms

The Operator was developed and tested with PostgreSQL versions 12.19, 13.15, 14.12, 15.7, and 16.3. Other options may also work but have not been tested. The Operator 2.4.0 provides connection pooling based on pgBouncer 1.22.1 and high-availability implementation based on Patroni 3.3.0.

The following platforms were tested and are officially supported by the Operator 2.4.0:

- [Google Kubernetes Engine \(GKE\)](#)  1.27 - 1.29
- [Amazon Elastic Container Service for Kubernetes \(EKS\)](#)  1.27 - 1.30
- [OpenShift](#)  4.12.59 - 4.15.18
- [Minikube](#)  1.33.1

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

# Percona Operator for PostgreSQL 2.3.1

- **Date**

January 23, 2024

- **Installation**

[Installing Percona Operator for PostgreSQL](#)

## Release Highlights

This release provides a number of bug fixes, including fixes for the following vulnerabilities in PostgreSQL, pgBackRest, and pgBouncer images used by the Operator:

- OpenSSH could cause remote code execution by ssh-agent if a user establishes an SSH connection to a compromised or malicious SSH server and has agent forwarding enabled ([CVE-2023-38408](#) ). This vulnerability affects pgBackRest and PostgreSQL images.
- The c-ares library could cause a Denial of Service with 0-byte UDP payload ([CVE-2023-32067](#) ). This vulnerability affects pgBouncer image.

**Both Operator 1.x (including version 1.5.0) and Operator 2.x (including version 2.3.0) are affected. The 2.x versions [upgrade](#) to 2.3.1 is recommended to resolve these issues.**

## Bugs Fixed

- [K8SPG-493](#): Fix a regression due to which the Operator could run scheduled backup only one time
- [K8SPG-494](#): Fix vulnerabilities in PostgreSQL, pgBackRest, and pgBouncer images
- [K8SPG-496](#): Fix the bug where setting the `pause` Custom Resource option to `true` for the cluster with a backup running would not take effect even after the backup completed

## Supported platforms

The Operator was developed and tested with PostgreSQL versions 12.17, 13.13, 14.10, 15.5, and 16.1. Other options may also work but have not been tested. The Operator 2.3.1 provides connection pooling based on pgBouncer 1.21.0 and high-availability implementation based on Patroni 3.1.0.

The following platforms were tested and are officially supported by the Operator 2.3.1:

- [Google Kubernetes Engine \(GKE\)](#)  1.24 - 1.28
- [Amazon Elastic Container Service for Kubernetes \(EKS\)](#)  1.24 - 1.28

- [OpenShift](#)  4.11.55 - 4.14.6
- [Minikube](#)  1.32

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

# Percona Operator for PostgreSQL 2.3.0

- **Date**

December 21, 2023

- **Installation**

[Installing Percona Operator for PostgreSQL](#)

## Release Highlights

### PostGIS support

Modern businesses heavily rely on location-based data to gain valuable insights and make data-driven decisions. However, integrating geospatial functionality into the existing database systems has often posed a challenge for enterprises. PostGIS, an open-source software extension for PostgreSQL, addresses this difficulty by equipping users with extensive geospatial operations for handling geographic data efficiently. Percona Operator now supports PostGIS, available through a separate container image. You can read more about PostGIS and how to use it with the Operator in our [documentation](#).

### OpenShift and PostgreSQL 16 support

The Operator [is now compatible](#) with the OpenShift platform empowering enterprise customers with seamless on-premise or cloud deployments on the platform of their choice. Also, PostgreSQL 16 was added to the range of supported database versions and is used by default starting with this release.

### Experimental support for custom PostgreSQL extensions

One of great features of PostgreSQL is support for [Extensions](#) [↗](#), which allow adding new functionality to the database on a plugin basis. Starting from this release, users can add custom PostgreSQL extensions dynamically, without the need to rebuild the container image (see [this HowTo](#) on how to create and connect yours).

## New features

- [K8SPG-311](#) and [K8SPG-389](#): A new `loadBalancerSourceRanges` Custom Resource option allows to customize the range of IP addresses from which the load balancer should be reachable
- [K8SPG-375](#): Experimental support for custom PostgreSQL extensions [was added](#) to the Operator
- [K8SPG-391](#): The Operator [is now compatible](#) with the OpenShift platform

- [K8SPG-434](#): The Operator now supports Percona Distribution for PostgreSQL version 16 and uses it as default database version

## Improvements

- [K8SPG-413](#): The Operator documentation now includes a [compatibility matrix](#) for each Operator version, specifying exact versions of all core components as well as supported versions of the database and platforms
- [K8SPG-332](#): Creating backups and [pausing the cluster](#) do not interfere with each other: the Operator either postpones the pausing until the active backup ends, or postpones the scheduled backup on the paused cluster
- [K8SPG-370](#): [Logging management](#) is now aligned with other Percona Operators, allowing to use structured logging and to control log level
- [K8SPG-372](#): The multi-namespace (cluster-wide) mode of the Operator was improved, making it possible to customize the list of Kubernetes namespaces under the Operator's control
- [K8SPG-400](#): The documentation now explains how to allow application users to connect to a database cluster [without TLS](#) (for example, for testing or demonstration purposes)
- [K8SPG-410](#): Scheduled backups now create `pg-backup` object to simplify backup management and tracking
- [K8SPG-416](#): PostgreSQL custom configuration is now supported in the Helm chart
- [K8SPG-422](#) and [K8SPG-447](#): The user can now see backup type and status in the output of `kubectl get pg-backup` and `kubectl get pg-restore` commands
- [K8SPG-458](#): Affinity configuration examples were added to the `default/cr.yaml` configuration file

## Bugs Fixed

- [K8SPG-435](#): Fix a bug with insufficient size of `/tmp` filesystem which caused PostgreSQL Pods to be recreated every few days due to running out of free space on it
- [K8SPG-453](#): Bug in `pg_stat_monitor` PostgreSQL extensions could hang PostgreSQL
- [K8SPG-279](#): Fix regression which made the Operator to crash after creating a backup if there was no `backups.pgbackrest.manual` section in the Custom Resource
- [K8SPG-310](#): Documentation didn't explain how to apply `pgBackRest verifyTLS` option which can be used to explicitly enable or disable TLS verification for it
- [K8SPG-432](#): Fix a bug due to which backup jobs and Pods were not deleted on deleting the backup object
- [K8SPG-442](#): The Operator didn't allow to append custom items to the PostgreSQL `shared_preload_libraries` option
- [K8SPG-443](#): Fix a bug due to which only English locale was installed in the PostgreSQL image, missing

other languages support

- [K8SPG-450](#): Fix a bug which prevented PostgreSQL to initialize the database on Kubernetes working nodes with enabled huge memory pages if Pod resource limits didn't allow using them
- [K8SPG-401](#): Fix a bug which caused Operator crash if deployed with no `pmm` section in the `deploy/cr.yaml` configuration file

## Supported platforms

The Operator was developed and tested with PostgreSQL versions 12.17, 13.13, 14.10, 15.5, and 16.1. Other options may also work but have not been tested. The Operator 2.3.0 provides connection pooling based on pgBouncer 1.21.0 and high-availability implementation based on Patroni 3.1.0.

The following platforms were tested and are officially supported by the Operator 2.3.0:

- [Google Kubernetes Engine \(GKE\)](#)  1.24 - 1.28
- [Amazon Elastic Container Service for Kubernetes \(EKS\)](#)  1.24 - 1.28
- [OpenShift](#)  4.11.55 - 4.14.6
- [Minikube](#)  1.32

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

# Percona Operator for PostgreSQL 2.2.0

- **Date**

June 30, 2023

- **Installation**

[Installing Percona Operator for PostgreSQL](#)

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

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

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

## Improvements

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

## Bugs Fixed

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

#### Known limitations

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

## Supported platforms

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

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

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

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

# Percona Operator for PostgreSQL 2.1.0 (Tech preview)

- **Date**

May 4, 2023

- **Installation**

[Installing Percona Operator for PostgreSQL](#)

The Percona Operator built with best practices of configuration and setup of [Percona Distribution for PostgreSQL on Kubernetes](#) [↗](#).

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

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



## Note

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

## Release Highlights

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

## New Features

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

## Improvements

- [K8SPG-262](#): The Operator now does not attempt to start Percona Monitoring and Management (PMM) client if the corresponding secret does not contain the `pmmserver` or `pmmserverkey` key
- [K8SPG-285](#): To improve the Operator we capture anonymous telemetry and usage data. In this release we [add more data points](#) to it
- [K8SPG-295](#): Additional information was added to the status of the Operator Custom Resource, which now shows `name`, `endpoint`, `status`, and `age` fields
- [K8SPG-304](#): The Operator stops using trust authentication method in `pg_hba.conf` for better security
- [K8SPG-325](#): Custom Resource options previously named `paused` and `shutdown` were renamed to `unmanaged` and `pause` for better alignment with other Percona Operators

## Bugs Fixed

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

## Supported platforms

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

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

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

# Percona Operator for PostgreSQL 2.0.0 (Tech preview)

- **Date**

December 30, 2022

- **Installation**

[Installing Percona Operator for PostgreSQL](#)

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



## Note

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

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

## Architecture

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

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

## Backups

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

- Not possible to change backup configuration for the existing cluster

- Restoration from backup to the newly deployed cluster required workarounds

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

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

## Operations

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

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

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

## Try it out now

Excited with what you read above?

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