

Crunchy Postgres for Kubernetes from Crunchy Data

[Crunchy Postgres for Kubernetes](#) is the leading Kubernetes native Postgres solution. Built on [PGO](#), the Postgres Operator from Crunchy Data, Crunchy Postgres for Kubernetes gives you a declarative Postgres solution that automatically manages your PostgreSQL clusters providing:

- Fast, easy deployment
- High availability
- Backup management and disaster recovery
- Connection management and scaling
- Performance and health monitoring
- Much more

Topics to get started



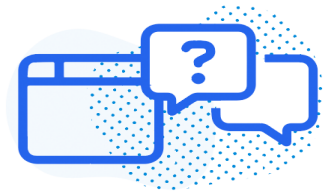
Get started

Create and connect to your cluster



Architecture

Understand the key components of Crunchy Postgres for Kubernetes



Supported platforms

Guidance on supported Kubernetes, OpenShift, and Postgres versions.

Quickstart

Can't wait to try out [Crunchy Postgres for Kubernetes](#)? Let us show you the quickest possible path to getting up and running.

This quick start is for **kustomize** and **kubectl**. We also have instructions for installing via Helm and OperatorHub, as well as more detailed instructions for **kustomize**.

Prerequisites

Please be sure you have the following utilities installed on your host machine:

- **kubectl**
- **git**

Installation

Step 1: Download the Examples

First, go to GitHub and [fork the Postgres Operator examples](#) repository:

<https://github.com/CrunchyData/postgres-operator-examples/fork>

Once you have forked this repo, you can download it to your working environment with a command similar to this:

```
YOUR_GITHUB_UN="$YOUR_GITHUB_USERNAME"
git clone --depth 1 "git@github.com:${YOUR_GITHUB_UN}/postgres-operator-examples.git"
cd postgres-operator-examples
```

For Powershell environments:

```
$env:YOUR_GITHUB_UN="YOUR_GITHUB_USERNAME"
git clone --depth 1 "git@github.com:$env:YOUR_GITHUB_UN/postgres-operator-examples.git"
cd postgres-operator-examples
```

Step 2: Install PGO, the Postgres Operator

You can install PGO, the Postgres Operator from Crunchy Data, using the command below:

```
kubectl apply -k kustomize/install/namespace
kubectl apply --server-side -k kustomize/install/default
```

This will create a namespace called **postgres-operator** and create all of the objects required to deploy PGO.

To check on the status of your installation, you can run the following command:

```
kubectl -n postgres-operator get pods --selector=postgres-operator.crunchydata.com/control-plane=postgres-operator --field-selector=status.phase=Running
```

If the PGO Pod is healthy, you should see output similar to:

NAME	READY	STATUS	RESTARTS	AGE
postgres-operator-9dd545d64-t4h8d	1/1	Running	0	3s

Create a Postgres Cluster

Let's create a simple Postgres cluster. You can do this by executing the following command:

```
kubectl apply -k kustomize/postgres
```

This will create a Postgres cluster named **hippo** in the **postgres-operator** namespace. You can track the progress of your cluster using the following command:

```
kubectl -n postgres-operator describe postgresclusters.postgres-operator.crunchydata.com hippo
```

Connect to the Postgres cluster

As part of creating a Postgres cluster, the Postgres Operator creates a PostgreSQL user account. The credentials for this account are stored in a Secret that has the name `<clusterName>-pguser-<userName>`.

Within this Secret are attributes that provide information to let you log into the PostgreSQL cluster. These include:

- **user**: The name of the user account.
- **password**: The password for the user account.
- **dbname**: The name of the database that the user has access to by default.
- **host**: The name of the host of the database. This references the [Service](#) of the primary Postgres instance.
- **port**: The port that the database is listening on.
- **uri**: A [PostgreSQL connection URI](#) that provides all the information for logging into the Postgres database.
- **jdbc-uri**: A [PostgreSQL JDBC connection URI](#) that provides all the information for logging into the Postgres database via the JDBC driver.

If you deploy your Postgres cluster with the [PgBouncer](#) connection pooler, there are additional values that are populated in the user Secret, including:

- **pgbouncer-host**: The name of the host of the PgBouncer connection pooler. This references the [Service](#) of the PgBouncer connection pooler.
- **pgbouncer-port**: The port that the PgBouncer connection pooler is listening on.
- **pgbouncer-uri**: A [PostgreSQL connection URI](#) that provides all the information for logging into the Postgres database via the PgBouncer connection pooler.
- **pgbouncer-jdbc-uri**: A [PostgreSQL JDBC connection URI](#) that provides all the information for logging into the Postgres database via the PgBouncer connection pooler using the JDBC driver.

Note that **all connections use TLS**. PGO sets up a public key infrastructure (PKI) for your Postgres clusters. You can also choose to bring your own PKI / certificate authority; this is covered later in the documentation.

Connect via `psql` in the Terminal

Connect Directly

If you are on the same network as your PostgreSQL cluster, you can connect directly to it using the following command:

```
psql $(kubectl -n postgres-operator get secrets hippo-pguser-hippo -o go-template='{{.data.uri | base64decode}}')
```

Connect Using a Port-Forward

In a new terminal, create a port forward. If you are using Bash, you can run the following commands:

```
PG_CLUSTER_PRIMARY_POD=$(kubectl get pod -n postgres-operator -o name -l postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/role=master)
kubectl -n postgres-operator port-forward "${PG_CLUSTER_PRIMARY_POD}" 5432:5432
```

For Powershell environments:

```
$env:PG_CLUSTER_PRIMARY_POD=(kubectl get pod -n postgres-operator -o name -l postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/role=master)
kubectl -n postgres-operator port-forward "$env:PG_CLUSTER_PRIMARY_POD" 5432:5432
```

Establish a connection to the PostgreSQL cluster. If you are using Bash, you can run:

```
PG_CLUSTER_USER_SECRET_NAME=hippo-pguser-hippo

PGPASSWORD=$(kubectl get secrets -n postgres-operator "${PG_CLUSTER_USER_SECRET_NAME}" -o go-template='{{.data.password | base64decode}}') \
PGUSER=$(kubectl get secrets -n postgres-operator "${PG_CLUSTER_USER_SECRET_NAME}" -o go-template='{{.data.user | base64decode}}') \
PGDATABASE=$(kubectl get secrets -n postgres-operator "${PG_CLUSTER_USER_SECRET_NAME}" -o go-template='{{.data.dbname | base64decode}}') \
psql -h localhost
```

For Powershell environments:

```
$env:PG_CLUSTER_USER_SECRET_NAME="hippo-pguser-hippo"

$env:PGPASSWORD=(kubectl get secrets -n postgres-operator "$env:PG_CLUSTER_USER_SECRET_NAME" -o go-template='{{.data.password | base64decode}}')
$env:PGUSER=(kubectl get secrets -n postgres-operator "$env:PG_CLUSTER_USER_SECRET_NAME" -o go-template='{{.data.user | base64decode}}')
$env:PGDATABASE=(kubectl get secrets -n postgres-operator "$env:PG_CLUSTER_USER_SECRET_NAME" -o go-template='{{.data.dbname | base64decode}}')
psql -h localhost
```

Create a user schema

Starting in [Postgres 15](#), **PUBLIC** creation permission on the public schema has been removed, but there is a simple way forward to allow you to start writing queries.

As described in our helpful [blog post](#) on the subject, after connecting via **psql** as the **hippo** user, just execute

```
CREATE SCHEMA hippo AUTHORIZATION hippo;
```

and you will be able to create tables in the **hippo** schema without any additional steps or permissions.

Info

Want all the users you define in the spec to have schemas automatically created for them? As of v5.6.1, you can do that! See how to in our section on [Automatically Creating Schema for Users](#).

Connect an Application

The information provided in the user Secret will allow you to connect an application directly to your PostgreSQL database.

For example, let's connect [Keycloak](#). Keycloak is a popular open source identity management tool that is backed by a PostgreSQL database. Using the **hippo** cluster we created, we can deploy the following manifest file:

```

cat <<EOF >> keycloak.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: keycloak
  namespace: postgres-operator
  labels:
    app.kubernetes.io/name: keycloak
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: keycloak
  template:
    metadata:
      labels:
        app.kubernetes.io/name: keycloak
    spec:
      containers:
        - image: quay.io/keycloak/keycloak:latest
          args: ["start-dev"]
          name: keycloak
          env:
            - name: DB_VENDOR
              value: "postgres"
            - name: DB_ADDR
              valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: host } }
            - name: DB_PORT
              valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: port } }
            - name: DB_DATABASE
              valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: dbname } }
            - name: DB_USER
              valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: user } }
            - name: DB_PASSWORD
              valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: password } }
            - name: KEYCLOAK_ADMIN
              value: "admin"
            - name: KEYCLOAK_ADMIN_PASSWORD
              value: "admin"
            - name: KC_PROXY
              value: "edge"
          ports:
            - name: http
              containerPort: 8080
            - name: https
              containerPort: 8443
          readinessProbe:
            httpGet:
              path: /realms/master
              port: 8080
          restartPolicy: Always
      EOF

kubectl apply -f keycloak.yaml

```

There is a full example for how to deploy Keycloak with the Postgres Operator in the **customize/keycloak** folder.

Next Steps

Congratulations, you've got your Postgres cluster up and running, perhaps with an application connected to it!

You can find out more about the `postgresclusters` custom resource definition through the documentation and through `kubectl explain`.e.:

```
kubectl explain postgresclusters
```

You've seen how easy it is to get a Postgres database up and running and connected to your applications using Crunchy Postgres for Kubernetes. In the next section we will take a closer look at CPK and how its different components work together to provide everything you need for a production-ready Postgres cluster.

Overview

[Crunchy Postgres for Kubernetes](#) is the leading Kubernetes native Postgres solution. Built on [PGO](#), the Postgres Operator from Crunchy Data, Crunchy Postgres for Kubernetes gives you a declarative Postgres solution that automatically manages your PostgreSQL clusters.

Designed for seamless integration with your GitOps workflows, getting started with Postgres on Kubernetes is effortless. Within minutes, you can deploy a production-grade Postgres cluster featuring high availability, disaster recovery, and monitoring, all secured with TLS communications. Crunchy Postgres for Kubernetes also allows for easy customization to tailor the cluster to your specific workload needs. Additionally, you have the flexibility to run Postgres on your own infrastructure or choose a fully managed solution with Crunchy Bridge.

With conveniences like cloning Postgres clusters to using rolling updates to safely roll out disruptive changes with minimal downtime, Crunchy Postgres for Kubernetes is ready to support your Postgres data at every stage of your release pipeline. Built for resiliency and uptime, Crunchy Postgres for Kubernetes will keep your desired Postgres in a desired state so you do not need to worry about it.

Crunchy Postgres for Kubernetes is developed with many years of production experience in automating Postgres management on Kubernetes, providing a seamless cloud native Postgres solution to keep your data always available.

Key Components

Crunchy Postgres for Kubernetes is designed to provide production ready Kubernetes-native Postgres clusters using a few key components:

- PGO, the Postgres Operator from Crunchy Data, is the brains behind Crunchy Postgres for Kubernetes enabling users to interact with their Postgres clusters through PGO. To accomplish this, PGO extends Kubernetes to provide a higher-level abstraction for rapid creation and management of PostgreSQL clusters by leveraging "[Custom Resources](#)" to create several [custom resource definitions \(CRDs\)](#) that allow for the management of PostgreSQL clusters. PGO itself runs as a Deployment and is composed of a single container.
- Crunchy Postgres, Crunchy Data's open source distribution of Postgres, along with leading Postgres tools and extensions such as pgbackrest, Patroni, pgaudit, PostGIS, and more. Each of the components within Crunchy Postgres are built with upstream source code and compiled, tested and certified by Crunchy Data. These components are provided as a series of containers via the Crunchy Data access and developer portals.
- The Crunchy Postgres for Kubernetes monitoring stack, a fully integrated solution for monitoring and visualizing key metrics pertaining to your Postgres databases, as well the containers they run within. Built on industry standards for

monitoring and metrics collection, the Crunchy Postgres for Kubernetes monitoring stack ensures you have the real-time insights needed to keep all of your Postgres databases running smoothly and efficiently.

- Installers for Kustomize, Helm and OLM, providing flexibility to seamlessly and easily install and deploy Postgres clusters regardless of your specific Kubernetes distribution, or your preferred tooling for deploying to Kubernetes.

For more detailed architecture information or a full list of components include in Crunchy Postgres for Kubernetes, see:

- Architecture
- Supported Platforms
- Release Notes

Architecture

Several pieces must come together to create a production-ready Postgres cluster and Crunchy Postgres for Kubernetes provides everything that you need. From high-availability to disaster recovery and monitoring, we'll cover how a Crunchy Postgres for Kubernetes deployment fits the pieces together.

Operator

PGO, the Postgres Operator from Crunchy Data, runs as a [Kubernetes Deployment](#) and is composed of a single container. This PGO container holds a collection of Kubernetes controllers that manage native Kubernetes resources (Jobs, Pods) as well as [Custom Resources](#) (PostgresCluster). As a user, you provide Kubernetes with the specification of what you want your Postgres cluster to look like and PGO uses a [Custom Resource Definition](#) (CRD) to teach Kubernetes how to handle those specifications. PGO's controllers do the work of making your specifications a reality. The main custom resource definition is `postgresclusters.postgres-operator.crunchydata.com`. This CRD allows you to control all the information about a Postgres cluster, including:

- Resource allocation
- High availability
- Backup management
- Where and how your cluster is deployed (affinity, tolerations, topology spread constraints)
- Disaster Recovery / standby clusters
- Monitoring
- and more.

Crunchy Postgres

Crunchy Postgres for Kubernetes enables you to deploy Kubernetes-native production ready clusters of Crunchy Postgres, Crunchy Data's open source Postgres distribution. When you use one of Crunchy Data's installers, you're given the option to install and deploy a range of Crunchy Postgres versions and specify the number of replicas (in addition to your primary Postgres instance) in your cluster. The spec you create for the deployment will command Kubernetes to create a number of Pods corresponding to the number of Postgres clusters, each running a container with Crunchy Postgres inside.

Crunchy Postgres for Kubernetes uses [Kubernetes Statefulsets](#) to create Postgres instance groups and support advanced operations such as rolling updates to minimize Postgres downtime as well as affinity and toleration rules to force one or more replicas to run on nodes in different regions.

pgBackRest

A production-ready Postgres cluster demands a disaster recovery solution. Crunchy Postgres for Kubernetes uses pgBackRest to backup and restore your data. With [pgBackRest](#), you can perform scheduled backups, one-off backups and point-in-time recoveries. Crunchy Postgres for Kubernetes enables pgBackRest by default. When a new Postgres cluster is created, a pgBackRest repository is created too. Crunchy Postgres for Kubernetes runs pgBackrest in the same pod that runs your Crunchy Postgres container. A separate pgBackRest pod can be used to manage backups through cloud storage services such as S3, GCS, and Azure.

Patroni

You want your data to always be available. Maintaining high availability requires a cluster of Postgres instances where there is one leader and some number of replicas. If the leader instance goes down, Crunchy Postgres for Kubernetes uses Patroni to promote a new leader from your replicas. Each container running a Crunchy Postgres instance comes loaded with Patroni to handle failover and keep your data available.

Monitoring Stack

Resource starvation happens. You can run out of storage space and you can run out of computing power. Crunchy Postgres for Kubernetes provides a monitoring stack to help you track the health of your Postgres cluster, replete with dashboards, alerts, and insights into your workloads. While having high availability, backups, and disaster recovery systems in place helps in the event of something going wrong with your Postgres cluster, monitoring helps you anticipate problems before they happen. The monitoring stack includes components provided by [pgMonitor](#) and [pgnodemx](#) and deploys as a collection of pods containing [Grafana](#), [Alertmanager](#), and [Prometheus](#).

Supported Platforms

Kubernetes, OpenShift, Postgres Versions

Crunchy Postgres for Kubernetes is compatible with the following Kubernetes and OpenShift versions. Crunchy Postgres for Kubernetes is generally compatible with Kubernetes, and for specific distribution compatibility, please feel free to contact us.

Crunchy Postgres for Kubernetes Series	Kubernetes Version	OpenShift Version	Postgres version	Status
5.8.x	1.30–32	4.14–18	13–17 ¹	Active / Developer
5.7.x	1.28–32	4.12–18	13–17 ¹	Active / Developer
5.6.x	1.27–32	4.12–18	13–16 ¹	Active
5.5.x	1.25–30	4.10–15	13–16 ¹	Active
5.4.x	1.24–29	4.10–15	11–16 ¹	Extended
5.3.x	1.22–26	4.8–13	11–15	Extended
5.2.x	1.21–24	4.6–10	11–14	Extended
5.1.x	1.20–24	4.6–10	11–14	Extended
5.0.x	1.20–24	4.6–10	10–14	Extended
4.7.x	1.17–26	4.4–12	11–13	Extended
4.6.x	1.17–21	4.4–12	11–13	Extended

In accordance with the Crunchy Developer Program, the latest two major versions of Postgres are published to the Crunchy Data Developer registry and the Red Hat certified image catalog.

Availability

- **Active:** Available through Crunchy Data Subscription.
- **Extended:** Crunchy Data 'Extended' Support Subscription Available.
- **Developer:** Available through Developer Program.

If you want to check all of the version information for a release, see Components and Compatibility.

Release Frequency

Crunchy Postgres for Kubernetes plans to release on the following frequency.

Monthly Patch Updates	Postgres Minor Versions	Postgres Major Versions	Crunchy Postgres for Kubernetes Updates
Developer Portal			
RedHat Marketplace			
Customer Portal			

Crunchy Data Subscription provides customers with access to all available Crunchy Postgres for Kubernetes versions, including updates and bug fixes. Crunchy Data will generally maintain the current and two past versions as Active. For more information about version life cycle or Crunchy Data update and release, please see our [contact us](#) or contact us directly via email at info@crunchydata.com.

Installation

This section provides detailed instructions for anything and everything related to installing Crunchy Postgres for Kubernetes. This includes instructions for installing according to a variety of methods, along with information for customizing an installation to your specific needs.

Guidance on adjusting which images your cluster will run can be found in Configuring Cluster Images.

Install Crunchy Postgres for Kubernetes

- Kustomize Install
- Helm Install
- OperatorHub Install

Next Step: Create a Postgres Cluster

Now that you've installed Crunchy Postgres for Kubernetes, you're ready to Create a Postgres Cluster.

Next Step: Install Monitoring

No installation of Crunchy Postgres for Kubernetes is complete without monitoring! See our Tutorial on installing monitoring for details.

Kustomize

Installing Crunchy Postgres for Kubernetes Using Kustomize

If you are deploying using the installer from the [Crunchy Data Customer Portal](#), please refer to the guide there for alternative setup information.

Prerequisites

First, go to GitHub and [fork the Postgres Operator examples](#) repository, which contains the Crunchy Postgres for Kubernetes Kustomize installer.

<https://github.com/CrunchyData/postgres-operator-examples/fork>

Once you have forked this repo, you can download it to your working environment with a command similar to this:

```
YOUR_GITHUB_UN="$YOUR_GITHUB_USERNAME"
git clone --depth 1 "git@github.com:${YOUR_GITHUB_UN}/postgres-operator-examples.git"
cd postgres-operator-examples
```

For Powershell environments:

```
$env:YOUR_GITHUB_UN="YOUR_GITHUB_USERNAME"
git clone --depth 1 "git@github.com:$env:YOUR_GITHUB_UN/postgres-operator-examples.git"
cd postgres-operator-examples
```

The Crunchy Postgres for Kubernetes installation project is located in the **kustomize/install** directory.

Configuration

While the default Kustomize install should work in most Kubernetes environments, it may be necessary to further customize the Kustomize project(s) according to your specific needs.

For instance, to customize the image tags utilized for the Crunchy Postgres for Kubernetes Deployment, the **images** setting in the **kustomize/install/default/kustomization.yaml** file can be modified:

```
images:
- name: postgres-operator
  newName: registry.developers.crunchydata.com/crunchydata/postgres-operator
  newTag: ubi9-5.8.2-0
```

If you are deploying using the images from the [Crunchy Data Customer Portal](#), please refer to the private registries guide for additional setup information.

Please note that the Kustomize install project will also create a namespace for Crunchy Postgres for Kubernetes by default (though it is possible to install without creating the namespace, as shown below). To modify the name of namespace created by the installer, the `kustomize/install/namespace/namespace.yaml` should be modified:

```
apiVersion: v1
kind: Namespace
metadata:
  name: custom-namespace
```

The `namespace` setting in `kustomize/install/default/kustomization.yaml` should be modified accordingly.

```
namespace: custom-namespace
```

By default, Crunchy Postgres for Kubernetes deploys with debug logging turned on. If you wish to disable this, you need to set the `CRUNCHY_DEBUG` environmental variable to `"false"` that is found in the `kustomize/install/manager/manager.yaml` file. Alternatively, you can add the following to your `kustomize/install/manager/kustomization.yaml` to disable debug logging:

```
patchesStrategicMerge:
- |-
  apiVersion: apps/v1
  kind: Deployment
  metadata:
    name: pgo
  spec:
    template:
      spec:
        containers:
        - name: operator
          env:
            - name: CRUNCHY_DEBUG
              value: "false"
```

You can also create additional Kustomize overlays to further patch and customize the installation according to your specific needs.

Installation Mode

When Crunchy Postgres for Kubernetes is installed, it can be configured to manage PostgreSQL clusters in all namespaces within the Kubernetes cluster, just those within a single namespace, or, starting in CPK 5.7, those in a select set of namespaces. When managing PostgreSQL clusters in multiple namespaces, a ClusterRole and ClusterRoleBinding is created to ensure Crunchy Postgres for Kubernetes has the permissions it requires to properly manage PostgreSQL clusters across all namespaces. However, when Crunchy Postgres for Kubernetes is configured to manage PostgreSQL clusters within a single namespace only, a Role and RoleBinding is created instead.

The installation of the necessary resources for a cluster-wide or a single-namespace-limited operator is done automatically by Kustomize, as described below in the Install section. If you wish for the operator to only manage PostgreSQL clusters in a select set of namespaces, you will need to make a change to the `kustomize/install/manager/manager.yaml` file. Open the file and to the list of operator container environment variables, add a variable with the name `PGO_TARGET_NAMESPACES`, and for the value enter the desired namespaces in a double-quoted, comma-separated list. For example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pgo
spec:
  template:
    spec:
      containers:
      - name: operator
        env:
        - name: PGO_TARGET_NAMESPACES
          value: "namespace-one, namespace-two, namespace-three"
```

The only other potential change you may need to make is to the Namespace resource and the **namespace** field if using a namespace other than the default **postgres-operator**.

High Availability

Starting in CPK 5.7, the operator can run in a typical hot/cold high availability configuration. When enabled, one pod will be the leader, while others wait to become the leader should the current leader fail. This capability is controlled by the **PGO_CONTROLLER_LEASE_NAME** environment variable on the PGO deployment. That value names the [Lease](#) object used to elect a leader of the deployment. The default is **cpk-leader-election-lease**, so you can achieve high availability by setting the Deployment **replicas** greater than one.

If you wish to disable this capability, empty or remove the **PGO_CONTROLLER_LEASE_NAME** environment variable and set **replicas** to 1.

Health Probes

Starting in CPK 5.7, the operator has the ability to perform liveness and readiness health probes. These probes are set on the operator Deployment and are enabled by default with the following settings:

```
livenessProbe:
  httpGet:
    path: /readyz
    port: 8081
  initialDelaySeconds: 15
  periodSeconds: 20
readinessProbe:
  httpGet:
    path: /healthz
    port: 8081
  initialDelaySeconds: 5
  periodSeconds: 10
```

To disable these probes, simply remove them from the operator Deployment.

Install

Once the Kustomize project has been modified according to your specific needs, Crunchy Postgres for Kubernetes can then be installed using **kubect1** and Kustomize. To create the target namespace, run the following:

```
kubect1 apply -k kustomize/install/namespace
```

This will create the default **postgres-operator** namespace, unless you have edited the **kustomize/install/namespace/namespace.yaml** resource. That **Namespace** resource should have the same value as the **namespace** field in the **kustomization.yaml** file (located either at **kustomize/install/default** or **kustomize/install/singlenamespace**, depending on whether you are deploying the operator with cluster-wide or single-namespace-limited permissions).

To install Crunchy Postgres for Kubernetes itself in cluster-wide mode (or multi-namespace mode if you have added the **PGO_TARGET_NAMESPACES** environment variable), apply the kustomization file in the **default** folder:

```
kubectl apply --server-side -k kustomize/install/default
```

To install Crunchy Postgres for Kubernetes itself in single-namespace-limited mode, apply the kustomization file in the **singlenamespace** folder:

```
kubectl apply --server-side -k kustomize/install/singlenamespace
```

The **kustomization.yaml** files in those folders take care of applying the appropriate permissions.

Install the Custom Resource Definition using Older Kubectl

This installer is optimized for Kustomize v4.0.5 or later, which is included in **kubectl** v1.21.

If you are using an earlier version of **kubectl** to manage your Kubernetes objects, you should be able to create the namespace as described above, but when you run the **kubectl apply --server-side -k kustomize/install/default** command, you will get an error like:

```
Error: json: unknown field "labels"
```

To fix this error, download the most recent version of [Kustomize](#).

Once you have installed Kustomize v4.0.5 or later, you can use it to produce valid Kubernetes yaml:

```
kustomize build kustomize/install/default
```

The output from the **kustomize build** command can be captured to a file or piped directly to **kubectl**:

```
kustomize build kustomize/install/default | kubectl apply --server-side -f -
```

Automated Upgrade Checks

By default, Crunchy Postgres for Kubernetes will automatically check for updates to itself and software components by making a request to a URL. If Crunchy Postgres for Kubernetes detects there are updates available, it will print them in the logs. As part of the check, Crunchy Postgres for Kubernetes will send aggregated, anonymized information about the current deployment to the endpoint. An upcoming release will allow for Crunchy Postgres for Kubernetes to opt-in to receive and apply updates to software components automatically.

Crunchy Postgres for Kubernetes will check for updates upon startup and once every 24 hours. Any errors in checking will have no impact on the operation of Crunchy Postgres for Kubernetes. To disable the upgrade check, you can set the **CHECK_FOR_UPGRADES** environmental variable on the **pgo** Deployment to **"false"**.

For more information about collected data, see the Crunchy Data [collection notice](#).

Uninstall

Once Crunchy Postgres for Kubernetes has been installed, it can also be uninstalled using **kubect1** and Kustomize. To uninstall Crunchy Postgres for Kubernetes (assuming it was installed in cluster-wide mode), the following command can be utilized:

```
kubect1 delete -k kustomize/install/default
```

To uninstall Crunchy Postgres for Kubernetes installed with only namespace permissions, use:

```
kubect1 delete -k kustomize/install/singlenamespace
```

The namespace created with this installation can likewise be cleaned up with:

```
kubect1 delete -k kustomize/install/namespace
```

Next Step: Create a Postgres Cluster

Now that you've installed Crunchy Postgres for Kubernetes, you're ready to Create a Postgres Cluster.

Next Step: Install Monitoring

No installation of Crunchy Postgres for Kubernetes is complete without monitoring! See our tutorial on installing monitoring with Kustomize for details.

Helm

Installing Crunchy Postgres for Kubernetes Using Helm

This section provides instructions for installing and configuring Crunchy Postgres for Kubernetes using Helm.

There are two sources for the Crunchy Postgres for Kubernetes Helm chart:

- the Postgres Operator examples repo;
- the Helm chart hosted on the Crunchy container registry, which supports direct Helm installs.

The Postgres Operator Examples repo

Prerequisites

First, go to GitHub and [fork the Postgres Operator examples](https://github.com/CrunchyData/postgres-operator-examples) repository, which contains the Crunchy Postgres for Kubernetes Helm installer.

<https://github.com/CrunchyData/postgres-operator-examples/fork>

Once you have forked this repo, you can download it to your working environment with a command similar to this:

```
YOUR_GITHUB_UN="$YOUR_GITHUB_USERNAME"
git clone --depth 1 "git@github.com:${YOUR_GITHUB_UN}/postgres-operator-examples.git"
cd postgres-operator-examples
```

For Powershell environments:

```
$env:YOUR_GITHUB_UN="YOUR_GITHUB_USERNAME"
git clone --depth 1 "git@github.com:$env:YOUR_GITHUB_UN/postgres-operator-examples.git"
cd postgres-operator-examples
```

The Crunchy Postgres for Kubernetes Helm chart is located in the `helm/install` directory of this repository.

Configuration

The `values.yaml` file for the Helm chart contains all of the available configuration settings for Crunchy Postgres for Kubernetes. The default `values.yaml` settings should work in most Kubernetes environments, but it may require some customization depending on your specific environment and needs.

For instance, it might be necessary to customize the image tags that are utilized using the `controllerImages` setting:

```
controllerImages:
  cluster: registry.developers.crunchydata.com/crunchydata/postgres-operator:ubi9-5.8.2-0
```

Please note that the `values.yaml` file is located in `helm/install`.

Logging

By default, Crunchy Postgres for Kubernetes deploys with debug logging turned on. If you wish to disable this, you need to set the `debug` attribute in the `values.yaml` to false, e.g.:

```
debug: false
```

Installation Mode

When Crunchy Postgres for Kubernetes is installed, it can be configured to manage PostgreSQL clusters in all namespaces within the Kubernetes cluster, just those within a single namespace, or, starting in CPK 5.7, those in a select set of namespaces. When managing PostgreSQL clusters in multiple namespaces, a ClusterRole and ClusterRoleBinding is created to ensure Crunchy Postgres for Kubernetes has the permissions it requires to properly manage PostgreSQL clusters across all namespaces. However, when Crunchy Postgres for Kubernetes is configured to manage PostgreSQL clusters within a single namespace only, a Role and RoleBinding is created instead.

In order to select between the multi-namespace and single-namespace modes when installing Crunchy Postgres for Kubernetes using Helm, the `singleNamespace` setting in the `values.yaml` file can be utilized:

```
singleNamespace: false
```

Specifically, if this setting is set to **false** (which is the default), then a ClusterRole and ClusterRoleBinding will be created, and Crunchy Postgres for Kubernetes will be able to manage PostgreSQL clusters in multiple namespaces. By default, the operator will manage PostgreSQL clusters in all namespaces. However, if you wish for the operator to only manage PostgreSQL clusters in a select set of namespaces, you will need to make a change to the **helm/install/templates/manager.yaml** file. Open the file and to the list of operator container environment variables, add a variable with the name **PGO_TARGET_NAMESPACES**, and for the value enter the desired namespaces in a double-quoted, comma-separated list. For example:

```
spec:
  {{- include "install.imagePullSecrets" . | indent 6 }}
  serviceAccountName: {{ include "install.serviceAccountName" . }}
  containers:
    - name: operator
      image: {{ required ".Values.controllerImages.cluster is required" .Values.controllerImages.cluster | quote }}
      env:
        - name: PGO_TARGET_NAMESPACES
          value: "namespace-one, namespace-two, namespace-three"
```

However, if the **singleNamespace** setting is set to **true**, then a Role and RoleBinding will be created instead, allowing Crunchy Postgres for Kubernetes to only manage PostgreSQL clusters in the same namespace utilized when installing the Crunchy Postgres for Kubernetes Helm chart.

High Availability

Starting in CPK 5.7, the operator can run in a typical hot/cold high availability configuration. When enabled, one pod will be the leader, while others wait to become the leader should the current leader fail. This capability is controlled by the **pgoControllerLeaseName** value in the **values.yaml** file. That value names the [Lease](#) object used to elect a leader of the PGO deployment via the **PGO_CONTROLLER_LEASE_NAME** environment variable. The default is **cpk-leader-election-lease**, so you can achieve high availability by setting **replicas** greater than one.

If you wish to disable this capability, set **pgoControllerLeaseName** to an empty value and **replicas** to 1.

Health Probes

Starting in CPK 5.7, the operator has the ability to perform liveness and readiness health probes. These probes are set on the operator Deployment and are enabled by default with the following settings:

```
livenessProbe:
  httpGet:
    path: /readyz
    port: 8081
  initialDelaySeconds: 15
  periodSeconds: 20
readinessProbe:
  httpGet:
    path: /healthz
    port: 8081
```

```
initialDelaySeconds: 5
periodSeconds: 10
```

To disable these probes, simply remove them from the operator Deployment.

Install

Once you have configured the Helm chart according to your specific needs, it can then be installed using **helm**:

```
helm install $NAME -n $NAMESPACE helm/install
```

Automated Upgrade Checks

By default, Crunchy Postgres for Kubernetes will automatically check for updates to itself and software components by making a request to a URL. If Crunchy Postgres for Kubernetes detects there are updates available, it will print them in the logs. As part of the check, Crunchy Postgres for Kubernetes will send aggregated, anonymized information about the current deployment to the endpoint. An upcoming release will allow for Crunchy Postgres for Kubernetes to opt-in to receive and apply updates to software components automatically.

Crunchy Postgres for Kubernetes will check for updates upon startup and once every 24 hours. Any errors in checking will have no impact on the operation of Crunchy Postgres for Kubernetes. To disable the upgrade check, you can set the **disable_check_for_upgrades** value in the Helm chart to **true**.

For more information about collected data, see the Crunchy Data [collection notice](#).

Uninstall

To uninstall Crunchy Postgres for Kubernetes, remove all your PostgresCluster objects, then use the **helm uninstall** command:

```
helm uninstall $NAME -n $NAMESPACE
```

Helm [leaves the CRDs][helm-crd-limits] in place. You can remove them with **kubectl delete**

```
kubectl delete -f helm/install/crds
```

The Crunchy Container Registry

Installing directly from the registry

Crunchy Data hosts an OCI registry that **helm** can use directly. (Not all **helm** commands support OCI registries. For more information on which commands can be used, see [the Helm documentation](#).)

You can install Crunchy Postgres for Kubernetes directly from the registry using the **helm install** command:

```
helm install pgo oci://registry.developers.crunchydata.com/crunchydata/pgo
```

Or to see what values are set in the default `values.yaml` before installing, you could run a `helm show` command just as you would with any other registry:

```
helm show values oci://registry.developers.crunchydata.com/crunchydata/pgo
```

Downloading from the registry

Rather than deploying directly from the Crunchy registry, you can instead use the registry as the source for the Helm chart.

To do so, download the latest Helm chart from the Crunchy Container Registry:

```
helm pull oci://registry.developers.crunchydata.com/crunchydata/pgo
```

Once the Helm chart has been downloaded, uncompress the bundle

```
tar -xvf pgo-5.x.y.tgz
```

And from there, you can follow the instructions above on setting the [Configuration](#) and installing a local Helm chart.

Next Step: Create a Postgres Cluster

Now that you've installed Crunchy Postgres for Kubernetes, you're ready to Create a Postgres Cluster.

Next Step: Install Monitoring

No installation of Crunchy Postgres for Kubernetes is complete without monitoring! See our Tutorial on installing monitoring with Helm for details.

OperatorHub

Installing Crunchy Postgres for Kubernetes Using OperatorHub on OpenShift

Crunchy Postgres for Kubernetes can be installed on OpenShift through the OperatorHub point-and-click experience. Under Operators > OperatorHub, search for Crunchy and you'll find Marketplace, Certified and Community installers. Choose the installer that fits your needs and consider installing in a specific [namespace](#).

Registering your installation

OperatorHub installers come with a registration requirement. Users who register their installations will experience uninterrupted Crunchy Postgres for Kubernetes service during upgrades. Registration is achieved by visiting our [token creation page](#) with a Crunchy Data account.

If you already are a Crunchy Customer and have a Crunchy Account, use your Access Portal credentials to [log in here](#). If you are not a customer, [request an account](#).

Installing your Token

To obtain your token for Crunchy Postgres for Kubernetes, go to the [token creation page](#).

Once you have your token, create a file called `cpk_token` and paste the token into the file. Use `cpk_token` to create a Secret, and then restart the Crunchy Postgres for Kubernetes Deployment.

```
oc create secret generic cpk-registration --from-file=cpk_token -n $NAMESPACE
oc rollout restart deployment pgo -n $NAMESPACE
```

And that's it! Your installation is now fully enabled.

How Registration Affects Your Installation

OperatorHub installers require a registration token to upgrade from the installed version of Postgres. Once you apply a token to your installation, the token will be internally validated by the operator. Token validation does not require an internet connection.

Without a token, existing Postgres clusters will continue running uninterrupted. You will be able to create and destroy them, but you won't be able to upgrade existing Postgres clusters until you complete the registration process.

Registration Events

If your Crunchy Postgres for Kubernetes installation is properly registered, you will not see any registration-related events. However, if you have not yet registered, certain events may be generated for each PostgresCluster.

For instance, an event such as the following will be generated for any PostgresCluster managed by an unregistered installation:

```
Crunchy Postgres for Kubernetes requires registration for upgrades. Register now to be ready for your next upgrade. See https://access.crunchydata.com/register-cpk for tails.
```

This warning event simply indicates that registration will be required when upgrading.

If you are seeing this event, please be sure to register your installation as soon as possible.

Additional events will then be generated indicating successful (or unsuccessful) registration. For instance, the following informational event will be generated once you have successfully registered your installation:

```
Thank you for registering your installation of Crunchy Postgres for Kubernetes.
```

FAQ

Q: Your containers are build on UBI-8, will they work on my RHEL-9 hosts?

A: Crunchy Postgres for Kubernetes is certified by Red Hat, ensuring it meets stringent security and compatibility standards. This certification encompasses our use of containers, specifically adhering to the "Container image requirements" set

forth by Red Hat. According to Red Hat's guidelines, container images must use a Universal Base Image (UBI) provided by Red Hat. The version of the UBI base image must be supported on the RHEL version undergoing certification. Our UBI 8 containers fully comply with Red Hat's requirements. The Red Hat Enterprise Linux Container Compatibility Matrix confirms that UBI 8 containers are supported across all host types, including RHEL 7, RHEL 8, and RHEL 9. This means our containers are supported on both older RHEL hosts (e.g., RHEL 7 for older OpenShift versions) and newer RHEL hosts (e.g., RHEL 9 on OpenShift v4.13+).

Q: What OperatorHub installers require a registration token?

A: All OperatorHub installers currently require a registration token.

Q: What happens if I don't install a token?

A: A valid token is required to be able to perform any upgrades after Crunchy Postgres for Kubernetes is installed. Your running Postgres instances will remain unaffected. New installs are also unaffected.

Q: What if my Crunchy Postgres for Kubernetes clusters can't establish an internet connection?

A: The token's validation is processed internally within Crunchy Postgres for Kubernetes. An active internet connection isn't needed for this verification process.

Q: How do I get a token?

A: To obtain a token, head to the Crunchy Data Token Portal at <https://tokens.crunchydata.com>. You'll be prompted to either log in via your Access Portal credentials or initiate an account request. You can also manage and view your existing tokens at this site.

Q: Do I need a token to install from the Red Hat Marketplace?

A: The token is only required to upgrade a running instance of Crunchy Postgres. Installs do not currently require a token. However, obtaining one as part of the install process is advised, so you do not need to worry about it when it's time to upgrade.

Q: What kinds of upgrades require a token?

A: Postgres introduces new features in a new major version once each year. With Crunchy Postgres for Kubernetes, you choose when to apply these upgrades, and they do require a token. Bug fixes and minor version upgrades for Postgres happen automatically when upgrading the operator, which also requires a token.

Q: Will I need a new token for each upgrade?

A: No. A token simply unlocks the ability to upgrade, and is not tied to a specific version of Crunchy Postgres for Kubernetes.

Q: When does this take effect?

A: The token system and website have been launched alongside the release of Crunchy Postgres for Kubernetes version 5.5. It's important to note that to upgrade beyond version 5.5, you will require a token. For instance, if you have Crunchy Postgres for Kubernetes version 5.5 installed, you will need a token when upgrading to version 5.5.1 or version 5.6.

Q: What versions of Postgres are published to the Crunchy Developer Portal registry and the Red Hat certified image catalog for use with OperatorHub installations?

A: In accordance with the [Crunchy Developer Program](#), the latest two major versions of Postgres are published to the Crunchy Data Developer registry and the Red Hat certified image catalog.

Private Registries

Crunchy Postgres for Kubernetes can use containers that are stored in private registries. There are a variety of techniques that are used to load containers from private registries, including [image pull secrets](#). This guide will demonstrate how to install Crunchy Postgres for Kubernetes and deploy a Postgres cluster using the [Crunchy Data Customer Portal](#) registry as an example.

Create an Image Pull Secret

The Kubernetes documentation provides several methods for creating [image pull secrets](#). You can choose the method that is most appropriate for your installation. You will need to create image pull secrets in the namespace that Crunchy Postgres for Kubernetes is deployed and in each namespace where you plan to deploy Postgres clusters.

For example, to create an image pull secret for accessing the Crunchy Data Customer Portal image registry in the **postgres-operator** namespace, you can execute the following commands:

```
kubectl create ns postgres-operator

kubectl create secret docker-registry crunchy-regcred -n postgres-operator --docker-server=registry.crunchydata.com --docker-username=$YOUR_USERNAME --docker-email=$YOUR_EMAIL --docker-password=$YOUR_PASSWORD
```

This creates an image pull secret named **crunchy-regcred** in the **postgres-operator** namespace.

Install Crunchy Postgres for Kubernetes from a Private Registry

To install Crunchy Postgres for Kubernetes from a private registry, you will need to set an image pull secret on the installation manifest.

Kustomize

When using the Kustomize install method, you can set up the image pull secret by adding a patch to the **kustomize/install/default/kustomization.yaml** manifest. In this example, we will use the **crunchy-regcred** secret that we created earlier:

```
patches:
- target: { group: apps, version: v1, kind: Deployment, name: pgo }
  patch: |-
    - path: /spec/template/spec/imagePullSecrets
      op: add
      value:
        - name: crunchy-regcred
```

If you are using a version of `kubect1` prior to `v1.21.0`, you will have to create an explicit patch file named `install-ops.yaml`:

```
-path: /spec/template/spec/imagePullSecrets
  op: add
  value:
    - name: crunchy-regcred
```

and add the following to the manifest:

```
patchesJson6902:
  - target: { group: apps, version: v1, kind: Deployment, name: pgo }
    path: install-ops.yaml
```

You can then install Crunchy Postgres for Kubernetes from the private registry using the standard installation procedure, e.g.:

```
kubect1 apply --server-side -k kustomize/install/default
```

Helm

To set up an image pull secret when using the Helm installer, you need to edit the `values.yaml` file, adding the name of the image pull secret to the `imagePullSecretNames` array:

```
#imagePullSecretNames is a list of secret names to use for pulling controller images.
# More info: https://kubernetes.io/docs/concepts/containers/images/#specifying-imagepullsecrets-on-a-pod
imagePullSecretNames: [crunchy-regcred]
```

You can then install Crunchy Postgres for Kubernetes from the private registry using the standard installation procedure, e.g.:

```
helm install $NAME -n $NAMESPACE helm/install
```

Deploy a Postgres cluster from a Private Registry

To deploy a Postgres cluster using images from a private registry, you will need to set the value of `spec.imagePullSecrets` on a `PostgresCluster` custom resource.

Kustomize

To deploy a Postgres cluster in the `postgres-operator` namespace, with an image pull secret containing credentials for the [Crunchy Data Customer Portal](#), you can use the following manifest:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  imagePullSecrets:
    - name: crunchy-regcred
  image: registry.crunchydata.com/crunchydata/crunchy-postgres:ubi9-17.5-2520
```

```

postgresVersion: 17
instances:
  - name: instance1
    dataVolumeClaimSpec:
      accessModes:
        - 'ReadWriteOnce'
      resources:
        requests:
          storage: 1Gi
backups:
  pgbackrest:
    image: registry.crunchydata.com/crunchydata/crunchy-pgbackrest:ubi9-2.54.2-2520
    repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
              - 'ReadWriteOnce'
            resources:
              requests:
                storage: 1Gi

```

Helm

To deploy a Postgres cluster with Helm, you wouldn't edit the `PostgresCluster` manifest directly, but would edit the `values.yaml` file in the chart, adding the name of the image pull secret to the `imagePullSecrets` array:

```

#imagePullSecrets references Secrets that credentials for pulling image from
# private repositories
imagePullSecrets: [crunchy-regcred]

```

Feature Gate Installation Guide

This page provides an overview of the feature gates an administrator can enable or disable during installation of Crunchy Postgres for Kubernetes. If you've downloaded the installer from the [Crunchy Data Customer Portal](#), please refer to the customer guide there for alternative setup information.

Feature Gates Available in Crunchy Postgres for Kubernetes

PGO Feature Gate	Default setting	Since	Until
AppendCustomQueries	false	v5.5.0	-
AutoCreateUserSchema	false	v5.6.1	v5.6.2
AutoCreateUserSchema	true	v5.6.2	-
AutoGrowVolumes	false	v5.6.0	-
InstanceSidecars	false	v5.2.0	v5.8.0
InstanceSidecars	true	v5.8.0	-
OpenTelemetryLogs	false	v5.8.0	-
OpenTelemetryMetrics	false	v5.8.0	-
PGUpgradeCPUConcurrency	false	v5.6.4	v5.8.0
PGUpgradeCPUConcurrency	true	v5.8.0	-

PGBouncerSidecars	false	v5.2.0	-
TablespaceVolumes	false	v5.4.0	-
VolumeSnapshots	false	v5.7.0	-

¹ Use the values in the PGO Feature Gate column in place of FeatureName in the installation instructions below.

Helm

First, go to GitHub and [fork the Postgres Operator examples](#) repository, which contains the Crunchy Postgres for Kubernetes Helm installer.

To enable feature gates with Helm, find `helm/install/values.yaml` in the examples repository and uncomment the **features** key.

Add a key from the table above for each PGO Feature Gate you want to enable, and set the value to true. For example, you can enable disk auto-grow and custom queries for monitoring like this:

```
features:
  AutoGrowVolumes: true
  AppendCustomQueries: true
```

If you haven't installed the operator yet, run:

```
helm install $NAME -n $NAMESPACE helm/install
```

Otherwise, run:

```
helm upgrade $NAME -n $NAMESPACE helm/install
```

Kustomize

First, go to GitHub and [fork the Postgres Operator examples](#) repository, which contains the Crunchy Postgres for Kubernetes Kustomize installer.

PGO Feature Gates can be enabled with Kustomize by setting the `PGO_FEATURE_GATES` env variable in your container spec. In the `kustomize/install/default/kustomization.yaml` file in the examples repository you will see a section like this:

```
patches:
- patch: |-
  apiVersion: apps/v1
  kind: Deployment
  metadata:
    name: pgo
  spec:
    template:
      spec:
        containers:
        - name: operator
          env:
            - name: PGO_FEATURE_GATES
              value: ""
```

...which patches the operator Deployment, adding the `PGO_FEATURE_GATES` environment variable.

To turn on feature gates, set the value for `PGO_FEATURE_GATES` as "FeatureName=true,FeatureName2=true,FeatureName3=true",

where each FeatureName is the PGO Feature Gate you want to enable. You can list as many PGO Feature Gates as you need.

To apply the changes, run:

```
kubectl apply --server-side -k kustomize/install/default
```

OLM

After Crunchy Postgres for Kubernetes has been installed from OperatorHub, you can set feature gates by clicking on Installed Operators and selecting Crunchy Postgres for Kubernetes. From there, select Subscription and from the Actions dropdown menu select Edit Subscription. Scroll to the spec section and you can create a config block to set environment variables like this:

```
spec:
  config:
    env:
      - name: PGO_FEATURE_GATES
        value: "FeatureName=true, FeatureName2=true, FeatureName3=true"
```

...where each FeatureName is the PGO Feature Gate you want to enable. You can list as many PGO Feature Gates as you

need. After you've adjusted the Subscription to meet your needs, save it and observe that the environment variables in your PGO pod have updated.

Checking which Feature Gates are enabled

You can check what features are enabled by checking the logs when the operator pod is first deployed.

The logs include both the user-defined `PGO_FEATURE_GATES` environment variable and what feature gates are actually enabled. This way you can check both what you've set and what features are on by default (and haven't been explicitly disabled).

Tutorials

Ready to get started with [PGO](#), the [Postgres Operator](#) from [Crunchy Data](#)? Us too!

This tutorial covers several concepts around day-to-day life managing a Postgres cluster with PGO. While going through and looking at various "HOWTOs" with PGO, we will also cover concepts and features that will help you have a successful cloud native Postgres journey!

In this tutorial, you will learn:

- How to create a Postgres cluster
- How to connect to a Postgres cluster

- How to scale and create a high availability (HA) Postgres cluster
- How to resize your cluster
- How to set up proper disaster recovery and manage backups and restores
- How to apply software updates to Postgres and other components
- How to set up connection pooling
- How to delete your cluster

and more.

You will also see:

- How PGO helps your Postgres cluster achieve high availability
- How PGO can heal your Postgres cluster and ensure all objects are present and available
- How PGO sets up disaster recovery
- How to manage working with PGO in a single namespace or in a multi-namespace installation of PGO.

Let's get started!

Basic Setup

Setting up your environment

The first thing that you will need is a Kubernetes or Openshift environment running a supported version. You can see all of the versions in our documentation. You can deploy to your environment locally, in the cloud, or even run it via a managed Kubernetes offering.

You will also need to insure that you have a modern version of `git` installed locally, as well as `kubect1` installed and configured on your local workstation. You can install those from your OS's package manager. You can refer to the reference for [git](#) if you are not already familiar with it, or you need to install it by hand. You can also visit the [kubect1](#) reference for more information about how to install and use this tool.

Once you have your tools and environment set up, we can move on to installing Crunchy Postgres for Kubernetes.

Download the Examples

First, go to GitHub and [fork the Postgres Operator examples](#) repository:

<https://github.com/CrunchyData/postgres-operator-examples/fork>

Once you have forked this repository, you can download it to your working directory with a command similar to this:

```
cd <Your Working Directory>
YOUR_GITHUB_UN="$YOUR_GITHUB_USERNAME"
git clone --depth 1 "git@github.com:${YOUR_GITHUB_UN}/postgres-operator-examples.git"
```

For Powershell environments:

```
cd <Your Working Directory>
$env:YOUR_GITHUB_UN="YOUR_GITHUB_USERNAME"
git clone --depth 1 "git@github.com:$env:YOUR_GITHUB_UN/postgres-operator-examples.git"
```

With the examples repo cloned into your working directory, navigate (for example, `cd postgres-operator-examples`) to the top level folder of the repo. If you use `ls -lah` it should look something like this:

```
~/Code/Crunchy/postgres-operator-examples ls -lah
total 32
drwxr-xr-x  8 hippo  staff   256B May 22 14:27 .
drwxr-xr-x  9 hippo  staff   288B Jun 29 13:59 ..
drwxr-xr-x 14 hippo  staff   448B May  9 12:00 .git
drwxr-xr-x  3 hippo  staff    96B Jul 19  2022 .github
-rw-r--r--  1 hippo  staff   11K Apr  3 12:17 LICENSE.md
-rw-r--r--@ 1 hippo  staff   1.1K May  9 11:27 README.md
drwxr-xr-x  4 hippo  staff   128B Jul 19  2022 helm
drwxr-xr-x 12 hippo  staff   384B Jul 19  2022 kustomize
```

Once you have your local environment set up, we can press onwards to installing Crunchy Postgres for Kubernetes...

Install Crunchy Postgres for Kubernetes

Our next task is to install Crunchy Postgres for Kubernetes into a namespace in Kubernetes. This example uses a default namespace of `postgres-operator`. However, you can install it in other namespaces or even cluster wide if you need. You can read more about that in our advanced install guides.

First, we need to set up the namespace that we are going to use. Use this command to create the default namespace:

```
kubectl apply -k kustomize/install/namespace
```

Next, you will need to install the various containers and configuration that makes up Crunchy Postgres for Kubernetes. Here is the command to do that:

```
kubectl apply --server-side -k kustomize/install/default
```

To check on the status of your installation, you can run the following command:

```
kubectl -n postgres-operator get pods --selector=postgres-operator.crunchydata.com/control-plane=postgres-operator --field-selector=status.phase=Running
```

If the PGO Pod is healthy, you should see output similar to:

NAME	READY	STATUS	RESTARTS	AGE
postgres-operator-9dd545d64-t4h8d	1/1	Running	0	3s

Now that we have installed all of the supporting containers and configuration, it's time to roll our sleeves up and set up a Postgres cluster...

Create a Postgres Cluster

If you came here through the quickstart, you may have already created a cluster. If you created a cluster by using the example in the `kustomize/postgres` directory, feel free to skip to connecting to a cluster or follow our instructions on deleting your cluster for a fresh start.

Use Kustomize to Create a Postgres Cluster

Creating a Postgres cluster is pretty simple from a [fork of our examples repository](#). Using the example in the `kustomize/postgres` directory, all we have to do is run:

```
kubectl apply -k kustomize/postgres
```

and PGO will create a simple Postgres cluster named `hippo` in the `postgres-operator` namespace. You can track the status of your Postgres cluster using `kubectl describe` on the `postgresclusters.postgres-operator.crunchydata.com` custom resource:

```
kubectl -n postgres-operator describe postgresclusters.postgres-operator.crunchydata.com hippo
```

and you can track the state of the Postgres Pod using the following command:

```
kubectl -n postgres-operator get pods --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instance
```

Use Helm to Create a Postgres Cluster

Creating a Postgres cluster is pretty simple from a [fork of our examples repository](#).

Let's assume that you've installed Crunchy Postgres for Kubernetes from the examples repository like this:

```
helm install cpk helm/install --namespace postgres-operator --create-namespace
```

You can create a Postgres Cluster in the `postgres-operator` namespace with a command like this:

```
helm install hippo helm/postgres --namespace postgres-operator
```

and you can track the state of the Postgres Pod using the following command:

```
kubectl -n postgres-operator get pods --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instance
```

What Happens When a Postgres Cluster is Created

Crunchy Postgres for Kubernetes created a Postgres cluster based on the information provided to it from either the Kustomize manifests in the `kustomize/postgres` directory or the Helm chart in the `helm/postgres` directory.

Let's better understand what happened by inspecting the `kustomize/postgres/postgres.yaml` file:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
```

```
spec:
  postgresVersion: 17
  instances:
    - name: instance1
      dataVolumeClaimSpec:
        accessModes:
          - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - "ReadWriteOnce"
              resources:
                requests:
                  storage: 1Gi
```

When we ran the `kubectl apply` command earlier, what we did was create a `PostgresCluster` custom resource in Kubernetes. PGO detected that we added a new `PostgresCluster` resource and started to create all the objects needed to run Postgres in Kubernetes!

What else happened? PGO read the value from `metadata.name` to provide the Postgres cluster with the name `hippo`. Additionally, PGO knew which containers to use for Postgres and pgBackRest by looking at the values in `spec.image` and `spec.backups.pgbackrest.image` respectively. The value in `spec.postgresVersion` is important as it will help PGO track which major version of Postgres you are using.

PGO knows how many Postgres instances to create through the `spec.instances` section of the manifest. While `name` is optional, we opted to give it the name `instance1`. We could have also created multiple replicas and instances during cluster initialization, but we will cover that more when we discuss how to scale and create a HA Postgres cluster.

A very important piece of your `PostgresCluster` custom resource is the `dataVolumeClaimSpec` section. This describes the storage that your Postgres instance will use. It is modeled after the [Persistent Volume Claim](#). If you do not provide a `spec.instances.dataVolumeClaimSpec.storageClassName`, then the default storage class in your Kubernetes environment is used.

As part of creating a Postgres cluster, we also specify information about our backup archive. PGO uses [pgBackRest](#), an open source backup and restore tool designed to handle terabyte-scale backups. As part of initializing our cluster, we can specify where we want our backups and archives ([write-ahead logs or WAL](#)) stored. We will talk about this portion of the `PostgresCluster` spec in greater depth in the disaster recovery section of this tutorial, and also see how we can store backups in Amazon S3, Google GCS, and Azure Blob Storage.

Troubleshooting

PostgreSQL / pgBackRest Pods Stuck in Pending Phase

The most common occurrence of this is due to PVCs not being bound. Ensure that you have set up your storage options correctly in any `volumeClaimSpec`. You can always update your settings and reapply your changes with `kubectl apply`

Also ensure that you have enough persistent volumes available: your Kubernetes administrator may need to provision more.

If you are on OpenShift, you may need to set `spec.openshift` to `true`.

Next Steps

We're up and running -- now let's connect to our Postgres cluster!

Connect to a Postgres Cluster

It's one thing to create a Postgres cluster; it's another thing to connect to it. Let's explore how PGO makes it possible to connect to a Postgres cluster!

Background: Services, Secrets, and TLS

PGO creates a collection of Kubernetes [Services](#) to provide stable endpoints for connecting to your Postgres databases. These endpoints make it easy to provide a consistent way for your application to maintain connectivity to your data. To inspect what services are available, you can run the following command:

```
kubectl -n postgres-operator get svc --selector=postgres-operator.crunchydata.com/cluster=hippo
```

which will yield something similar to:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
hippo-ha	ClusterIP	10.103.73.92	<none>	5432/TCP	3h14m
hippo-ha-config	ClusterIP	None	<none>	<none>	3h14m
hippo-pods	ClusterIP	None	<none>	<none>	3h14m
hippo-primary	ClusterIP	None	<none>	5432/TCP	3h14m
hippo-replicas	ClusterIP	10.98.110.215	<none>	5432/TCP	3h14m

You do not need to worry about most of these Services, as they are used to help manage the overall health of your Postgres cluster. For the purposes of connecting to your database, the Service of interest is called `hippo-primary`. Thanks to PGO, you do not need to even worry about that, as that information is captured within a Secret!

When your Postgres cluster is initialized, PGO will bootstrap a database and create a Postgres user that your application can use to access the database. This information is stored in a Secret named with the pattern `<cluster-Name>-pguser-<userName>`. For our `hippo` cluster, this Secret is called `hippo-pguser-hippo`. This Secret contains the information you need to connect your application to your Postgres database:

- `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 Postgres instance.
- `port`: The port that the database is listening on.

- **uri**: A [PostgreSQL connection URI](#) that provides all the information for logging into the Postgres database.
- **jdbc-uri**: A [PostgreSQL JDBC connection URI](#) that provides all the information for logging into the Postgres database via the JDBC driver.

All connections are over TLS. PGO provides its own certificate authority (CA) to allow you to securely connect your applications to your Postgres clusters. This allows you to use the [verify-full "SSL mode"](#) of Postgres, which provides eavesdropping protection and prevents MITM attacks. You can also choose to bring your own CA, which is described later in this tutorial in the Customize Cluster section.

Modifying Service Type, NodePort Value and Metadata

By default, PGO deploys Services with the **ClusterIP** Service type. Based on how you want to expose your database, you may want to modify the Services to use a different [Service type](#) and [NodePort value](#).

You can modify the Services that PGO manages from the following attributes:

- **spec.service** - this manages the Service for connecting to a Postgres primary.
- **spec.replicaService** - this manages the Service for connecting to a Postgres replica.
- **spec.proxy.pgBouncer.service** - this manages the Service for connecting to the PgBouncer connection pooler.

For example, say you want to set the Postgres primary to use a **NodePort** service, a specific **nodePort** value, and set a specific annotation and label, you would add the following to your manifest:

```
spec:
  service:
    metadata:
      annotations:
        my-annotation: value1
      labels:
        my-label: value2
    type: NodePort
    nodePort: 32000
```

For our **hippo** cluster, you would see the Service type and nodePort modification as well as the annotation and label. For example:

```
kubectl -n postgres-operator get svc --selector=postgres-operator.crunchydata.com/cluster=hippo
```

will yield something similar to:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
hippo-ha	NodePort	10.105.57.191	<none>	5432:32000/TCP	48s
hippo-ha-config	ClusterIP	None	<none>	<none>	48s
hippo-pods	ClusterIP	None	<none>	<none>	48s
hippo-primary	ClusterIP	None	<none>	5432/TCP	48s
hippo-replicas	ClusterIP	10.106.18.99	<none>	5432/TCP	48s

and the top of the output from running

```
kubectl -n postgres-operator describe svc hippo-ha
```

will show our custom annotation and label have been added:

```
NAME:          hippo-ha
Namespace:     postgres-operator
Labels:        my-label=value2
               postgres-operator.crunchydata.com/cluster=hippo
               postgres-operator.crunchydata.com/patroni=hippo-ha
Annotations:   my-annotation: value1
```

Note that setting the `nodePort` value is not allowed when using the (default) `ClusterIP` type, and it must be in-range and not otherwise in use or the operation will fail. Additionally, be aware that any annotations or labels provided here will win in case of conflicts with any annotations or labels a user configures elsewhere.

Finally, if you are exposing your Services externally and are relying on TLS verification, you will need to use the custom TLS features of PGO).

Connect via `psql` in the Terminal

Connect Directly

If you are on the same network as your PostgreSQL cluster, you can connect directly to it using the following command:

```
psql $(kubectl -n postgres-operator get secrets hippo-pguser-hippo -o go-template='{{.data.uri | base64decode}}')
```

Connect Using a Port-Forward

In a new terminal, create a port forward. If you are using Bash, you can run the following commands:

```
PG_CLUSTER_PRIMARY_POD=$(kubectl get pod -n postgres-operator -o name -l postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/role=master)
kubectl -n postgres-operator port-forward "${PG_CLUSTER_PRIMARY_POD}" 5432:5432
```

For Powershell environments:

```
$env:PG_CLUSTER_PRIMARY_POD=(kubectl get pod -n postgres-operator -o name -l postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/role=master)
kubectl -n postgres-operator port-forward "$env:PG_CLUSTER_PRIMARY_POD" 5432:5432
```

Establish a connection to the PostgreSQL cluster. If you are using Bash, you can run:

```
PG_CLUSTER_USER_SECRET_NAME=hippo-pguser-hippo

PGPASSWORD=$(kubectl get secrets -n postgres-operator "${PG_CLUSTER_USER_SECRET_NAME}" -o go-template='{{.data.password | base64decode}}') \
PGUSER=$(kubectl get secrets -n postgres-operator "${PG_CLUSTER_USER_SECRET_NAME}" -o go-template='{{.data.user | base64decode}}') \
PGDATABASE=$(kubectl get secrets -n postgres-operator "${PG_CLUSTER_USER_SECRET_NAME}" -o go-template='{{.data.dbname | base64decode}}') \
psql -h localhost
```

For Powershell environments:

```
$env:PG_CLUSTER_USER_SECRET_NAME="hippo-pguser-hippo"

$env:PGPASSWORD=(kubectl get secrets -n postgres-operator "$env:PG_CLUSTER_USER_SE-
CRET_NAME" -o go-template='{{.data.password | base64decode}}')
$env:PGUSER=(kubectl get secrets -n postgres-operator "$env:PG_CLUSTER_USER_SE-
CRET_NAME" -o go-template='{{.data.user | base64decode}}')
$env:PGDATABASE=(kubectl get secrets -n postgres-operator "$env:PG_CLUSTER_USER_SE-
CRET_NAME" -o go-template='{{.data.dbname | base64decode}}')
psql -h localhost
```

Connecting With pgAdmin

Crunchy Postgres for Kubernetes also provides a pgAdmin image for users who prefer working with a graphical user interface. For more information, see our documentation on pgAdmin.

Connect an Application

For this tutorial, we are going to connect [Keycloak](#), an open source identity management application. Keycloak can be deployed on Kubernetes and is backed by a Postgres database. While we provide an [example of deploying Keycloak and a PostgresCluster](#) in the [Postgres Operator examples](#) repository, the manifest below deploys it using our **hippo** cluster that is already running:

```
kubectl apply --filename=- <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: keycloak
  namespace: postgres-operator
  labels:
    app.kubernetes.io/name: keycloak
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: keycloak
  template:
    metadata:
      labels:
        app.kubernetes.io/name: keycloak
    spec:
      containers:
        - image: quay.io/keycloak/keycloak:latest
          args: ["start-dev"]
          name: keycloak
          env:
            - name: DB_VENDOR
              value: "postgres"
            - name: DB_ADDR
              valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: host } }
            - name: DB_PORT
              valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: port } }
            - name: DB_DATABASE
              valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: dbname } }
            - name: DB_USER
              valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: user } }
            - name: DB_PASSWORD
              valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: password } }
            - name: KEYCLOAK_ADMIN
```

```

      value: "admin"
    - name: KEYCLOAK_ADMIN_PASSWORD
      value: "admin"
    - name: KC_PROXY
      value: "edge"
  ports:
    - name: http
      containerPort: 8080
    - name: https
      containerPort: 8443
  readinessProbe:
    httpGet:
      path: /realms/master
      port: 8080
  restartPolicy: Always
EOF

```

Notice this part of the manifest:

```

-name: DB_ADDR
  valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: host } }
- name: DB_PORT
  valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: port } }
- name: DB_DATABASE
  valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: dbname } }
- name: DB_USER
  valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: user } }
- name: DB_PASSWORD
  valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: password } }

```

The above manifest shows how all of these values are derived from the `hippo-pguser-hippo` Secret. This means that we do not need to know any of the connection credentials or have to insecurely pass them around -- they are made directly available to the application!

Using this method, you can tie an application directly into your GitOps pipeline that connects to Postgres without any prior knowledge of how PGO will deploy Postgres: all of the information your application needs is propagated into the Secret!

Next Steps

Now that we have seen how to connect an application to a cluster, let's learn how to create a high availability Postgres cluster!

Connection Pooling

Connection pooling can be helpful for scaling and maintaining overall availability between your application and the database. PGO helps facilitate this by supporting the [PgBouncer](#) connection pooler and state manager.

Let's look at how we can add a connection pooler and connect it to our application!

Adding a Connection Pooler

We will explore adding a connection pooler using the `kustomize/keycloak` example in the [Postgres Operator examples](#) repository.

Connection poolers are added using the `spec.proxy` section of the custom resource. Currently, the only connection pooler supported is [PgBouncer](#).

You can add a PgBouncer connection pooler by providing the `spec.proxy.pgBouncer` attribute and leaving it empty. In the `kustomize/keycloak/postgres.yaml` file, add the following YAML to the spec:

```
proxy:
  pgBouncer: {}
```

(You can also find an example of this in the `kustomize/examples/high-availability` example).

Save your changes and run:

```
kubectl apply -k kustomize/keycloak
```

PGO will detect the change and create a new PgBouncer Deployment!

That was fairly easy to set up, so now let's look at how we can connect our application to the connection pooler.

Connecting to a Connection Pooler

When a connection pooler is deployed to the cluster, PGO adds additional information to the user Secrets to allow for applications to connect directly to the connection pooler. Recall that in this example, our user Secret is called `keycloakdb-pguser-keycloakdb`. Describe the user Secret:

```
kubectl -n postgres-operator describe secrets keycloakdb-pguser-keycloakdb
```

You should see that there are several new attributes included in this Secret that allow for you to connect to your Postgres instance via the connection pooler:

- **pgbouncer-host**: The name of the host of the PgBouncer connection pooler. This references the [Service](#) of the PgBouncer connection pooler.
- **pgbouncer-port**: The port that the PgBouncer connection pooler is listening on.
- **pgbouncer-uri**: A [PostgreSQL connection URI](#) that provides all the information for logging into the Postgres database via the PgBouncer connection pooler.
- **pgbouncer-jdbc-uri**: A [PostgreSQL JDBC connection URI](#) that provides all the information for logging into the Postgres database via the PgBouncer connection pooler using the JDBC driver. Note that by default, the connection string disables JDBC managing prepared transactions for [optimal use with PgBouncer](#).

Open up the file in `kustomize/keycloak/keycloak.yaml`. Update the `DB_ADDR` and `DB_PORT` values to be the following:

```
-name: DB_ADDR
  valueFrom: { secretKeyRef: { name: keycloakdb-pguser-keycloakdb, key: pgbouncer-host } }
- name: DB_PORT
  valueFrom: { secretKeyRef: { name: keycloakdb-pguser-keycloakdb, key: pgbouncer-port } }
```

This changes Keycloak's configuration so that it will now connect through the connection pooler.

Apply the changes:

```
kubectl apply -k kustomize/keycloak
```

Kubernetes will detect the changes and begin to deploy a new Keycloak Pod. When it is completed, Keycloak will now be connected to Postgres via the PgBouncer connection pooler!

TLS

PGO deploys every cluster and component over TLS. This includes the PgBouncer connection pooler. If you are using your own custom TLS setup, you will need to provide a Secret reference for a TLS key / certificate pair for PgBouncer in `spec.proxy.pgBouncer.customTLSSecret`.

Your TLS certificate for PgBouncer should have a Common Name (CN) setting that matches the PgBouncer Service name. This is the name of the cluster suffixed with `-pgbouncer`. For example, for our `hippo` cluster this would be `hippo-pgbouncer`. For the `keycloakdb` example, it would be `keycloakdb-pgbouncer`.

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

```
data:
  ca.crt: $VALUE
  tls.crt: $VALUE
  tls.key: $VALUE
```

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

```
kubectl create secret generic -n postgres-operator keycloakdb-pgbound-
er.tls --from-file=ca.crt=ca.crt --from-file=tls.key=keycloakdb-pgbound-
er.key --from-file=tls.crt=keycloakdb-pgbouncer.crt
```

You can specify the custom TLS Secret in the `spec.proxy.pgBouncer.customTLSSecret.name` field in your `postgrescluster.postgres-operator.crunchydata.com` custom resource, e.g.:

```
spec:
  proxy:
    pgBouncer:
      customTLSSecret:
        name: keycloakdb-pgbouncer.tls
```

Customizing

The PgBouncer connection pooler is highly customizable, both from a configuration and Kubernetes deployment standpoint. Let's explore some of the customizations that you can do!

Configuration

[PgBouncer configuration](#) can be customized through `spec.proxy.pgBouncer.config`. After making configuration changes, PGO will roll them out to any PgBouncer instance and automatically issue a "reload".

There are several ways you can customize the configuration:

- `spec.proxy.pgBouncer.config.global`: Accepts key-value pairs that apply changes globally to PgBouncer.
- `spec.proxy.pgBouncer.config.databases`: Accepts key-value pairs that represent PgBouncer [database definitions](#).
- `spec.proxy.pgBouncer.config.users`: Accepts key-value pairs that represent [connection settings applied to specific users](#).
- `spec.proxy.pgBouncer.config.files`: Accepts a list of files that are mounted in the `/etc/pgbouncer` directory and loaded before any other options are considered using PgBouncer's [include directive](#).

For example, to set the connection pool mode to `transaction`, you would set the following configuration:

```
spec:
  proxy:
    pgBouncer:
      config:
        global:
          pool_mode: transaction
```

For a reference on [PgBouncer configuration](#) please see:

<https://www.pgbouncer.org/config.html>

Replicas

PGO deploys one PgBouncer instance by default. You may want to run multiple PgBouncer instances to have some level of redundancy, though you still want to be mindful of how many connections are going to your Postgres database!

You can manage the number of PgBouncer instances that are deployed through the `spec.proxy.pgBouncer.replicas` attribute.

Resources

You can manage the CPU and memory resources given to a PgBouncer instance through the `spec.proxy.pgBouncer.resources` attribute. The layout of `spec.proxy.pgBouncer.resources` should be familiar: it follows the same pattern as the standard Kubernetes structure for setting [container resources](#).

For example, let's say we want to set some CPU and memory limits on our PgBouncer instances. We could add the following configuration:

```
spec:
  proxy:
    pgBouncer:
      resources:
        limits:
          cpu: 200m
          memory: 128Mi
```

As PGO deploys the PgBouncer instances using a [Deployment](#) these changes are rolled out using a rolling update to minimize disruption between your application and Postgres instances!

Annotations / Labels

You can apply custom annotations and labels to your PgBouncer instances through the `spec.proxy.pgBouncer.metadata.annotations` and `spec.proxy.pgBouncer.metadata.labels` attributes respectively. Note that any changes to either of these two attributes take precedence over any other custom labels you have added.

Pod Anti-Affinity / Pod Affinity / Node Affinity

You can control the [pod anti-affinity, pod affinity, and node affinity](#) through the `spec.proxy.pgBouncer.affinity` attribute, specifically:

- `spec.proxy.pgBouncer.affinity.nodeAffinity`: controls node affinity for the PgBouncer instances.
- `spec.proxy.pgBouncer.affinity.podAffinity`: controls Pod affinity for the PgBouncer instances.
- `spec.proxy.pgBouncer.affinity.podAntiAffinity`: controls Pod anti-affinity for the PgBouncer instances.

Each of the above follows the [standard Kubernetes specification for setting affinity](#).

For example, to set a preferred Pod anti-affinity rule for the `kustomize/keycloak` example, you would want to add the following to your configuration:

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

Tolerations

You can deploy PgBouncer instances to [Nodes with Taints](#) by setting [Tolerations](#) through `spec.proxy.pgBouncer.tolerations`. This attribute follows the Kubernetes standard tolerations layout.

For example, if there were a set of Nodes with a Taint of `role=connection-poolers:NoSchedule` that you want to schedule your PgBouncer instances to, you could apply the following configuration:

```
spec:
  proxy:
    pgBouncer:
      tolerations:
        - effect: NoSchedule
          key: role
```

```
operator: Equal
value: connection-poolers
```

Note that setting a toleration does not necessarily mean that the PgBouncer instances will be assigned to Nodes with those taints. [Tolerations act as a key: they allow for you to access Nodes](#). If you want to ensure that your PgBouncer instances are deployed to specific nodes, you need to combine setting tolerations with node affinity.

Pod Spread Constraints

Besides using affinity, anti-affinity and tolerations, you can also set [Topology Spread Constraints](#) through `spec.proxy.pgBouncer.topologySpreadConstraints`. This attribute follows the Kubernetes standard topology spread constraint layout.

For example, since each of our pgBouncer Pods will have the standard `postgres-operator.crunchydata.com/role: pgbouncer` Label set, we can use this Label when determining the `maxSkew`. In the example below, since we have 3 nodes with a `maxSkew` of 1 and we've set `whenUnsatisfiable` to `ScheduleAnyway`, we should ideally see 1 Pod on each of the nodes, but our Pods can be distributed less evenly if other constraints keep this from happening.

```
proxy:
  pgBouncer:
    replicas: 3
    topologySpreadConstraints:
      - maxSkew: 1
        topologyKey: my-node-label
        whenUnsatisfiable: ScheduleAnyway
        labelSelector:
          matchLabels:
            postgres-operator.crunchydata.com/role: pgbouncer
```

If you want to ensure that your PgBouncer instances are deployed more evenly (or not deployed at all), you need to update `whenUnsatisfiable` to `DoNotSchedule`.

Administration

PgBouncer provides an admin console that can be utilized to obtain connection pooler statistics, and to control the PgBouncer instance via various process control commands.

To access the admin console, you will need to configure the PgBouncer `admin_users` setting as follows:

```
proxy:
  pgBouncer:
    config:
      global:
        admin_users: _crunchypgbouncer
```

With this setting in place, the `_crunchypgbouncer` system account will now be allowed to connect to the PgBouncer admin console.

To obtain the password for the `_crunchypgbouncer` user, you can leverage the `<clusterName>-pgbouncer` Secret.

```
kubectl get secret hippo-pgbouncer --template='{{index .data "pgbouncer-password" | base64decode}}'
```

For Powershell environments, you need to escape the double quotes around "pgbouncer-password":

```
kubectl get secret hippo-pgbouncer --template='{{index .data \"pgbouncer-password\" | base64decode }}'
```

And from there you can connect to the **pgbouncer** database and access the PgBouncer admin console:

```
$psql -U _crunchypgbouncer -h hippo-pgbouncer.postgres-operator.svc pgbouncer
Password for user _crunchypgbouncer:
psql (16.3, server 1.22.1/bouncer)
WARNING: psql major version 16, server major version 1.22.
         Some psql features might not work.
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

pgbouncer=# SHOW HELP;
NOTICE:  Console usage
DETAIL:
 SHOW HELP|CONFIG|DATABASES|POOLS|CLIENTS|SERVERS|USERS|VERSION
 SHOW PEERS|PEER_POOLS
 SHOW FDS|SOCKETS|ACTIVE_SOCKETS|LISTS|MEM|STATE
 SHOW DNS_HOSTS|DNS_ZONES
 SHOW STATS|STATS_TOTALS|STATS_AVERAGES|TOTALS
 SET key = arg
 RELOAD
 PAUSE [<db>]
 RESUME [<db>]
 DISABLE <db>
 ENABLE <db>
 RECONNECT [<db>]
 KILL <db>
 SUSPEND
 SHUTDOWN
 WAIT_CLOSE [<db>]
 SHOW
```

See the [PgBouncer Admin Console documentation](#) for additional details about available statistics and process control commands.

Next Steps

Now that we can enable connection pooling in a cluster, let's explore some ways that we can manage users and databases in our Postgres cluster using PGO.

User / Database Management

Crunchy Postgres for Kubernetes comes with some out-of-the-box conveniences for managing users and databases in your Postgres cluster. However, you may have requirements where you need to create additional users, adjust user privileges or add additional databases to your cluster.

For detailed information for how user and database management works in Crunchy Postgres for Kubernetes, please see the User Management section of the architecture guide.

Creating a New User

You can create a new user with the following snippet in the `postgrescluster` custom resource. Let's add this to our `hippo` database:

```
spec:
  users:
    - name: rhino
```

You can now apply the changes and see that the new user is created. Note the following:

- The user would only be able to connect to the default `postgres` database.
- The user will not have any connection credentials populated into the `hippo-pguser-rhino` Secret.
- The user is unprivileged.

Creating a New Database

Let's create a new database named `zoo` that we will let the `rhino` user access:

```
spec:
  users:
    - name: rhino
      databases:
        - zoo
```

Inspect the `hippo-pguser-rhino` Secret. You should now see that the `dbname` and `uri` fields are now populated!

We can set role privileges by using the standard [role attributes](#) that Postgres provides and adding them to the `spec.users.options`. Let's say we want the rhino to become a superuser (be careful about doling out Postgres superuser privileges!). You can add the following to the spec:

```
spec:
  users:
    - name: rhino
      databases:
        - zoo
      options: 'SUPERUSER'
```

There you have it: we have created a Postgres user named `rhino` with superuser privileges that has access to the `zoo` database (though a superuser has access to all databases!).

Adjusting Privileges

Let's say you want to revoke the superuser privilege from `rhino`. You can do so with the following:

```
spec:
  users:
    - name: rhino
      databases:
        - zoo
      options: 'NOSUPERUSER'
```

If you want to add multiple privileges, you can add each privilege with a space between them in `options`, e.g.:

```
spec:
  users:
    - name: rhino
      databases:
        - zoo
      options: 'CREATEDB CREATEROLE'
```

Managing the postgres User

By default, Crunchy Postgres for Kubernetes does not give you access to the `postgres` user. However, you can get access to this account by doing the following:

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

This will create a Secret of the pattern `<clusterName>-pguser-postgres` that contains the credentials of the `postgres` account. For our `hippo` cluster, this would be `hippo-pguser-postgres`.

Skipping user and database creation

In this tutorial, we've described two different PGO behaviors:

- if you leave out the `spec.users` section, the default user and database get created;
- if you fill in the `spec.users` section, those custom users and databases get created, but not the default user and database.

But what if you want to avoid creating the default user and database AND avoid creating custom users and databases? That can be accomplished by setting `spec.users` to an empty list:

```
spec:
  users: []
```

For example, if we created a PostgresCluster with the above empty list for `spec.users`, that cluster would have only the roles required by Crunchy Postgres for Kubernetes, and only the databases that a new PostgreSQL cluster would have.

Deleting a User

Crunchy Postgres for Kubernetes does not delete users automatically: after you remove the user from the spec, it will still 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 Postgres cluster.

For example, with the above `rhino` user, you would run the following:

```
DROP OWNED BY rhino;
DROP ROLE rhino;
```

Note that you may need to run `DROP OWNED BY rhino CASCADE;` upon your object ownership structure -- be very careful with this command!

Deleting a Database

Crunchy Postgres for Kubernetes does not delete databases automatically: after you remove all instances of the database from the spec, it will still exist in your cluster. To completely remove the database, you must run the `DROP DATABASE` command as a Postgres superuser.

For example, to remove the `zoo` database, you would execute the following:

```
DROP DATABASE zoo;
```

Creating a Declarative Password for a New User

You can declaratively create a password for a new user by creating a `Secret`. This allows you to easily predefine passwords for your various Postgres users per your specific password requirements/needs. This also means you can also keep passwords for your various users consistent when creating and recreating `PostgresClusters`.

Let's create a secret by using the following manifest:

```
apiVersion: v1
kind: Secret
metadata:
  name: hippo-pguser-rhino
  labels:
    postgres-operator.crunchydata.com/cluster: hippo
    postgres-operator.crunchydata.com/pguser: rhino
stringData:
  password: river
```

Note that this `Secret` has a name that matches the pattern of our other user secrets: `<clusterName>-pguser-<user-Name>`. Also note that this `Secret` has two labels:

```
postgres-operator.crunchydata.com/cluster: hippo
postgres-operator.crunchydata.com/pguser: rhino
```

These labels associate this `Secret` with the `hippo` cluster and the `rhino` user.

To apply the secret manifest to your Kubernetes cluster, use the `kubectl apply` command:

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

Now let's add the `rhino` user to your `hippo PostgresCluster` custom resource exactly as shown in the [Creating a New User](#) section above:

```
spec:
  users:
    - name: rhino
      databases: [grasslands, forest]
```

You can now apply the changes and see that the new user is created. This user is created with the same permissions and privileges as if you had created them without declaring a **Secret** first. For instance, in the above example, the **rhino** user has permissions to both the **grasslands** and **forest** databases. And just as if this **Secret** were created by the Operator, the **Secret** you've made will be connected to the **PostgresCluster**. This means that if you delete the **PostgresCluster**, the **Secret** will also be deleted.

Note: If multiple **Secrets** have the same labels, the **Secret** with a name in the `<clusterName>-pguser-<userName>` pattern will be used. Otherwise, the secrets will be ordered based on their creation timestamp and CPK will use the secret with the oldest timestamp.

Automatically Creating Per-User Schemas

You can set Crunchy Postgres for Kubernetes to automatically create schemas for users defined in the **spec.users** field of the **PostgresCluster** custom resource. If enabled for a cluster, Crunchy Postgres for Kubernetes will create a schema

- for every user defined in **spec.users**
- named after the user
- for every database that user is given access to in the **spec.users[index].databases** field.

Note: Crunchy Postgres for Kubernetes does not delete Postgres objects or revoke permissions. If you remove the annotation that led to schema creation for a user or remove a user from **spec.users**, Crunchy Postgres for Kubernetes will not remove that user's schema. By removing the annotation, you are telling Crunchy Postgres for Kubernetes not to create any new schemas, but the schemas that were created before will still exist.

Why is this feature here?

Postgres long recommended that permissions for the **public** schema be revoked to prevent one user from tricking another into using a different Postgres object. (See this [CVE](#) for more info.)

As of Postgres 15, this recommendation became the standard behavior. This change results in more secure database behavior, but it also introduced difficulties for people used to the behavior of **public** in Postgres 14 and below. That is, you could no longer start up a Postgres database and connect as a user to a database and start writing tables -- unless you set up your schemas somehow.

While the **spec.databaseInitSQL** field could be used to run SQL to create schemas for users, this solution didn't fit all use-cases, particularly those users who might be running a central Postgres database with several different applications attached.

This feature to automatically create schemas helps users start up a database that they can use to point their applications at, as well as presenting a way to keep the database schemas up to date with changing requirements.

Why is the schema named after the user?

Postgres **search_path** defaults to `"$user", public`, so by creating a schema with the same name as the user, we do not have to alter the **search_path**. By keeping changes minimal, we ensure a Postgres experience that is closer to the baseline.

How can I enable this feature for my cluster?

You can enable Crunchy Postgres for Kubernetes' automatic schema creation feature for any cluster by setting the `postgres-operator.crunchydata.com/autoCreateUserSchema` annotation:

```
kubectl annotate -n postgres-operator postgrescluster hippo \
postgres-operator.crunchydata.com/autoCreateUserSchema=true
```

Once enabled for this cluster, Crunchy Postgres for Kubernetes will handle the schema creation for any user defined in `spec.users` as long as

- the user has some databases defined for them in the spec;
- the user is not named after a reserved schema name.

The reserved schema names are the names of schemas required for proper functioning: `pgbouncer` and `monitor`. Further, Postgres will reject any attempt to make a user named `public`.

For instance, if you were to create a PostgresCluster with the following `users`:

```
spec:
  users:
    - name: rhino
      databases: [grasslands, forest]
    - name: giraffe
      databases: [grasslands, river]
    - name: pgbouncer
      databases: [grasslands]
    - name: crocodile
```

This feature would then create the following:

- `rhino` schemas in database `grasslands` and database `forest` owned by `rhino` user;
- `giraffe` schemas in database `grasslands` and database `river` owned by `giraffe` user;
- no schema created for `pgbouncer` user since that is one of the reserved names;
- no schema created for `crocodile` user since that user has no databases defined for it.

If a schema named `rhino` already existed in the database `grasslands` but was owned by a different role, the Crunchy Postgres for Kubernetes operator would not recreate or change the existing schema.

How can I disable this feature for my cluster?

If you no longer want Crunchy Postgres for Kubernetes to automatically create schemas for users, you can remove the annotation or set it to `false`:

```
kubectl annotate -n postgres-operator postgrescluster hippo \
postgres-operator.crunchydata.com/autoCreateUserSchema-

kubectl annotate -n postgres-operator postgrescluster hippo \
postgres-operator.crunchydata.com/autoCreateUserSchema=false --overwrite
```

By removing the annotation or setting it to `false`, you will prevent the automatic creation of schemas for different users/databases. As noted above, turning this feature off does not remove any schemas that have already been created.

Authenticating Against a TLS-enabled LDAP Server (ldaps)

If you are using LDAP for authentication and connecting to an LDAP server using TLS, you *must* provide the certificate to the certification authority (CA) in the PostgreSQL cluster's spec file.

Consider the following configuration:

```
spec:
  config:
    files:
      - secret:
          name: ldapsecret
          items:
            - key: ca.crt
              path: ldap/ca.crt
    authentication:
      rules:
        - connection: hostssl
          method: ldap
          options:
            ldapserver: myhostname
            ldapport: 636
            ldapbasedn: dc=example,dc=org
            ldapscheme: ldaps
```

In the first section, `spec.config.files` will mount the `ca.crt` file from the Secret named `ldapsecret` to `/etc/postgres/ldap/ca.crt`. This is the path expected by the `LDAPTLS_CACERT` environment variable and allows Postgres to utilize the provided CA when connecting to an LDAP server that requires TLS.

The second section, `spec.authentication.rules`, allows you to configure the appropriate settings for your LDAP server. For more information on the proper settings for your LDAP configuration, please see the [pg_hba.conf](#) and [auth-ldap](#) documentation.

Warning

Providing CA certificates is required for all connections to TLS-enabled LDAP servers for authentication.

Delete a Postgres Cluster

There comes a time when it is necessary to delete your cluster. If you have been [following along with the example](#), you can delete your Postgres cluster by simply running:

```
kubectl delete -k customize/postgres
```

PGO will remove all of the objects associated with your cluster.

With data retention, this is subject to the [retention policy of your PVC](#). For more information on how Kubernetes manages data retention, please refer to the [Kubernetes docs on volume reclaiming](#).

Backup and Disaster Recovery

Database backups create exciting opportunities. When you need to provision development and staging environments, your backups help you to mimic production.

When you need to share data across teams, backing up to shared buckets makes access easy. And most importantly, when a worst case scenario arises, having the ability to restore from your backups will keep you safe from catastrophe.

In Backup Configuration we'll show you how to backup your data to multiple locations for safe keeping. In Backup Management we'll show you how to create backup schedules, retention policies and how to take one-off backups whenever you want. In Disaster Recovery and Cloning we'll show you how to design against disaster with standby clusters and how to practice disaster recovery, so that you'll have the hands-on experience to handle a worst case scenario.

Backup Configuration

An important part of a healthy Postgres cluster is maintaining backups. PGO optimizes its use of open source [pgBackRest](#) to be able to support terabyte size databases. What's more, PGO makes it convenient to perform many common and advanced actions that can occur during the lifecycle of a database, including:

- Setting automatic backup schedules and retention policies
- Backing data up to multiple locations• Support for backup storage in Kubernetes, AWS S3 (or S3-compatible systems like MinIO), Google Cloud Storage (GCS), and Azure Blob Storage
- Taking one-off / ad hoc backups
- Performing a "point-in-time-recovery"
- Cloning data to a new instance

and more.

Let's explore the various disaster recovery features in PGO by first looking at how to set up backups.

Understanding Backup Configuration and Basic Operations

The backup configuration for a PGO managed Postgres cluster resides in the `spec.backups.pgbackrest` section of a custom resource. In addition to indicating which version of pgBackRest to use, this section allows you to configure the fundamental backup settings for your Postgres cluster, including:

- `spec.backups.pgbackrest.image` - image to use for pgBackRest containers. Keep in mind the pgBackRest version used needs to be compatible with operator and Postgres images according to the compatibility matrix.
- `spec.backups.pgbackrest.configuration` - additional configuration and references to Secrets that are needed for configuration of your backups. For example, this may reference a Secret that contains your S3 credentials.
- `spec.backups.pgbackrest.global` - global [pgBackRest configuration](#). An example of this may be setting the global pgBackRest logging level (e.g. `log-level-console: info`) or providing configuration to optimize performance.
- `spec.backups.pgbackrest.repos` - information on each specific pgBackRest backup repository. This allows you to configure where and how your backups and WAL archive are stored. You can keep backups in up to four (4) different locations!

You can configure the **repos** section based on the backup storage system you are looking to use. There are four storage types supported in **spec.backups.pgbackrest.repos**:

Storage Type	Description
azure	For use with Azure Blob Storage.
gcs	For use with Google Cloud Storage (GCS).
s3	For use with Amazon S3 or any S3 compatible storage system such as MinIO.
volume	For use with a Kubernetes Persistent Volume .

spec.backups.pgbackrest.repos.name - **requires** a name, and that name must follow pgBackRest's convention of assigning configuration to a specific repository using a **repoN** format, e.g. **repo1**, **repo2**, etc. You can customize your configuration based upon the name that you assign in the spec. Please see Set up Multiple Backup Repositories.

By default, backups are stored in a directory that follows the pattern **pgbackrest/repoN** where **N** is the number of the repo. This typically does not present issues when storing your backup information in a Kubernetes volume, but it can present complications if you are storing all of your backups in the same backup in a blob storage system like S3/GCS/Azure. You can avoid conflicts by setting the **repoN-path** variable in **spec.backups.pgbackrest.global**. The convention we recommend for setting this variable is **/pgbackrest/\$NAMESPACE/\$CLUSTER_NAME/repoN**. For example, if I have a cluster named **hippo** in the namespace **postgres-operator**, I would set the following:

```
spec:
  backups:
    pgbackrest:
      global:
        repo1-path: /pgbackrest/postgres-operator/hippo/repo1
```

As mentioned earlier, you can store backups in up to four different repositories. You can also mix and match, e.g. you could store your backups in two different S3 repositories. Each storage type does have its own required attributes that you need to set. We will cover that later in this section.

Now that we've covered the basics, let's learn how to set up our backup repositories.

Setting Up a Backup Repository

As mentioned above, PGO, the Postgres Operator from Crunchy Data, supports multiple ways to store backups. Regardless of which way you choose to store your backups, PGO will create a repo host Pod that functions as a command execution server for your pgBackRest backups. This Pod will be the primary location for running pgBackRest commands and will be configured to work with all Postgres Instances. It will also be the main storage location of your pgBackRest logs, assuming at least one Kubernetes storage volume repo is defined.

With all that in mind, let's look into each method and see how you can ensure your backups and archives are being safely stored.

Using Kubernetes Volumes

The simplest way to get started storing backups is to use a Kubernetes Volume. This was already configured as part of the create a Postgres cluster example. Let's take a closer look at some of that configuration:

```
-name: repo1
  volume:
    volumeClaimSpec:
      accessModes:
        - "ReadWriteOnce"
      resources:
        requests:
          storage: 1Gi
```

The one requirement of volume is that you need to fill out the `volumeClaimSpec` attribute. This attribute uses the same format as a [persistent volume claim](#) spec. In fact, we performed a similar set up when we created a Postgres cluster.

In the above example, we assume that the Kubernetes cluster is using a default storage class. If your cluster does not have a default storage class, or you wish to use a different storage class, you will have to set `spec.backups.pgbackrest.repos.volume.volumeClaimSpec.storageClassName`.

Using S3

Setting up backups in S3 requires a few additional modifications to your custom resource spec and either

- the use of a Secret to protect your S3 credentials, or
- setting up identity providers in AWS to allow pgBackRest to assume a role with permissions.

Using S3 Credentials

There is an example for creating a Postgres cluster that uses S3 for backups in the `kustomize/s3` directory in the [Postgres Operator examples](#) repository. In this directory, there is a file called `s3.conf.example`. Copy this example file to `s3.conf`:

```
cp s3.conf.example s3.conf
```

Note that `s3.conf` is protected from commit by a `.gitignore`.

Open up `s3.conf`, you will see something similar to:

```
repo1-s3-key=$YOUR_AWS_S3_KEY
repo1-s3-key-secret=$YOUR_AWS_S3_KEY_SECRET
```

Replace the values with your AWS S3 credentials and save.

Now, open up `kustomize/s3/postgres.yaml`. In the `s3` section, you will see something similar to:

```
s3:
  bucket: "$YOUR_AWS_S3_BUCKET_NAME"
  endpoint: "$YOUR_AWS_S3_ENDPOINT"
  region: "$YOUR_AWS_S3_REGION"
```

Again, replace these values with the values that match your S3 configuration. For `endpoint`, only use the domain and, if necessary, the port (e.g. `s3.us-east-2.amazonaws.com`).

Note that `region` is required by S3, as does pgBackRest. If you are using a storage system with a S3 compatibility layer that does not require `region`, you can fill in region with a random value.

If you are using MinIO, you may need to set the URI style to use **path** mode. You can do this from the global settings, e.g. for **repo1**:

```
spec:
  backups:
    pgbackrest:
      global:
        repo1-s3-uri-style: path
```

When your configuration is saved, you can deploy your cluster:

```
kubectl apply -k kustomize/s3
```

Watch your cluster: you will see that your backups and archives are now being stored in S3!

Using an AWS-integrated identity provider and role

If you deploy PostgresClusters to AWS Elastic Kubernetes Service, you can take advantage of their IAM role integration. When you attach a certain annotation to your PostgresCluster spec, AWS will automatically mount an AWS token and other needed environment variables. These environment variables will then be used by pgBackRest to assume the identity of a role that has permissions to upload to an S3 repository.

This method requires [additional setup in AWS IAM](#). Use the procedure in the linked documentation for the first two steps described below:

- Create an OIDC provider for your EKS cluster.
- Create an IAM policy for bucket access and an IAM role with a trust relationship with the OIDC provider in step 1.

The third step is to associate that IAM role with a ServiceAccount, but there's no need to do that manually, as PGO does that for you. First, make a note of the IAM role's **ARN**.

You can then make the following changes to the files in the **kustomize/s3** directory in the [Postgres Operator examples](#) repository:

1. Add the **s3** section to the spec in **kustomize/s3/postgres.yaml** as discussed in the [Using S3 Credentials](#) section above. In addition to that, add the required **eks.amazonaws.com/role-arn** annotation to the PostgresCluster spec using the IAM **ARN** that you noted above.

For instance, given an IAM role with the ARN **arn:aws:iam::123456768901:role/allow_bucket_access**, you would add the following to the PostgresCluster spec:

```
spec:
  metadata:
    annotations:
      eks.amazonaws.com/role-arn: "arn:aws:iam::123456768901:role/allow_bucket_access"
```

That **annotations** field will get propagated to the ServiceAccounts that require it automatically.

2. Copy the **s3.conf.example** file to **s3.conf**:

```
cp s3.conf.example s3.conf
```

Update that `kustomize/s3/s3.conf` file so that it looks like this:

```
repo1-s3-key-type=web-id
```

That `repo1-s3-key-type=web-id` line will tell [pgBackRest](#) to use the IAM integration.

With those changes saved, you can deploy your cluster:

```
kubectl apply -k kustomize/s3
```

And watch as it spins up and backs up to S3 using pgBackRest's IAM integration.

Using Google Cloud Storage (GCS)

Similar to S3, setting up backups in Google Cloud Storage (GCS) requires a few additional modifications to your custom resource spec and the use of a Secret to protect your GCS credentials.

There is an example for creating a Postgres cluster that uses GCS for backups in the `kustomize/gcs` directory in the [Postgres Operator examples](#) repository. In order to configure this example to use GCS for backups, you will need do two things.

First, copy your GCS key secret (which is a JSON file) into `kustomize/gcs/gcs-key.json`. Note that a `.gitignore` directive prevents you from committing this file.

Next, open the `postgres.yaml` file and edit `spec.backups.pgbackrest.repos.gcs.bucket` to the name of the GCS bucket that you want to back up to.

Save this file, and then run:

```
kubectl apply -k kustomize/gcs
```

Watch your cluster: you will see that your backups and archives are now being stored in GCS!

Using Azure Blob Storage

Similar to the above, setting up backups in Azure Blob Storage requires a few additional modifications to your custom resource spec and the use of a Secret to protect your Azure Storage credentials.

There is an example for creating a Postgres cluster that uses Azure for backups in the `kustomize/azure` directory in the [Postgres Operator examples](#) repository. In this directory, there is a file called `azure.conf.example`. Copy this example file to `azure.conf`:

```
cp azure.conf.example azure.conf
```

Note that `azure.conf` is protected from commit by a `.gitignore`.

Open up `azure.conf`, you will see something similar to:

```
repo1-azure-account=$YOUR_AZURE_ACCOUNT
repo1-azure-key=$YOUR_AZURE_KEY
```

Replace the values with your Azure credentials and save.

Now, open up `kustomize/azure/postgres.yaml`. In the `azure` section, you will see something similar to:

```
azure:
  container: "$YOUR_AZURE_CONTAINER"
```

Again, replace these values with the values that match your Azure configuration.

When your configuration is saved, you can deploy your cluster:

```
kubectl apply -k kustomize/azure
```

Watch your cluster: you will see that your backups and archives are now being stored in Azure!

Set Up Multiple Backup Repositories

It is possible to store backups in multiple locations. For example, you may want to keep your backups both within your Kubernetes cluster and S3. There are many reasons for doing this:

- It is typically faster to heal Postgres instances when your backups are closer
- You can set different backup retention policies based upon your available storage
- You want to ensure that your backups are distributed geographically

and more.

PGO lets you store your backups in up to four locations simultaneously. You can mix and match: for example, you can store backups both locally and in GCS, or store your backups in two different GCS repositories. Note that regardless of how many repo Volumes are defined, only one repo host Pod will be created.

The [multi-backup-repo](#) example in the Postgres Operator examples repository sets up backups in four different locations using each storage type. You can modify this example to match your desired backup topology.

Additional Notes

While storing Postgres archives (write-ahead log [WAL] files) occurs in parallel when saving data to multiple pgBackRest repos, you cannot take parallel backups to different repos at the same time. PGO will ensure that all backups are taken serially. Future work in pgBackRest will address parallel backups to different repos. Please don't confuse this with parallel backup: pgBackRest does allow for backups to use parallel processes when storing them to a single repo!

Encryption

You can encrypt your backups using AES-256 encryption using the CBC mode. This can be used independent of any encryption that may be supported by an external backup system.

To encrypt your backups, you need to set the cipher type and provide a passphrase. The passphrase should be long and random (e.g. the pgBackRest documentation recommends `openssl rand -base64 48`). The passphrase should be kept in a Secret.

Let's use our **hippo** cluster as an example. Let's create a new directory. First, create a file called **pgbackrest-secrets.conf** in this directory. It should look something like this:

```
repo1-cipher-pass=your-super-secure-encryption-key-passphrase
```

This contains the passphrase used to encrypt your data.

Next, create a **kustomization.yaml** file that looks like this:

```
namespace: postgres-operator

secretGenerator: - name: hippo-pgbackrest-secrets
  files:
    - pgbackrest-secrets.conf

generatorOptions:  disableNameSuffixHash: true

resources: - postgres.yaml
```

Finally, create the manifest for the Postgres cluster in a file named **postgres.yaml** that is similar to the following:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  postgresVersion: 17
  instances:
    - dataVolumeClaimSpec:
        accessModes:
          - 'ReadWriteOnce'
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      configuration:
        - secret:
            name: hippo-pgbackrest-secrets
      global:
        repo1-cipher-type: aes-256-cbc
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - 'ReadWriteOnce'
              resources:
                requests:
                  storage: 1Gi
```

Notice the reference to the Secret that contains the encryption key:

```
spec:
  backups:
    pgbackrest:
      configuration:
```

```
- secret:  
  name: hippo-pgbackrest-secrets
```

as well as the configuration for enabling AES-256 encryption using the CBC mode:

```
spec:  
  backups:  
    pgbackrest:  
      global:  
        repo1-cipher-type: aes-256-cbc
```

You can now create a Postgres cluster that has encrypted backups!

Limitations

Currently the encryption settings cannot be changed on backups after they are established.

Custom Backup Configuration

Most of your backup configuration can be configured through the `spec.backups.pgbackrest.global` attribute, or through information that you supply in the ConfigMap or Secret that you refer to in `spec.backups.pgbackrest.configuration`. You can also provide additional Secret values if need be, e.g. `repo1-cipher-pass` for encrypting backups.

The full list of pgBackRest configuration options is available at <https://pgbackrest.org/configuration.html>.

Warning

Some pgBackRest options require write access to paths with adequate storage capacity within your container. For example, if you enable [archive-async](#), make sure you also add a proper [spool-path](#).

Reducing Primary Instance Load with the Backup from Standby Option

Info

FEATURE AVAILABILITY: *Available in v5.7.0 and above*

You can now configure the pgBackRest [Backup from Standby Option](#) in order to reduce the load on the primary Postgres Instance Pod. The necessary settings can be configured as follows:

```
spec:  
  instances:  
    - name: instance1  
      replicas: 2  
  ...  
  backups:  
    pgbackrest:  
      global:  
        backup-standby: "y"
```

Warning

As shown above, the **backup-standby** option will require at least one Postgres Instance replica. If at least one replica is not accessible when taking a backup, it will fail with the following error, "ERROR: [056]: unable to find standby cluster - cannot proceed."

As described in the pgBackRest [documentation](#), configuring the **backup-standby** option causes the vast majority of the backup files to be pulled from a replica Postgres Instance (that is, a "standby database") rather than all of them coming from the primary Postgres Instance (the "primary database"). Additionally, this pgBackRest backup Job will always execute on the repo host Pod. Taken together, this will greatly reduce the load on the primary Postgres Instance when performing a backup.

IPv6 Support

If you are running your cluster in an IPv6-only environment, you will need to add an annotation to your PostgresCluster so that PGO knows to set pgBackRest's **tls-server-address** to an IPv6 address. Otherwise, **tls-server-address** will be set to **0.0.0.0**, making pgBackRest inaccessible, and backups will not run. The annotation should be added as shown below:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  annotations:
    postgres-operator.crunchydata.com/pgbackrest-ip-version: IPv6
```

Next Steps

We've now seen how to use PGO to get our backups and archives set up and safely stored. Now let's take a look at backup management and how we can do things such as set backup frequency, set retention policies, and even take one-off backups!

Backup Management

In the previous section, we looked at a brief overview of the full disaster recovery feature set that PGO provides and explored how to configure backups for our Postgres cluster.

Now that we have backups set up, let's look at some of the various backup management tasks we can perform. These include:

- Setting up scheduled backups
- Setting backup retention policies
- Taking one-off / ad hoc backups

Managing Scheduled Backups

PGO sets up your Postgres clusters so that they are continuously archiving the [write-ahead log](#): your data is constantly being stored in your backup repository. Effectively, this is a backup!

However, in a disaster recovery scenario, you likely want to get your Postgres cluster back up and running as quickly as possible (e.g. a short "[recovery time objective \(RTO\)](#)"). What helps accomplish this is to take periodic backups. This makes it faster to restore!

[pgBackRest](#), the backup management tool used by PGO, provides different backup types to help both from a space management and RTO optimization perspective. These backup types include:

- **full**: A backup of your entire Postgres cluster. This is the largest of all of the backup types.
- **differential**: A backup of all of the data since the last **full** backup.
- **incremental**: A backup of all of the data since the last **full**, **differential**, or **incremental** backup.

Selecting the appropriate backup strategy for your Postgres cluster is outside the scope of this tutorial, but let's look at how we can set up scheduled backups.

Backup schedules are stored in the `spec.backups.pgbackrest.repos.schedules` section. Each value in this section accepts a [cron-formatted](#) string that dictates the backup schedule.

Let's say that our backup policy is to take a full backup weekly on Sunday at 1am and take differential backups daily at 1am on every day except Sunday. We would want to add configuration to our spec that looks similar to:

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

To manage scheduled backups, PGO will create several Kubernetes [CronJobs](#) that will perform backups on the specified periods. The backups will use the configuration that you specified.

Ensuring you take regularly scheduled backups is important to maintaining Postgres cluster health. However, you don't need to keep all of your backups: this could cause you to run out of space! As such, it's also important to set a backup retention policy.

Managing Backup Retention

PGO lets you set backup retention on full and differential backups. When a full backup expires, either through your retention policy or through manual expiration, pgBackRest will clean up any backup and WAL files associated with it. For example, if you have a full backup with four associated incremental backups, when the full backup expires, all of its incremental backups also expire.

There are two different types of backup retention you can set:

- **count**: This is based on the number of backups you want to keep. This is the default.

- **time**: This is based on the total number of days you would like to keep a backup.

Let's look at an example where we keep full backups for 14 days. The most convenient way to do this is through the **spec.backups.pgbackrest.global** section:

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

The full list of available configuration options is in the [pgBackRest configuration](#) guide.

Taking a One-Off Backup

There are times where you may want to take a one-off backup, such as before major application changes or updates. This is not your typical declarative action -- in fact a one-off backup is imperative in its nature! -- but it is possible to take a one-off backup of your Postgres cluster with PGO.

First, you need to configure the **spec.backups.pgbackrest.manual** section to be able to take a one-off backup. This contains information about the type of backup you want to take and any other [pgBackRest configuration](#) options.

Let's configure the custom resource to take a one-off full backup:

```
spec:
  backups:
    pgbackrest:
      manual:
        repoName: repo1
        options:
          - --type=full
```

This does not trigger the one-off backup -- you have to do that by adding the **postgres-operator.crunchydata.com/pgbackrest-backup** annotation to your custom resource. The best way to set this annotation is with a timestamp, so you know when you initialized the backup.

For example, for our **hippo** cluster, we can run the following command to trigger the one-off backup:

```
kubectl annotate -n postgres-operator postgrescluster hippo postgres-operator.crunchydata.com/pgbackrest-backup="$(date)"
```

PGO will detect this annotation and create a new, one-off backup Job!

If you intend to take one-off backups with similar settings in the future, you can leave those in the spec; just update the annotation to a different value the next time you are taking a backup.

To re-run the command above, you will need to add the **--overwrite** flag so the annotation's value can be updated, i.e.

```
kubectl annotate -n postgres-operator postgrescluster hippo --overwrite postgres-operator.crunchydata.com/pgbackrest-backup="$(date)"
```

Next Steps

We've covered the fundamental tasks with managing backups. What about restores? Or cloning data into new Postgres clusters? Let's explore!

Disaster Recovery and Cloning

Warning

Cloning requires a `backups` section to be defined in both source and clone cluster specs.
See [Backup Configuration](#) for details.

Perhaps someone accidentally dropped the `users` table. Perhaps you want to clone your production database to a step-down environment. Perhaps you want to exercise your disaster recovery system (and it is important that you do!).

Regardless of scenario, it's important to know how you can perform a "restore" operation with PGO to be able to recovery your data from a particular point in time, or clone a database for other purposes.

Let's look at how we can perform different types of restore operations. First, let's understand the core restore properties on the custom resource.

Restore Properties

Info

As of v5.0.5, PGO offers the ability to restore from an existing PostgresCluster or a remote cloud-based data source, such as S3, GCS, etc. For more on that, see the [Clone From Backups Stored in S3 / GCS / Azure Blob Storage](#) section.

Note that you **cannot** use both a local PostgresCluster data source and a remote cloud-based data source at one time; if both the `dataSource.postgresCluster` and `dataSource.pgbackrest` fields are filled in, the local PostgresCluster data source will take precedence.

There are several attributes on the custom resource that are important to understand as part of the restore process. All of these attributes are grouped together in the `spec.dataSource.postgresCluster` section of the custom resource.

Please review the table below to understand how each of these attributes work in the context of setting up a restore operation.

- `spec.dataSource.postgresCluster.clusterName`: The name of the cluster that you are restoring from. This corresponds to the `metadata.name` attribute on a different `postgrescluster` custom resource.
- `spec.dataSource.postgresCluster.clusterNamespace`: The namespace of the cluster that you are restoring from. Used when the cluster exists in a different namespace.

- **spec.dataSource.postgresCluster.repoName**: The name of the pgBackRest repository from the **spec.dataSource.postgresCluster.clusterName** to use for the restore. Can be one of **repo1**, **repo2**, **repo3**, or **repo4**. The repository must exist in the other cluster.
- **spec.dataSource.postgresCluster.options**: Any additional [pgBackRest restore options](#) or general options that PGO allows. For example, you may want to set **--process-max** to help improve performance on larger databases; but you will not be able to set **--target-action**, since that option is currently disallowed. (PGO always sets it to **promote** if a **--target** is present, and otherwise leaves it blank.)
- **spec.dataSource.postgresCluster.resources**: Setting [resource limits and requests](#) of the restore job can ensure that it runs efficiently.
- **spec.dataSource.postgresCluster.affinity**: Custom [Kubernetes affinity](#) rules constrain the restore job so that it only runs on certain nodes.
- **spec.dataSource.postgresCluster.tolerations**: Custom [Kubernetes tolerations](#) allow the restore job to run on [tainted](#) nodes.

Let's walk through some examples for how we can clone and restore our databases.

Clone a Postgres Cluster

Let's create a clone of our **hippo** cluster that we created previously. We know that our cluster is named **hippo** (based on its **metadata.name**) and that we only have a single backup repository called **repo1**.

Let's call our new cluster **elephant**. We can create a clone of the **hippo** cluster using a manifest like this:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: elephant
spec:
  dataSource:
    postgresCluster:
      clusterName: hippo
      repoName: repo1
  postgresVersion: 17
  instances:
    - dataVolumeClaimSpec:
        accessModes:
          - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - "ReadWriteOnce"
              resources:
                requests:
                  storage: 1Gi
```

Note this section of the spec:

```
spec:
  dataSource:
    postgresCluster:
      clusterName: hippo
      repoName: repo1
```

This is the part that tells PGO to create the **elephant** cluster as an independent copy of the **hippo** cluster.

The above is all you need to do to clone a Postgres cluster! PGO will work on creating a copy of your data on a new persistent volume claim (PVC) and work on initializing your cluster to spec. Easy!

Perform a Point-in-time-Recovery (PITR)

Did someone drop the user table? You may want to perform a point-in-time-recovery (PITR) to revert your database back to a state before a change occurred. Fortunately, PGO can help you do that.

You can set up a PITR using the [restore](#) command of [pgBackRest](#), the backup management tool that powers the disaster recovery capabilities of PGO. You will need to set a few options on `spec.dataSource.postgresCluster.options` to perform a PITR. These options include:

- `--type=time`: This tells pgBackRest to perform a PITR.
- `--target`: Where to perform the PITR to. An example recovery target is `2021-06-09 14:15:11-04`. The timezone specified here as -04 for EDT. Please see the [pgBackRest documentation for other timezone options](#).
- `--set` (optional): Choose which backup to start the PITR from.

A few quick notes before we begin:

- To perform a PITR, you must have a backup that finished before your PITR time. In other words, you can't perform a PITR back to a time where you do not have a backup!
- All relevant WAL files must be successfully pushed for the restore to complete correctly.
- Be sure to select the correct repository name containing the desired backup!

With that in mind, let's use the **elephant** example above. Let's say we want to perform a point-in-time-recovery (PITR) to `2021-06-09 14:15:11-04`. We can use the following manifest:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: elephant
spec:
  dataSource:
    postgresCluster:
      clusterName: hippo
      repoName: repo1
      options:
        - --type=time
        - --target="2021-06-09 14:15:11-04"
  postgresVersion: 17
  instances:
    - dataVolumeClaimSpec:
        accessModes:
          - "ReadWriteOnce"
```

```

resources:
  requests:
    storage: 1Gi
backups:
  pgbackrest:
    repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
              - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi

```

The section to pay attention to is this:

```

spec:
  dataSource:
    postgresCluster:
      clusterName: hippo
      repoName: repo1
      options:
        - --type=time
        - --target="2021-06-09 14:15:11-04"

```

Notice how we put in the options to specify where to make the PITR.

Using the above manifest, PGO will go ahead and create a new Postgres cluster that recovers its data up until **2021-06-09 14:15:11-04**. At that point, the cluster is promoted and you can start accessing your database from that specific point in time!

Perform an In-Place Point-in-time-Recovery (PITR)

Similar to the PITR restore described above, you may want to perform a similar reversion back to a state before a change occurred, but without creating another PostgreSQL cluster. Fortunately, PGO can help you do this as well.

You can set up a PITR using the [restore](#) command of [pgBackRest](#), the backup management tool that powers the disaster recovery capabilities of PGO. You will need to set a few options on `spec.backups.pgbackrest.restore.options` to perform a PITR. These options include:

- `--type=time`: This tells pgBackRest to perform a PITR.
- `--target`: Where to perform the PITR to. An example recovery target is **2021-06-09 14:15:11-04**
- `--set` (optional): Choose which backup to start the PITR from.

A few quick notes before we begin:

- To perform a PITR, you must have a backup that finished before your PITR time. In other words, you can't perform a PITR back to a time where you do not have a backup!
- All relevant WAL files must be successfully pushed for the restore to complete correctly.
- Be sure to select the correct repository name containing the desired backup!

To perform an in-place restore, users will first fill out the restore section of the spec as follows:

```
spec:
  backups:
    pgbackrest:
      restore:
        enabled: true
        repoName: repo1
        options:
          - --type=time
          - --target="2021-06-09 14:15:11-04"
```

And to trigger the restore, you will then annotate the PostgresCluster as follows:

```
kubectl annotate -n postgres-operator postgrescluster hippo --overwrite postgres-operator.crunchydata.com/pgbackrest-restore="$(date)"
```

And once the restore is complete, in-place restores can be disabled:

```
spec:
  backups:
    pgbackrest:
      restore:
        enabled: false
```

Notice how we put in the options to specify where to make the PITR.

Using the above manifest, PGO will go ahead and re-create your Postgres cluster to recover its data up until

2021-06-09 14:15:11-04. At that point, the cluster is promoted and you can start accessing your database from that specific point in time!

Restore Individual Databases

You might need to restore specific databases from a cluster backup, for performance reasons or to move selected databases to a machine that does not have enough space to restore the entire cluster backup.

Warning

pgBackRest supports this case, but it is important to make sure this is what you want. Restoring in this manner will restore the requested database from backup and make it accessible, but all of the other databases in the backup will NOT be accessible after restore.

For example, if your backup includes databases **test1**, **test2**, and **test3**, and you request that **test2** be restored, the **test1** and **test3** databases will NOT be accessible after restore is completed. Please review the [pgBackRest documentation on the limitations on restoring individual databases](#).

You can restore individual databases from a backup using a spec similar to the following:

```
spec:
  backups:
    pgbackrest:
      restore:
        enabled: true
        repoName: repo1
```

```
options:
  - --db-include=hippo
```

where `--db-include=hippo` would restore only the contents of the `hippo` database.

Standby Cluster

Advanced high-availability and disaster recovery strategies involve spreading your database clusters across data centers to help maximize uptime. PGO provides ways to deploy postgresclusters that can span multiple Kubernetes clusters using an external storage system or PostgreSQL streaming replication. The disaster recovery architecture documentation provides a high-level overview of using standby clusters with PGO.

Creating a Standby Cluster

This tutorial section will describe how to create three different types of standby clusters, one using an external storage system, one that is streaming data directly from the primary, and one that takes advantage of both external storage and streaming. These example clusters can be created in the same Kubernetes cluster, using a single PGO instance, or spread across different Kubernetes clusters and PGO instances with the correct storage and networking configurations.

Repo-based Standby

A repo-based standby will recover from WAL files that a pgBackRest repo stored in external storage. The primary cluster should be created with a cloud-based backup configuration. The following manifest defines a Postgrescluster with `standby.enabled` set to true and `repoName` configured to point to the `s3` repo configured in the primary:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo-standby
spec:
  postgresVersion: 17
  instances:
    - dataVolumeClaimSpec: { accessModes: [ReadWriteOnce], resources: { requests: { storage: 1Gi } } }
  backups:
    pgbackrest:
      repos:
        - name: repo1
          s3:
            bucket: "my-bucket"
            endpoint: "s3.ca-central-1.amazonaws.com"
            region: "ca-central-1"
  standby:
    enabled: true
    repoName: repo1
```

Streaming Standby

A streaming standby relies on an authenticated connection to the primary over the network. The primary cluster should be accessible via the network and allow TLS authentication (TLS is enabled by default). In the following manifest, we have `standby.enabled` set to `true` and have provided both the `host` and `port` that point to the primary cluster. We

have also defined `customTLSSecret` and `customReplicationTLSSecret` to provide certs that allow the standby to authenticate to the primary. For this type of standby, you must use custom TLS:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo-standby
spec:
  postgresVersion: 17
  instances:
    - dataVolumeClaimSpec: { accessModes: [ReadWriteOnce], resources: { requests: { storage: 1Gi } } }
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec: { accessModes: [ReadWriteOnce], resources: { requests: { storage: 1Gi } } }
  customTLSSecret:
    name: cluster-cert
  customReplicationTLSSecret:
    name: replication-cert
  standby:
    enabled: true
    host: "192.0.2.2"
    port: 5432
```

Streaming Standby with an External Repo

Another option is to create a standby cluster using an external pgBackRest repo that streams from the primary. With this setup, the standby cluster will continue recovering from the pgBackRest repo if streaming replication falls behind. In this manifest, we have enabled the settings from both previous examples:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo-standby
spec:
  postgresVersion: 17
  instances:
    - dataVolumeClaimSpec: { accessModes: [ReadWriteOnce], resources: { requests: { storage: 1Gi } } }
  backups:
    pgbackrest:
      repos:
        - name: repo1
          s3:
            bucket: "my-bucket"
            endpoint: "s3.ca-central-1.amazonaws.com"
            region: "ca-central-1"
  customTLSSecret:
    name: cluster-cert
  customReplicationTLSSecret:
    name: replication-cert
  standby:
    enabled: true
    repoName: repo1
    host: "192.0.2.2"
    port: 5432
```

Monitoring a Standby Cluster

When deploying a standby cluster with monitoring enabled, additional configuration is required to allow the `postgres_exporter` to gather metrics from the database. The `ccp_monitoring` password stored in the standby is replicated from the primary database. Because the standby cluster is reconciled separately from the primary, the secret that is created does not have the correct credentials.

To enable monitoring within a standby cluster, you will need to ensure the password defined within the `$CLUSTER_NAME-monitoring` secret matches across both the primary and standby `PostgresClusters`. You can either copy the password from the secret in the primary cluster into the standby secret, or provide a custom password for both clusters. Reference the day-two monitoring tutorial for more information about setting a custom monitoring password.

After the standby cluster's monitoring secret contains the correct credentials for the `ccp_monitoring` user, the `postgres_exporter` processes will be able to connect to Postgres and gather metrics. These metrics will be available through [Grafana](#) and the rest of the monitoring stack.

Promoting a Standby Cluster

At some point, you will want to promote the standby to start accepting both reads and writes. This has the net effect of pushing WAL (transaction archives) to the pgBackRest repository, so we need to ensure we don't accidentally create a split-brain scenario. Split-brain can happen if two primary instances attempt to write to the same repository. If the primary cluster is still active, make sure you shutdown the primary before trying to promote the standby cluster.

Once the primary is inactive, we can promote the standby cluster by removing or disabling its `spec.standby` section:

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

This change triggers the promotion of the standby leader to a primary PostgreSQL instance and the cluster begins accepting writes.

Clone From Backups Stored in S3 / GCS / Azure Blob Storage

You can clone a Postgres cluster from backups that are stored in AWS S3 (or a storage system that uses the S3 protocol), GCS, or Azure Blob Storage without needing an active Postgres cluster! The method to do so is similar to how you clone from an existing `PostgresCluster`. This is useful if you want to have a data set for people to use but keep it compressed on cheaper storage.

For the purposes of this example, let's say that you created a Postgres cluster named `hippo` that has its backups stored in S3 that looks similar to this:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  postgresVersion: 17
  instances:
    - dataVolumeClaimSpec:
        accessModes:
```

```

      - 'ReadWriteOnce'
resources:
  requests:
    storage: 1Gi
backups:
  pgbackrest:
    configuration:
      - secret:
          name: pgo-s3-creds
    global:
      repo1-path: /pgbackrest/postgres-operator/hippo/repo1
    manual:
      repoName: repo1
      options:
        - --type=full
    repos:
      - name: repo1
        s3:
          bucket: 'my-bucket'
          endpoint: 's3.ca-central-1.amazonaws.com'
          region: 'ca-central-1'

```

Ensure that the credentials in **pgo-s3-creds** match your S3 credentials. For more details on deploying a Postgres cluster using S3 for backups, please see the Backups section of the tutorial.

For optimal performance when creating a new cluster from an active cluster, ensure that you take a recent full backup of the previous cluster. The above manifest is set up to take a full backup. Assuming **hippo** is created in the **postgres-operator** namespace, you can trigger a full backup with the following command. If you are using Bash:

```
kubectl annotate -n postgres-operator postgrescluster hippo --overwrite postgres-operator.crunchydata.com/pgbackrest-backup="$( date '+%F_%H:%M:%S' )"
```

For Powershell environments:

```
kubectl annotate -n postgres-operator postgrescluster hippo --overwrite postgres-operator.crunchydata.com/pgbackrest-backup="$( Get-Date -Format "yyyy-MM-dd_HH:mm:ss" )"
```

Wait for the backup to complete. Once this is done, you can delete the Postgres cluster.

Now, let's clone the data from the **hippo** backup into a new cluster called **elephant**. You can use a manifest similar to this:

```

apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: elephant
spec:
  postgresVersion: 17
  dataSource:
    pgbackrest:
      stanza: db
      configuration:
        - secret:
            name: pgo-s3-creds
    global:
      repo1-path: /pgbackrest/postgres-operator/hippo/repo1
  repo:
    name: repo1
    s3:

```

```

      bucket: 'my-bucket'
      endpoint: 's3.ca-central-1.amazonaws.com'
      region: 'ca-central-1'
instances:
  - dataVolumeClaimSpec:
      accessModes:
        - 'ReadWriteOnce'
      resources:
        requests:
          storage: 1Gi
backups:
  pgbackrest:
    configuration:
      - secret:
          name: pgo-s3-creds
    global:
      repo1-path: /pgbackrest/postgres-operator/elephant/repo1
    repos:
      - name: repo1
        s3:
          bucket: 'my-bucket'
          endpoint: 's3.ca-central-1.amazonaws.com'
          region: 'ca-central-1'

```

There are a few things to note in this manifest. First, note that the `spec.dataSource.pgbackrest` object in our new `PostgresCluster` is very similar but slightly different from the old `PostgresCluster`'s `spec.backups.pgbackrest` object. The key differences are:

- No image is necessary when restoring from a cloud-based data source
- `stanza` is a required field when restoring from a cloud-based data source
- `backups.pgbackrest` has a `repos` field, which is an array
- `dataSource.pgbackrest` has a `repo` field, which is a single object

Note also the similarities:

- We are reusing the secret for both (because the new restore pod needs to have the same credentials as the original backup pod)
- The `repo` object is the same
- The `global` object is the same

This is because the new restore pod for the `elephant` `PostgresCluster` will need to reuse the configuration and credentials that were originally used in setting up the `hippo` `PostgresCluster`.

In this example, we are creating a new cluster which is also backing up to the same S3 bucket; only the `spec.backups.pgbackrest.global` field has changed to point to a different path. This will ensure that the new `elephant` cluster will be pre-populated with the data from `hippo`'s backups, but will backup to its own folders, ensuring that the original backup repository is appropriately preserved.

Deploy this manifest to create the `elephant` `Postgres` cluster. Observe that it comes up and running:

```
kubectl -n postgres-operator describe postgrescluster elephant
```

When it is ready, you will see that the number of expected instances matches the number of ready instances, e.g.:

```
Instances:
  Name:                00
  Ready Replicas:      1
  Replicas:            1
  Updated Replicas:    1
```

The previous example shows how to use an existing S3 repository to pre-populate a PostgresCluster while using a new S3 repository for backing up. But PostgresClusters that use cloud-based data sources can also use local repositories.

For example, assuming a PostgresCluster called **rhino** that was meant to pre-populate from the original **hippo** PostgresCluster, the manifest would look like this:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: rhino
spec:
  postgresVersion: 17
  dataSource:
    pgbackrest:
      stanza: db
      configuration:
        - secret:
            name: pgo-s3-creds
      global:
        repo1-path: /pgbackrest/postgres-operator/hippo/repo1
      repo:
        name: repo1
        s3:
          bucket: 'my-bucket'
          endpoint: 's3.ca-central-1.amazonaws.com'
          region: 'ca-central-1'
  instances:
    - dataVolumeClaimSpec:
        accessModes:
          - 'ReadWriteOnce'
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - 'ReadWriteOnce'
              resources:
                requests:
                  storage: 1Gi
```

Next Steps

Now that we've learned the basics of setting up a cluster and have seen how to set up backups and disaster recovery, let's look at some Day Two Tasks such as making our cluster highly available, enabling a monitoring stack, and making customizations to our cluster.

WAL Management

In Crunchy Postgres for Kubernetes, archiving of the write-ahead log (WAL) is handled by pgBackRest, the same tool used to manage backups and restores. It's important to keep an archive of WAL for recovery purposes. A backup only ever captures the state of your database *on disk*. WAL captures the state of your database *in memory*.

Together, a backup and WAL can restore your database to its production state just before an outage.

Keeping a separate WAL volume

It's best to keep WAL on a separate volume from your `pgdata` directory. Doing so is more performant and prevents disk exhaustion on the `pgdata` volume from affecting WAL storage. You can provision a dedicated WAL volume like this:

```
spec:
  instances:
    - name: instance1
      walVolumeClaimSpec:
        accessModes:
          - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
```

WAL archiving

When pgBackRest archives WAL, log files get copied out of the `wal` directory and compressed at their destination. Postgres can then recycle WAL files in the `wal` directory, reducing the amount of space required for normal operations. Crunchy Postgres for Kubernetes v5.7+ configures pgBackRest to use asynchronous archiving for robust and performant offloading of WAL.

WAL can be stored in either mounted storage or a cloud-based object store. A mounted volume can be allocated like this:

```
spec:
  backups:
    pgBackRest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - "ReadWriteOnce"
              resources:
                requests:
                  storage: 1Gi
```

An object store, like s3, can be allocated like this:

```
spec:
  backups:
```

```
pgBackRest :
  repos:
  - name: repo1
    s3:
      bucket: "the-name-of-your-bucket"
      endpoint: "s3.us-east-1.amazonaws.com"
      region: "us-east-1"
```

For details on configuring different object stores and using multiple repos, see our tutorial on [Backup Configuration](#).

For further information on the relationship between WAL retention and backup retention, see the [--repo-retention-archive](#) section of the [pgBackRest configuration docs](#).

If for any reason you would like to opt out of asynchronous archiving, apply the following configuration:

```
spec:
  backups:
    pgbackrest:
      global:
        archive-async: n
```

WAL archive logging

Logs for WAL archiving can be found in `pgdata/pgbackrest/log/`. The log level can be adjusted through pgBackRest's global settings.

```
spec:
  backups:
    pgbackrest:
      global:
        log-level-console: warn
        log-level-file: warn
```

Log levels less than `error` are not recommended. See the [pgBackRest Configuration docs](#) for further details.

Day Two Tasks

Working through the Basic Setup showed you how to install Crunchy Postgres for Kubernetes and how to get a Postgres cluster up and running.

You now have the power to deploy a Postgres cluster to production running on Kubernetes! However there are a few questions you should be asking yourself.

- Am I prepared to monitor and support this cluster?
- How will I know if my cluster is running out of resources?
- How can I protect against infrastructure outages?
- What if I need to change some configuration settings on my running cluster?

In the Day Two tutorials, we will show you how to install our monitoring stack, so that you can track the health of your cluster and anticipate problems before they arise. In our High Availability tutorial, we'll show you how easy it is to add replicas to your cluster and tailor your topology to mitigate downtime. Do you need to further customize your cluster for situations we haven't covered? We will show you how to Customize a Postgres Cluster.

High Availability

Postgres is known for its reliability: it is very stable and typically "just works." However, there are many things that can happen in a distributed environment like Kubernetes that can affect Postgres uptime, including:

- The database storage disk fails or some other hardware failure occurs
- The network on which the database resides becomes unreachable
- The host operating system becomes unstable and crashes
- A key database file becomes corrupted
- A data center is lost
- A Kubernetes component (e.g. a Service) is accidentally deleted

There may also be downtime events that are due to the normal case of operations, such as performing a minor upgrade, security patching of operating system, hardware upgrade, or other maintenance.

The good news: PGO is prepared for this, and your Postgres cluster is protected from many of these scenarios. However, to maximize your high availability (HA), let's first scale up your Postgres cluster.

HA Postgres: Adding Replicas to your Postgres Cluster

PGO provides several ways to add replicas to make a HA cluster:

- Increase the `spec.instances.replicas` value
- Add an additional entry in `spec.instances`

For the purposes of this tutorial, we will go with the first method and set `spec.instances.replicas` to `2`. Your manifest should look similar to:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  postgresVersion: 17
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
          - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      repos:
```

```
- name: repo1
  volume:
    volumeClaimSpec:
      accessModes:
        - "ReadWriteOnce"
      resources:
        requests:
          storage: 1Gi
```

Apply these updates to your Postgres cluster with the following command:

```
kubectl apply -k kustomize/postgres
```

Within moments, you should see a new Postgres instance initializing! You can see all of your Postgres Pods for the **hippo** cluster by running the following command:

```
kubectl -n postgres-operator get pods --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instance-set
```

Let's test our high availability set up.

Testing Your HA Cluster

An important part of building a resilient Postgres environment is testing its resiliency, so let's run a few tests to see how PGO performs under pressure!

Test #1: Remove a Service

Let's try removing the primary Service that our application is connected to. This test does not actually require a HA Postgres cluster, but it will demonstrate PGO's ability to react to environmental changes and heal things to ensure your applications can stay up.

Recall in connecting a Postgres cluster that we observed the Services that PGO creates. For example:

```
kubectl -n postgres-operator get svc --selector=postgres-operator.crunchydata.com/cluster=hippo
```

yields something similar to:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
hippo-ha	ClusterIP	10.103.73.92	<none>	5432/TCP	4h8m
hippo-ha-config	ClusterIP	None	<none>	<none>	4h8m
hippo-pods	ClusterIP	None	<none>	<none>	4h8m
hippo-primary	ClusterIP	None	<none>	5432/TCP	4h8m
hippo-replicas	ClusterIP	10.98.110.215	<none>	5432/TCP	4h8m

We also mentioned that the application is connected to the **hippo-primary** Service. What happens if we were to delete this Service?

```
kubectl -n postgres-operator delete svc hippo-primary
```

This would seem like it could create a downtime scenario, but run the above selector again:

```
kubectl -n postgres-operator get svc --selector=postgres-operator.crunchydata.com/cluster=hippo
```

You should see something similar to:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
hippo-ha	ClusterIP	10.103.73.92	<none>	5432/TCP	4h8m
hippo-ha-config	ClusterIP	None	<none>	<none>	4h8m
hippo-pods	ClusterIP	None	<none>	<none>	4h8m
hippo-primary	ClusterIP	None	<none>	5432/TCP	3s
hippo-replicas	ClusterIP	10.98.110.215	<none>	5432/TCP	4h8m

Wow -- PGO detected that the primary Service was deleted and it recreated it! Based on how your application connects to Postgres, it may not have even noticed that this event took place!

Now let's try a more extreme downtime event.

Test #2: Remove the Primary StatefulSet

[StatefulSets](#) are a Kubernetes object that provide helpful mechanisms for managing Pods that interface with stateful applications, such as databases. They provide a stable mechanism for managing Pods to help ensure data is retrievable in a predictable way.

What happens if we remove the StatefulSet that is pointed to the Postgres primary pod? First, let's determine which Pod is the primary. We'll store it in an environment variable for convenience. If you are using Bash, you can run the following command:

```
PRIMARY_POD=$(kubectl -n postgres-operator get pods --selector=postgres-operator.crunchydata.com/role=master -o jsonpath='{.items[*].metadata.labels.postgres-operator.crunchydata.com/instance}')
```

Inspect the environment variable to see which Pod is the current primary:

```
echo $PRIMARY_POD
```

This should yield something similar to:

```
hippo-instance1-zj5s
```

The equivalent commands in Powershell would be:

```
$env:PRIMARY_POD=(kubectl -n postgres-operator get pods --selector=postgres-operator.crunchydata.com/role=master -o jsonpath='{.items[*].metadata.labels.postgres-operator.crunchydata.com/instance}')
```

```
echo $env:PRIMARY_POD
```

We can use the value above to delete the StatefulSet associated with the current Postgres primary instance. If you are using Bash:

```
kubectl delete sts -n postgres-operator "${PRIMARY_POD}"
```

In Powershell:

```
kubectl delete sts -n postgres-operator "$env:PRIMARY_POD"
```

Let's see what happens. Try getting all of the StatefulSets for the Postgres instances in the **hippo** cluster:

```
kubectl get sts -n postgres-operator --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instance
```

You should see something similar to:

NAME	READY	AGE
hippo-instance1-6kbw	1/1	15m
hippo-instance1-zj5s	0/1	1s

PGO recreated the StatefulSet that was deleted! After this "catastrophic" event, PGO proceeds to heal the Postgres instance so it can rejoin the cluster. We cover the high availability process in greater depth later in the documentation.

What about the other instance? We can see that it became the new primary with the following command:

```
kubectl -n postgres-operator get pods --selector=postgres-operator.crunchydata.com/role=master -o jsonpath='{.items[*].metadata.labels.postgres-operator.crunchydata.com/instance}'
```

which should show something similar to:

```
hippo-instance1-6kbw
```

You can test that the failover occurred in a few ways. You can connect to the example Keycloak application that we deployed in the Connect an Application tutorial. Based on Keycloak's connection retry logic, you may need to wait a moment for it to reconnect, but you will see it's connected and able to read and write data. You can also connect to the Postgres instance directly and run the following command:

```
SELECT NOT pg_catalog.pg_is_in_recovery() is_primary;
```

If it returns **true** (or **t**), then the Postgres instance is a primary!

What if PGO was down during the downtime event? Failover would still occur: the Postgres HA system works independently of PGO and can maintain its own uptime. PGO will still need to assist with some of the healing aspects, but your application will still maintain read/write connectivity to your Postgres cluster!

Failsafe Mode

We've seen how the self-healing abilities of Crunchy Postgres for Kubernetes can protect your cluster from downtime. But what happens if your cluster's connection to Kubernetes itself is disrupted? Normally your primary would be demoted and all of your Postgres instances would go into a read-only state. Shifting the primary into a read-only state protects you from a split-brain scenario, where multiple instances believe they're the primary and your data becomes inconsistent.

While demotion of the primary is a nice safeguard, it's possible for you to prevent demotion and still avoid a split-brain scenario by running in failsafe mode. Enable failsafe mode like this:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  patroni:
    dynamicConfiguration:
      failsafe_mode: true
```

For more information on how Crunchy Postgres for Kubernetes maintains knowledge of which instance is the leader, see our documentation on high availability architecture.

Synchronous Replication

PostgreSQL supports synchronous replication, which is a replication mode designed to limit the risk of transaction loss. Synchronous replication waits for a transaction to be written to at least one additional server before it considers the transaction to be committed. For more information on synchronous replication, please read about PGO's high availability architecture

To add synchronous replication to your Postgres cluster, you can add the following to your spec:

```
spec:
  patroni:
    dynamicConfiguration:
      synchronous_mode: true
```

While PostgreSQL defaults [synchronous_commit](#) to `on`, you may also want to explicitly set it, in which case the above block becomes:

```
spec:
  config:
    parameters:
      synchronous_commit: 'on'
  patroni:
    dynamicConfiguration:
      synchronous_mode: true
```

Note that Patroni, which manages many aspects of the cluster's availability, will favor availability over synchronicity. This means that if a synchronous replica goes down, Patroni will allow for asynchronous replication to continue as well as writes to the primary. However, if you want to disable all writing if there are no synchronous replicas available, you can enable [synchronous_mode_strict](#) like this:

```
spec:
  patroni:
    dynamicConfiguration:
      synchronous_mode: true
      synchronous_mode_strict: true
```

Affinity

[Kubernetes affinity](#) rules, which include Pod anti-affinity and Node affinity, can help you to define where you want your workloads to reside. Pod anti-affinity is important for high availability: when used correctly, it ensures that your Postgres instances are distributed amongst different Nodes. Node affinity can be used to assign instances to specific Nodes, e.g. to utilize hardware that's optimized for databases.

Understanding Pod Labels

PGO sets up several labels for Postgres cluster management that can be used for Pod anti-affinity or affinity rules in general. These include:

- **postgres-operator.crunchydata.com/cluster**: This is assigned to all managed Pods in a Postgres cluster. The value of this label is the name of your Postgres cluster, in this case: **hippo**.
- **postgres-operator.crunchydata.com/instance-set**: This is assigned to all Postgres instances within a group of **spec.instances**. In the example above, the value of this label is **instance1**. If you do not assign a label, the value is automatically set by PGO using a **NN** format, e.g. **00**.
- **postgres-operator.crunchydata.com/instance**: This is a unique label assigned to each Postgres instance containing the name of the Postgres instance.

Let's look at how we can set up affinity rules for our Postgres cluster to help improve high availability.

Pod Anti-affinity

Kubernetes has two types of Pod anti-affinity:

- Preferred: With preferred (**preferredDuringSchedulingIgnoredDuringExecution**) Pod anti-affinity, Kubernetes will make a best effort to schedule Pods matching the anti-affinity rules to different Nodes. However, if it is not possible to do so, then Kubernetes may schedule one or more Pods to the same Node.
- Required: With required (**requiredDuringSchedulingIgnoredDuringExecution**) Pod anti-affinity, Kubernetes mandates that each Pod matching the anti-affinity rules **must** be scheduled to different Nodes. However, a Pod may not be scheduled if Kubernetes cannot find a Node that does not contain a Pod matching the rules.

There is a trade-off with these two types of pod anti-affinity: while "required" anti-affinity will ensure that all the matching Pods are scheduled on different Nodes, if Kubernetes cannot find an available Node, your Postgres instance may not be scheduled. Likewise, while "preferred" anti-affinity will make a best effort to schedule your Pods on different Nodes, Kubernetes may compromise and schedule more than one Postgres instance of the same cluster on the same Node.

By understanding these trade-offs, the makeup of your Kubernetes cluster, and your requirements, you can choose the method that makes the most sense for your Postgres deployment. We'll show examples of both methods below!

Using Preferred Pod Anti-Affinity

First, let's deploy our Postgres cluster with preferred Pod anti-affinity. Note that if you have a single-node Kubernetes cluster, you will not see your Postgres instances deployed to different nodes. However, your Postgres instances *will* be deployed.

We can set up our HA Postgres cluster with preferred Pod anti-affinity like so:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
```

```

name: hippo
spec:
  postgresVersion: 17
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
          - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
            - weight: 1
              podAffinityTerm:
                topologyKey: kubernetes.io/hostname
                labelSelector:
                  matchLabels:
                    postgres-operator.crunchydata.com/cluster: hippo
                    postgres-operator.crunchydata.com/instance-set: instance1
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - "ReadWriteOnce"
              resources:
                requests:
                  storage: 1Gi

```

Apply those changes in your Kubernetes cluster.

Let's take a closer look at this section:

```

affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        podAffinityTerm:
          topologyKey: kubernetes.io/hostname
          labelSelector:
            matchLabels:
              postgres-operator.crunchydata.com/cluster: hippo
              postgres-operator.crunchydata.com/instance-set: instance1

```

`spec.instances.affinity.podAntiAffinity` follows the standard Kubernetes [Pod anti-affinity spec](#). The values for the `matchLabels` are derived from what we described in the previous section: `postgres-operator.crunchydata.com/cluster` is set to our cluster name of `hippo`, and `postgres-operator.crunchydata.com/instance-set` is set to the instance set name of `instance1`. We choose a `topologyKey` of `kubernetes.io/hostname`, which is standard in Kubernetes clusters.

Preferred Pod anti-affinity will perform a best effort to schedule your Postgres Pods to different nodes. Let's see how you can require your Postgres Pods to be scheduled to different nodes.

Using Required Pod Anti-Affinity

Required Pod anti-affinity forces Kubernetes to schedule your Postgres Pods to different Nodes. Note that if Kubernetes is unable to schedule all Pods to different Nodes, some of your Postgres instances may become unavailable.

Using the previous example, let's indicate to Kubernetes that we want to use required Pod anti-affinity for our Postgres clusters:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  postgresVersion: 17
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
          - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - topologyKey: kubernetes.io/hostname
              labelSelector:
                matchLabels:
                  postgres-operator.crunchydata.com/cluster: hippo
                  postgres-operator.crunchydata.com/instance-set: instance1
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - "ReadWriteOnce"
              resources:
                requests:
                  storage: 1Gi
```

Apply those changes in your Kubernetes cluster.

If you are in a single Node Kubernetes clusters, you will notice that not all of your Postgres instance Pods will be scheduled. This is due to the **requiredDuringSchedulingIgnoredDuringExecution** preference. However, if you have enough Nodes available, you will see the Postgres instance Pods scheduled to different Nodes:

```
kubectl get pods -n postgres-operator -o wide --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instance
```

Node Affinity

Node affinity can be used to assign your Postgres instances to Nodes with specific hardware or to guarantee a Postgres instance resides in a specific zone. Node affinity can be set within the `spec.instances.affinity.nodeAffinity` attribute, following the standard Kubernetes [node affinity spec](#).

Let's see an example with required Node affinity. Let's say we have a set of Nodes that are reserved for database usage that have a label `workload-role=db`. We can create a Postgres cluster with a required Node affinity rule to scheduled all of the databases to those Nodes using the following configuration:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  postgresVersion: 17
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
          - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: workload-role
                    operator: In
                    values:
                      - db
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - "ReadWriteOnce"
              resources:
                requests:
                  storage: 1Gi
```

Pod Topology Spread Constraints

In addition to affinity and anti-affinity settings, [Kubernetes Pod Topology Spread Constraints](#) can also help you to define where you want your workloads to reside. However, while PodAffinity allows any number of Pods to be added to a qualifying topology domain, and PodAntiAffinity allows only one Pod to be scheduled into a single topology domain, topology spread constraints allow you to distribute Pods across different topology domains with a finer level of control.

API Field Configuration

The spread constraint [API fields](#) can be configured for instance, PgBouncer and pgBackRest repo host pods. The basic configuration is as follows:

```
topologySpreadConstraints:
  - maxSkew: $integer
    topologyKey: $string
    whenUnsatisfiable: $string
    labelSelector: $object
```

where "maxSkew" describes the maximum degree to which Pods can be unevenly distributed, "topologyKey" is the key that defines a topology in the Nodes' Labels, "whenUnsatisfiable" specifies what action should be taken when "maxSkew" can't be satisfied, and "labelSelector" is used to find matching Pods.

Example Spread Constraints

To help illustrate how you might use this with your cluster, we can review examples for configuring spread constraints on our Instance and pgBackRest repo host Pods. For this example, assume we have a three node Kubernetes cluster where the first node is labeled with **my-node-label=one**, the second node is labeled with **my-node-label=two** and the final node is labeled **my-node-label=three**. The label key **my-node-label** will function as our **topologyKey**. Note all three nodes in our examples will be schedulable, so a Pod could live on any of the three Nodes.

Instance Pod Spread Constraints

To begin, we can set our topology spread constraints on our cluster Instance Pods. Given this configuration

```
instances:
  - name: instance1
    replicas: 5
    topologySpreadConstraints:
      - maxSkew: 1
        topologyKey: my-node-label
        whenUnsatisfiable: DoNotSchedule
        labelSelector:
          matchLabels:
            postgres-operator.crunchydata.com/instance-set: instance1
```

we will expect 5 Instance pods to be created. Each of these Pods will have the standard **postgres-operator.crunchydata.com/instance-set: instance1** label set, so each Pod will be properly counted when determining the **maxSkew**. Since we have 3 nodes with a **maxSkew** of 1 and we've set **whenUnsatisfiable** to **DoNotSchedule**, we should see 2 Pods on 2 of the nodes and 1 Pod on the remaining Node, thus ensuring our Pods are distributed as evenly as possible.

pgBackRest Repo Pod Spread Constraints

We can also set topology spread constraints on our cluster's pgBackRest repo host pod. While we normally will only have a single pod per cluster, we could use a more generic label to add a preference that repo host Pods from different clusters are distributed among our Nodes. For example, by setting our **matchLabel** value to **postgres-operator.crunchydata.com/pgbackrest: "**and our **whenUnsatisfiable** value to **ScheduleAnyway**, we will allow our repo host Pods to be scheduled no matter what Nodes may be available, but attempt to minimize skew as much as possible.

```
repoHost:
  topologySpreadConstraints:
    - maxSkew: 1
      topologyKey: my-node-label
      whenUnsatisfiable: ScheduleAnyway
      labelSelector:
```

```
matchLabels:
  postgres-operator.crunchydata.com/pgbackrest: ""
```

Putting it All Together

Now that each of our Pods has our desired Topology Spread Constraints defined, let's put together a complete cluster definition:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  postgresVersion: 17
  instances:
    - name: instance1
      replicas: 5
      topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: DoNotSchedule
          labelSelector:
            matchLabels:
              postgres-operator.crunchydata.com/instance-set: instance1
      dataVolumeClaimSpec:
        accessModes:
          - "ReadWriteOnce"
        resources:
          requests:
            storage: 1G
  backups:
    pgbackrest:
      repoHost:
        topologySpreadConstraints:
          - maxSkew: 1
            topologyKey: my-node-label
            whenUnsatisfiable: ScheduleAnyway
            labelSelector:
              matchLabels:
                postgres-operator.crunchydata.com/pgbackrest: ""
  repos:
    - name: repo1
      volume:
        volumeClaimSpec:
          accessModes:
            - "ReadWriteOnce"
          resources:
            requests:
              storage: 1G
```

You can then apply those changes in your Kubernetes cluster.

Once your cluster finishes deploying, you can check that your Pods are assigned to the correct Nodes:

```
kubectl get pods -n postgres-operator -o wide --selector=postgres-operator.crunchydata.com/cluster=hippo
```

Next Steps

We've now seen how PGO helps your application stay "always on" with your Postgres database. Now let's see how we can monitor our Postgres cluster to detect and prevent issues from occurring.

Monitoring

While having high availability and disaster recovery systems in place helps in the event of something going wrong with your PostgreSQL cluster, monitoring helps you anticipate problems before they happen. Monitoring can also help you diagnose and resolve issues that may cause degraded performance.

The Crunchy Postgres for Kubernetes Monitoring stack is a fully integrated solution for monitoring and visualizing metrics captured from PostgreSQL clusters created using Crunchy Postgres for Kubernetes. By leveraging [pgMonitor](#) to configure and integrate the various tools, components and metrics needed to effectively monitor PostgreSQL clusters, Crunchy Postgres for Kubernetes Monitoring provides a powerful and easy-to-use solution to effectively monitor and visualize PostgreSQL database and container metrics. Included in the monitoring infrastructure are the following components:

- [pgMonitor](#) - Provides the configuration needed to enable the effective capture and visualization of PostgreSQL database metrics using the various tools comprising the PostgreSQL Operator Monitoring infrastructure
- [Grafana](#) - Enables visual dashboard capabilities for monitoring PostgreSQL clusters, specifically using Crunchy PostgreSQL Exporter data stored within Prometheus
- [Prometheus](#) - A multi-dimensional data model with time series data, which is used in collaboration with the Crunchy PostgreSQL Exporter to provide and store metrics
- [Alertmanager](#) - Handles alerts sent by Prometheus by deduplicating, grouping, and routing them to receiver integrations.

By leveraging the installation method described in this section, Crunchy Postgres for Kubernetes Monitoring can be deployed alongside Crunchy Postgres for Kubernetes.

Kustomize Install Crunchy Postgres for Kubernetes Monitoring

Examples of how to use Kustomize to install Crunchy Postgres for Kubernetes components can be found on GitHub in [the Postgres Operator examples](#) repository.

[Click here](#) to fork the repository.

Once you have forked the repo, you can download it to your working environment with a command similar to this:

```
YOUR_GITHUB_UN="$YOUR_GITHUB_USERNAME"
git clone --depth 1 "git@github.com:${YOUR_GITHUB_UN}/postgres-operator-examples.git"
cd postgres-operator-examples
```

For Powershell environments:

```
$env:YOUR_GITHUB_UN="YOUR_GITHUB_USERNAME"
git clone --depth 1 "git@github.com:$env:YOUR_GITHUB_UN/postgres-operator-examples.git"
cd postgres-operator-examples
```

You now have what you need to follow along with the installation steps.

Install the Crunchy Postgres Exporter Sidecar or OpenTelemetry Collector Sidecar

In order to export metrics from your PostgresCluster, CPK will add an exporter sidecar to some of the PostgresCluster components, depending on your chosen exporter. CPK can use the Crunchy Postgres Exporter or, if you are running CPK 5.8 or later and have the **OpenTelemetryMetrics** feature gate enabled, the OpenTelemetry collector.

Crunchy Postgres Exporter

The Crunchy Postgres Exporter sidecar can collect real-time metrics about a PostgreSQL database. Let's look at how we can add the sidecar to your cluster using the `kustomize/postgres` example in the Postgres Operator examples repository.

If you followed the Quickstart to create a Postgres cluster, go to the `kustomize/postgres/postgres.yaml` file and add the following YAML to the spec:

```
monitoring:
  pgmonitor:
    exporter: {}
```

Monitoring tools are added using the `spec.monitoring` section of the custom resource. Currently, the only monitoring tool supported is the Crunchy PostgreSQL Exporter configured with [pgMonitor](#). Save your changes and run:

```
kubectl apply -k kustomize/postgres
```

Crunchy Postgres for Kubernetes will detect the change and add the Exporter sidecar to all Postgres Pods that exist in your cluster. Crunchy Postgres for Kubernetes will also configure the Exporter to connect to the database and gather metrics. These metrics can be accessed using the Crunchy Postgres for Kubernetes Monitoring stack.

The OpenTelemetry Collector

For an in-depth look at OpenTelemetry, check out the Database Observability architecture. For this tutorial, what you need to know is that OpenTelemetry exports logs and metrics from several components of the PostgresCluster: the Postgres pods (including Postgres and Patroni), the pgbouncer pod, and (if present) the pgbackrest Repo Host pod. You can even use OpenTelemetry to export logs from a Standalone PgAdmin deployment.

At this time, OpenTelemetry export is only available in CPK 5.8 and later and is behind two feature gates: `OpenTelemetryLogs` and `OpenTelemetryMetrics`.

If your CPK is set up to allow one or both of those features gates, you can add OpenTelemetry to a PostgresCluster or PGAdmin by adding the following YAML to the spec:

```
instrumentation: {}
```

Save your changes and run:

```
kubectl apply -k kustomize/postgres
```

Crunchy Postgres for Kubernetes will detect the change and add the OpenTelemetry Collector sidecar to the correct components for your PostgresCluster or PGAdmin. With that minimal setup, the OpenTelemetry Collector will direct parsed logs to the console (accessible through **kubect1 logs**) and will expose metrics for scraping.

If you've installed the most recent CPK Monitoring stack, these metrics can be accessed in your Prometheus or Grafana.

For more custom options for the OpenTelemetry collector, see our pages on OpenTelemetry logging and OpenTelemetry metrics.

Locate a Kustomize installer for Monitoring

The Monitoring project is located in the **kustomize/monitoring** directory.

Configuration

While the default Kustomize install should work in most Kubernetes environments, it may be necessary to further customize the project according to your specific needs.

For instance, by default **fsGroup** is set to **26** for the **securityContext** defined for the various Deployments comprising the Monitoring stack:

```
securityContext:
  fsGroup: 26
```

In most Kubernetes environments this setting is needed to ensure processes within the container have the permissions needed to write to any volumes mounted to each of the Pods comprising the Monitoring stack. However, when installing in an OpenShift environment (and more specifically when using the **restricted** Security Context Constraint), the **fsGroup** setting should be removed since OpenShift will automatically handle setting the proper **fsGroup** within the Pod's **securityContext**.

Additionally, within this same section it may also be necessary to modify the **supplementalGroups** setting according to your specific storage configuration:

```
securityContext:
  supplementalGroups: 65534
```

Therefore, the following files (located under **kustomize/monitoring**) should be modified and/or patched (e.g. using additional overlays) as needed to ensure the **securityContext** is properly defined for your Kubernetes environment:

- **alertmanager/deployment.yaml**
- **grafana/deployment.yaml**
- **prometheus/deployment.yaml**

Those files should also be modified to set appropriate constraints on compute resources for the Grafana, Prometheus and/or AlertManager deployments. And to modify the configuration for the various storage resources (i.e. PersistentVolumeClaims) created by the Monitoring installer, modify the following files:

- **alertmanager/pvc.yaml**
- **grafana/pvc.yaml**

- `prometheus/pvc.yaml`

Additionally, it is also possible to further customize the configuration for the various components comprising the Monitoring stack (Grafana, Prometheus and/or AlertManager) by modifying the following configuration resources:

- `alertmanager/config/alertmanager.yaml`
- `grafana/config/crunchy_grafana_datasource.yaml`
- `prometheus/config/crunchy-alert-rules-pg.yaml`
- `prometheus/config/prometheus.yaml`

Finally, please note that the default username and password for Grafana can be updated by modifying the Secret `grafana-admin` defined in `kustomize/monitoring/grafana/kustomization.yaml`:

```
secretGenerator:
- name: grafana-admin
  literals:
    - "password=admin"
    - "username=admin"
```

Install

Once the Kustomize project has been modified according to your specific needs, Monitoring can then be installed using `kubectl` and Kustomize:

```
kubectl apply -k kustomize/monitoring
```

Once installed, use the `kubectl` port-forward [command](#) to immediately access the various Monitoring stack components. For example, to access the Grafana dashboards, use a command similar to

```
kubectl -n postgres-operator port-forward service/crunchy-grafana 3000:3000
```

and then login via a web browser pointed to `localhost:3000`.

If you are upgrading or altering a preexisting installation, see below for specific instructions for this use-case.

Install using Older Kubectl

This installer is optimized for Kustomize v4.0.5 or later, which is included in `kubectl` v1.21.

If you are using an earlier version of `kubectl` to manage your Kubernetes objects, the `kubectl apply -k kustomize/monitoring` command will produce an error:

```
Error: json: unknown field "labels"
```

To fix this error, download the most recent version of [Kustomize](#).

Once you have installed Kustomize v4.0.5 or later, you can use it to produce valid Kubernetes yaml:

```
kustomize build kustomize/monitoring
```

The output from the `kustomize build` command can be captured to a file or piped directly to `kubectl`:

```
kustomize build kustomize/monitoring | kubectl apply -f -
```

Uninstall

And similarly, once Monitoring has been installed, it can be uninstalled using **kubectl** and Kustomize:

```
kubectl delete -k kustomize/monitoring
```

Upgrading the Monitoring stack to v5.5.x

Several changes have been made to the kustomize installer for the Monitoring stack in order to make the project easier to read and modify:

- Project reorganization

The project has been reorganized so that each tranche of the Monitoring stack has its own folder. This should make it easier to find and modify the Kubernetes objects or configurations for each tranche. For example, if you want to modify the Prometheus configuration, you can find the source file in **prometheus/config/prometheus.yaml**; if you want to modify the PVC used by Prometheus, you can find the source file in **prometheus/pvc.yaml**.

- Image and configuration updating in line with pgMonitor

Crunchy Postgres for Kubernetes Monitoring used the Grafana dashboards and configuration set by the pgMonitor project. We have updated the installer to pgMonitor v4.9 settings, including updating the images for the Alertmanager, Grafana, and Prometheus Deployments.

- Regularize naming conventions

We have changed the following Kubernetes objects to regularize our installation:

- the ServiceAccount **prometheus-sa** is renamed **prometheus**
- the ClusterRole **prometheus-cr** is renamed **prometheus**
- the ClusterRoleBinding **prometheus-crb** is renamed **prometheus** (and has been updated to take into account the ClusterRole and ServiceAccount renaming)
- the ConfigMaps **alertmanager-rules-config** is renamed **alert-rules-config**
- the Secret **grafana-secret** is renamed **grafana-admin**

How to upgrade the Monitoring installation

First, verify that you are using a Monitoring installation from before these changes. To verify, you can check that the existing monitoring Deployments are lacking a **vendor** label:

```
kubectl get deployments -L vendor
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	VENDOR
crunchy-grafana	1/1	1	1	11s	
crunchy-prometheus	1/1	1	1	11s	
crunchy-alertmanager	1/1	1	1	11s	

If the **vendor** label shows **crunchydata**, then you are using an updated installer and do not need to follow the instructions here:

```
kubectl get deployments -L vendor
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	VENDOR
crunchy-grafana	1/1	1	1	16s	crunchydata
crunchy-prometheus	1/1	1	1	16s	crunchydata
crunchy-alertmanager	1/1	1	1	16s	crunchydata

Second, if you have an older version of the Monitoring stack installed, before upgrading to the new version, you should first remove the Deployments:

```
kubectl delete deployments crunchy-grafana crunchy-prometheus crunchy-alertmanager
```

Now you can install as usual:

```
kubectl apply -k kustomize/monitoring
```

This will leave some orphaned Kubernetes objects, that can be cleaned up manually without impacting performance. The objects to be cleaned up include all of the objects listed above in point 3 on Regularize naming conventions:

```
kubectl delete clusterrolebinding prometheus-crb
kubectl delete serviceaccount prometheus-sa
kubectl delete clusterrole prometheus-cr
kubectl delete configmap alertmanager-rules-config
kubectl delete secret grafana-secret
```

Alternatively, you can install the Monitoring stack with the `--prune --all` flags to remove the objects that are no longer managed by this manifest:

```
kubectl apply -k kustomize --prune --all
```

This will remove those objects that are namespaced: the ConfigMap, Secret, and ServiceAccount. To prune cluster-wide objects, see the `--prune-allowlist` flag.

Pruning is an automated feature and should be used with caution.

Helm Install Crunchy Postgres for Kubernetes Monitoring

Examples of how to use Helm to install Crunchy Postgres for Kubernetes components can be found on GitHub in [the Postgres Operator examples](#) repository.

[Click here](#) to fork this repository.

Once you have forked the repo, you can download it to your working environment with a command similar to this:

```
YOUR_GITHUB_UN="$YOUR_GITHUB_USERNAME"
git clone --depth 1 "git@github.com:${YOUR_GITHUB_UN}/postgres-operator-examples.git"
cd postgres-operator-examples
```

For Powershell environments:

```
$env:YOUR_GITHUB_UN="YOUR_GITHUB_USERNAME"
git clone --depth 1 "git@github.com:$env:YOUR_GITHUB_UN/postgres-operator-examples.git"
cd postgres-operator-examples
```

You now have what you need to follow along with the installation steps.

Install the Crunchy Postgres Exporter Sidecar or OpenTelemetry Collector Sidecar

In order to export metrics from your PostgresCluster, CPK will add an exporter sidecar to some of the PostgresCluster components, depending on your chosen exporter. CPK can use the Crunchy Postgres Exporter or, if you are running CPK 5.8 or later and have the **OpenTelemetryMetrics** feature gate enabled, the OpenTelemetry collector.

Crunchy Postgres Exporter

The Crunchy Postgres Exporter sidecar can collect real-time metrics about a PostgreSQL database. Let's look at how we can add the sidecar to your cluster using the helm/postgres example in the Postgres Operator examples repository.

Under **helm/postgres/values.yaml**, you will find various options for configuring a Crunchy Postgres for Kubernetes cluster. Uncomment the section that enables monitoring and set it to true:

```
monitoring: true
```

Then, uncomment the section that installs the Exporter sidecar:

```
imageExporter: registry.developers.crunchydata.com/crunchydata/crunchy-postgres-exporter:ubi8-x.x.x
```

If your cluster is already running through a helm installation, use **helm upgrade** to update your cluster. Otherwise, use **helm install** and you'll be ready to export metrics from your cluster.

Crunchy Postgres for Kubernetes will detect the change and add the Exporter sidecar to all Postgres Pods that exist in your cluster. Crunchy Postgres for Kubernetes will also configure the Exporter to connect to the database and gather metrics. These metrics can be accessed using the Crunchy Postgres for Kubernetes Monitoring stack.

The OpenTelemetry Collector

For an in-depth look at OpenTelemetry, check out the Database Observability architecture. For this tutorial, what you need to know is that OpenTelemetry exports logs and metrics from several components of the PostgresCluster: the Postgres pods (including Postgres and Patroni), the pgbouncer pod, and (if present) the pgbackrest Repo Host pod.

At this time, OpenTelemetry export is only available in CPK 5.8 and later and is behind two feature gates: OpenTelemetryLogs and OpenTelemetryMetrics.

Under **helm/postgres/values.yaml**, you will find various options for configuring a Crunchy Postgres for Kubernetes cluster. If your CPK is set up to allow one or both of those features gates, you can uncomment the section that enables instrumentation and set it to true:

```
instrumentation: true
```

If your cluster is already running through a helm installation, use **helm upgrade** to update your cluster. Otherwise, use **helm install** and you'll be ready to export metrics from your cluster.

Crunchy Postgres for Kubernetes will detect the change and add the OpenTelemetry Collector sidecar to the correct components for your PostgresCluster. With that minimal setup, the OpenTelemetry Collector will direct parsed logs to the console (accessible through `kubectl logs`) and will expose metrics for scraping.

If you've installed the most recent CPK Monitoring stack, these metrics can be accessed in your Prometheus or Grafana.

For more custom options for the OpenTelemetry collector, see our pages on OpenTelemetry logging and OpenTelemetry metrics.

Install directly from the registry

Crunchy Data hosts an OCI registry that `helm` can use directly. (Not all `helm` commands support OCI registries. For more information on which commands can be used, see [the Helm documentation](#).)

You can install Crunchy Postgres for Kubernetes Monitoring directly from the registry using the `helm install` command:

```
helm install crunchy oci://registry.developers.crunchydata.com/crunchydata/crunchy-monitoring
```

Or to see what values are set in the default `values.yaml` before installing, you could run a `helm show` command just as you would with any other registry:

```
helm show values oci://registry.developers.crunchydata.com/crunchydata/crunchy-monitoring
```

Once installed, use the `kubectl` port-forward [command](#) to immediately access the various Monitoring stack components. For example, to access the Grafana dashboards, use a command similar to

```
kubectl -n postgres-operator port-forward service/crunchy-grafana 3000:3000
```

Downloading from the registry

Rather than deploying directly from the Crunchy registry, you can instead use the registry as the source for the Helm chart. You might do this in order to configure the Helm chart before installing.

To do so, download the Helm chart from the Crunchy Container Registry:

```
#To pull down the most recent Helm chart
helm pull oci://registry.developers.crunchydata.com/crunchydata/crunchy-monitoring

# To pull down a specific Helm chart
helm pull oci://registry.developers.crunchydata.com/crunchydata/crunchy-monitoring --version 0.3.0
```

Once the Helm chart has been downloaded, uncompress the bundle

```
tar -xvf crunchy-monitoring-0.3.0.tgz
```

And from there, you can follow the instructions below on setting the [Configuration](#) and installing a local Helm chart.

Configuration

The **values.yaml** file for the Helm chart contains all of the available configuration settings for the Monitoring stack. The default **values.yaml** settings should work in most Kubernetes environments, but it may require some customization depending on your specific environment and needs.

For instance, it might be necessary to change the image versions for Alertmanager, Grafana, and/or Prometheus or to apply certain labels, etc. Each segment of the Monitoring stack has its own section. So if you needed to update only the Alertmanager image, you would update the **alertmanager.image** field.

Security Configuration

By default, the Crunchy Postgres for Kubernetes Monitoring Helm chart sets the **securityContext.fsGroup** to **26** for the Deployments comprising the Monitoring stack (i.e., Alertmanager, Grafana, and Prometheus).

In most Kubernetes environments this setting is needed to ensure processes within the container have the permissions needed to write to any volumes mounted to each of the Pods comprising the Monitoring stack. However, when installing in an OpenShift environment (and more specifically when using the **restricted** Security Context Constraint), the **fsGroup** setting should be removed since OpenShift will automatically handle setting the proper **fsGroup** within the Pod's **securityContext**.

The **fsGroup** setting can be removed by setting the **openShift** value to **true**. This can be done either by changing the value in the **values.yaml** file or by setting the value on the command line during installation or upgrade:

```
helm install crunchy oci://registry.developers.crunchydata.com/crunchydata/crunchy-monitoring --set openShift=true
```

If you need to make additional changes to pod's **securityContext**, it may be necessary to download the Helm chart and alter the Deployments directly rather than setting values in the **values.yaml**. For instance, if it is necessary to modify the **supplementalGroups** setting according to your specific storage configuration, you will need to update the Deployment files:

- **templates/alertmanager/deployment.yaml**
- **templates/grafana/deployment.yaml**
- **templates/prometheus/deployment.yaml**

Compute and Storage Resources Configuration

To set appropriate constraints on compute resources for the Grafana, Prometheus and/or AlertManager Deployments, update the Deployment files:

- **templates/alertmanager/deployment.yaml**
- **templates/grafana/deployment.yaml**
- **templates/prometheus/deployment.yaml**

Similarly, to modify the configuration for the various storage resources (i.e. PersistentVolumeClaims) created by the Monitoring installer, the **pvc.yaml** file can also be modified for the Alertmanager, Grafana, and Prometheus segments of the Monitoring stack.

Additional Configuration

Like the Kustomize installation, the Crunchy Postgres for Kubernetes Monitoring stack installation includes ConfigMaps with configurations for the various Deployments. It is possible to further customize the configuration for the various components comprising the Monitoring stack (Grafana, Prometheus and/or AlertManager) by modifying the configuration resources, which are located in the **config** directory:

- **alertmanager.yml**
- **crunchy-alert-rules-pg.yml**
- **crunchy_grafana_datasource.yml**
- **prometheus.yml**

If you want to make changes to the Grafana dashboards, those configurations and dashboard json files are located in the **dashboards** directory. If you wish to add a new dashboard as part of your Helm chart, you can accomplish that by putting the json file in the **dashboards** directory. All the json files in that directory are imported by the Helm chart and loaded in the Grafana configuration.

Finally, please note that the default username and password for Grafana can be updated by modifying the **values.yml**:

```
grafana:
  admin:
    password: admin
    username: admin
```

Uninstall

To uninstall the Monitoring stack, use the **helm uninstall** command:

```
helm uninstall crunchy -n $NAMESPACE
```

Next Steps

Now that we can monitor our cluster, it's a good time to see how we can customize Postgres cluster configuration. If your monitoring stack needs further configuration, see our docs on Exporter Configuration and Monitoring Architecture.

Customize a Postgres Cluster

Postgres is known for its ease of customization; PGO helps you to roll out changes efficiently and without disruption. Let's see how we can easily tweak our Postgres configuration.

Custom Postgres Configuration

Part of the trick of managing multiple instances in a Postgres cluster is ensuring all of the configuration changes are propagated to each of them. This is where PGO helps: when you make a Postgres configuration change for a cluster, PGO will apply it to all of the Postgres instances.

For example, let's say we wanted to tweak the Postgres settings **max_parallel_workers**, **max_worker_processes**, **shared_buffers**, and **work_mem** while also allowing an MD5 password for a legacy application. We can do this in the

spec.authentication and **spec.config** sections and the changes will be applied to all instances. Here is an example updated manifest that tweaks those settings:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  postgresVersion: 17
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
          - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - "ReadWriteOnce"
              resources:
                requests:
                  storage: 1Gi
  authentication:
    rules:
      - connection: hostssl
        users: [ legacy-app ]
        method: md5
      - connection: hostssl
        method: scram-sha-256
  config:
    parameters:
      max_parallel_workers: 2
      max_worker_processes: 2
      shared_buffers: 1GB
      work_mem: 2MB
```

In particular, we added the following to **spec**:

```
authentication:
  rules:
    - connection: hostssl
      users: [ legacy-app ]
      method: md5
    - connection: hostssl
      method: scram-sha-256
config:
  parameters:
    max_parallel_workers: 2
    max_worker_processes: 2
    shared_buffers: 1GB
    work_mem: 2MB
```

Apply these updates to your Postgres cluster with the following command:

```
kubectl apply -k customize/postgres
```

PGO will go and apply these settings, restarting each Postgres instance when necessary. You can verify that the changes are present using the Postgres **SHOW** command, e.g.

```
SHOW work_mem;
```

should yield something similar to:

```
work_mem
-----
2MB
```

Postgres Instance Log Configuration

By default, the Patroni logs for the Postgres instance Pods will be sent to stdout so that they can be viewed by executing a command such as

```
kubectl -n postgres-operator logs hippo-instance1-abcd-0
```

These logs are stored as described in the Kubernetes Logging Architecture [documentation](#). While this approach works for many users, it can limit your long term storage of the Patroni logs.

Instead, to store the Patroni logs on the pgData storage volume of the Postgres Instance Pod, you can configure the **spec.patroni.logging** section of your PostgresCluster manifest. For example, with this configuration

```
patroni:
  logging:
    level: "DEBUG"
    storageLimit: 100M
```

the Patroni logs will be set to the **DEBUG** level and space used will be limited to 100 MB. By default, these logs will be stored as JSON in a file named **/pgdata/patroni/log/patroni.log** until approximately half the size limit is reached. Once the **patroni.log** file reaches half the size limit, it will be renamed **patroni.log.1** and future logs will be written to a new **patroni.log** file. Once the storage limit is reached, the oldest log file will be overwritten. While the **storageLimit** setting is required, the **level** setting is optional (if not set, the log level defaults to **INFO**).

One thing to note, the Patroni [configuration settings](#) used to configure this logging behavior require a reload to take effect. If these settings are changed after the Postgres cluster is first created, they will take effect either the next time the instance Pod restarts or after a manual Patroni reload. This can be done with a command similar to

```
kubectl -n postgres-operator exec hippo-instance1-abcd-0 -- patronictl reload hippo-ha --force
```

which allows the configuration to be reloaded without restarting the instance Pod.

Customize TLS

All connections in PGO use TLS to encrypt communication between components. PGO sets up a PKI and certificate authority (CA) that allow you create verifiable endpoints. However, you may want to bring a different TLS infrastructure based upon your organizational requirements. The good news: PGO lets you do this!

If you want to use the TLS infrastructure that PGO provides, you can skip the rest of this section and move on to learning how to add custom labels.

How to Customize TLS

There are a few different TLS endpoints that can be customized for PGO, including those of the Postgres cluster and controlling how Postgres instances authenticate with each other. Let's look at how we can customize TLS by defining

- a `spec.customTLSSecret`, used to both identify the cluster and encrypt communications
- a `spec.customReplicationTLSSecret`, used for replication authentication

(For more information on the `spec.customTLSSecret` and `spec.customReplicationTLSSecret` fields, see the `PostgresCluster CRD`)

To customize the TLS for a Postgres cluster, you will need to create two Secrets in the Namespace of your Postgres cluster. One of these Secrets will be the `customTLSSecret` and the other will be the `customReplicationTLSSecret`. Both secrets contain a TLS key (`tls.key`), TLS certificate (`tls.crt`) and CA certificate (`ca.crt`) to use.

Note: If `spec.customTLSSecret` is provided you **must** also provide `spec.customReplicationTLSSecret` and both must contain the same `ca.crt`.

The custom TLS and custom replication TLS Secrets should contain the following fields (though see below for a workaround if you cannot control the field names of the Secret's `data`):

```
data:
  ca.crt: $VALUE
  tls.crt: $VALUE
  tls.key: $VALUE
```

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

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

After you create the Secrets, you can specify the custom TLS Secret in your `postgrescluster.postgres-operator.crunchydata.com` custom resource. For example, if you created a `hippo-cluster.tls` Secret and a `hippo-replication.tls` Secret, you would add them to your Postgres cluster:

```
spec:
  customTLSSecret:
    name: hippo-cluster.tls
  customReplicationTLSSecret:
    name: hippo-replication.tls
```

If you're unable to control the key-value pairs in the Secret, you can create a mapping to tell the Postgres Operator what key holds the expected value. That would look similar to this:

```
spec:
  customTLSecret:
    name: hippo.tls
    items:
      - key: <tls.crt key in the referenced hippo.tls Secret>
        path: tls.crt
      - key: <tls.key key in the referenced hippo.tls Secret>
        path: tls.key
      - key: <ca.crt key in the referenced hippo.tls Secret>
        path: ca.crt
```

For instance, if the `hippo.tls` Secret had the `tls.crt` in a key named `hippo-tls.crt`, the `tls.key` in a key named `hippo-tls.key`, and the `ca.crt` in a key named `hippo-ca.crt`, then your mapping would look like:

```
spec:
  customTLSecret:
    name: hippo.tls
    items:
      - key: hippo-tls.crt
        path: tls.crt
      - key: hippo-tls.key
        path: tls.key
      - key: hippo-ca.crt
        path: ca.crt
```

Note: Although the custom TLS and custom replication TLS Secrets share the same `ca.crt`, they do not share the same `tls.crt`:

- Your `spec.customTLSecret` TLS certificate should have a Common Name (CN) setting that matches the primary Service name. This is the name of the cluster suffixed with `-primary`. For example, for our `hippo` cluster this would be `hippo-primary`.
- Your `spec.customReplicationTLSecret` TLS certificate should have a Common Name (CN) setting that matches `crunchyrepl`, which is the preset replication user.

As with the other changes, you can roll out the TLS customizations with `kubectl apply`

Labels

There are several ways to add your own custom Kubernetes [Labels](#) to your Postgres cluster.

- Cluster: You can apply labels to any PGO managed object in a cluster by editing the `spec.metadata.labels` section of the custom resource.
- Postgres: You can apply labels to a Postgres instance set and its objects by editing `spec.instances.metadata.labels`.
- pgBackRest: You can apply labels to pgBackRest and its objects by editing `postgresclusters.spec.backups.pg-backrest.metadata.labels`.
- PgBouncer: You can apply labels to PgBouncer connection pooling instances by editing `spec.proxy.pgBouncer.metadata.labels`.

Annotations

There are several ways to add your own custom Kubernetes [Annotations](#) to your Postgres cluster.

- Cluster: You can apply annotations to any PGO managed object in a cluster by editing the `spec.metadata.annotations` section of the custom resource.
- Postgres: You can apply annotations to a Postgres instance set and its objects by editing `spec.instances.metadata.annotations`.
- pgBackRest: You can apply annotations to pgBackRest and its objects by editing `spec.backups.pgbackrest.metadata.annotations`.
- PgBouncer: You can apply annotations to PgBouncer connection pooling instances by editing `spec.proxy.pgBouncer.metadata.annotations`.

Pod Priority Classes

PGO allows you to use [pod priority classes](#) to indicate the relative importance of a pod by setting a `priorityClassName` field on your Postgres cluster. This can be done as follows:

- Instances: Priority is defined per instance set and is applied to all Pods in that instance set by editing the `spec.instances.priorityClassName` section of the custom resource.
- Dedicated Repo Host: Priority defined under the repoHost section of the spec is applied to the dedicated repo host by editing the `spec.backups.pgbackrest.repoHost.priorityClassName` section of the custom resource.
- PgBouncer: Priority is defined under the pgBouncer section of the spec and will apply to all PgBouncer Pods by editing the `spec.proxy.pgBouncer.priorityClassName` section of the custom resource.
- Backup (manual and scheduled): Priority is defined under the `spec.backups.pgbackrest.jobs.priorityClassName` section and applies that priority to all pgBackRest backup Jobs (manual and scheduled).
- Restore (data source or in-place): Priority is defined for either a "data source" restore or an in-place restore by editing the `spec.dataSource.postgresCluster.priorityClassName` section of the custom resource.
- Data Migration: The priority defined for the first instance set in the spec (array position 0) is used for the PGDATA and WAL migration Jobs. The pgBackRest repo migration Job will use the priority class applied to the repoHost.

Separate WAL PVCs

PostgreSQL commits transactions by storing changes in its [Write-Ahead Log \(WAL\)](#). Because the way WAL files are accessed and utilized often differs from that of data files, and in high-performance situations, it can be desirable to put WAL files on separate storage volume. With PGO, this can be done by adding the `walVolumeClaimSpec` block to your desired instance in your PostgresCluster spec, either when your cluster is created or anytime thereafter:

```
spec:
  instances:
    - name: instance
      walVolumeClaimSpec:
        accessModes:
          - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
```

This volume can be removed later by removing the `walVolumeClaimSpec` section from the instance. Note that when changing the WAL directory, care is taken so as not to lose any WAL files. PGO only deletes the PVC once there are no longer any WAL files on the previously configured volume.

Custom Sidecar Containers

PGO allows you to configure custom [sidecar Containers](#) for your PostgreSQL instance and pgBouncer Pods.

To use the custom sidecar features, you will need to enable them via the PGO [feature gate](#).

PGO feature gates are enabled by setting the `PGO_FEATURE_GATES` environment variable on the PGO Deployment. For a feature named 'FeatureName', that would look like

```
PGO_FEATURE_GATES="FeatureName=true"
```

Please note that it is possible to enable more than one feature at a time as this variable accepts a comma delimited list, for example:

```
PGO_FEATURE_GATES="FeatureName=true, FeatureName2=true, FeatureName3=true..."
```

Warning

Any feature name added to `PGO_FEATURE_GATES` must be defined by PGO and must be set to true or false. Any misconfiguration will prevent PGO from deploying. See the [considerations](#) below for additional guidance.

Custom Sidecar Containers for PostgreSQL Instance Pods

To configure custom sidecar Containers for any of your PostgreSQL instance Pods you will need to enable that feature via the PGO feature gate.

As mentioned above, PGO feature gates are enabled by setting the `PGO_FEATURE_GATES` environment variable on the PGO Deployment. For the PostgreSQL instance sidecar container feature, that will be

```
PGO_FEATURE_GATES="InstanceSidecars=true"
```

Once this feature is enabled, you can add your custom [Containers](#) as an array to `spec.instances.containers`. See the [custom sidecar example](#) below for more information!

Custom Sidecar Containers for pgBouncer Pods

Similar to your PostgreSQL instance Pods, to configure custom sidecar Containers for your pgBouncer Pods you will need to enable it via the PGO feature gate.

As mentioned above, PGO feature gates are enabled by setting the `PGO_FEATURE_GATES` environment variable on the PGO Deployment. For the pgBouncer custom sidecar container feature, that will be

```
PGO_FEATURE_GATES="PGBouncerSidecars=true"
```

Once this feature is enabled, you can add your custom [Containers](#) as an array to `spec.proxy.pgBouncer.containers`. See the [custom sidecar example](#) below for more information!

Custom Sidecar Example

As a simple example, consider

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: sidecar-hippo
spec:
  postgresVersion: 17
  instances:
    - name: instance1
      containers:
        - name: testcontainer
          image: mycontainer1:latest
        - name: testcontainer2
          image: mycontainer1:latest
      dataVolumeClaimSpec:
        accessModes:
          - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - "ReadWriteOnce"
              resources:
                requests:
                  storage: 1Gi
  proxy:
    pgBouncer:
      containers:
        - name: bouncertestcontainer1
          image: mycontainer1:latest
```

In the above example, we've added two sidecar Containers to the `instance1` Pod and one sidecar container to the `pgBouncer` Pod. These Containers can be defined in the manifest at any time, but the Containers will not be added to their respective Pods until the feature gate is enabled.

Considerations

- Volume mounts and other Pod details are subject to change between releases.
- The custom sidecar features are currently feature-gated. Any sidecar Containers, as well as any settings included in their configuration, are added and used at your own risk. Improperly configured sidecar Containers could impact the health and/or security of your PostgreSQL cluster!

- When adding a sidecar container, we recommend adding a unique prefix to the container name to avoid potential naming conflicts with the official PGO containers.

Database Initialization SQL

PGO can run SQL for you as part of the cluster creation and initialization process. PGO runs the SQL using the psql client so you can use meta-commands to connect to different databases, change error handling, or set and use variables. Its capabilities are described in the [psql documentation](#).

Initialization SQL ConfigMap

The Postgres cluster spec accepts a reference to a ConfigMap containing your init SQL file. Update your cluster spec to include the ConfigMap name, `spec.databaseInitSQL.name`, and the data key, `spec.databaseInitSQL.key`, for your SQL file. For example, if you create your ConfigMap with the following command:

```
kubectl -n postgres-operator create configmap hippo-init-sql --from-file=init.sql=/path/to/init.sql
```

You would add the following section to your Postgrescluster spec:

```
spec:
  databaseInitSQL:
    key: init.sql
    name: hippo-init-sql
```

Info

The ConfigMap must exist in the same namespace as your Postgres cluster.

After you add the ConfigMap reference to your spec, apply the change with `kubectl apply -k kustomize/postgres`. PGO will create your `hippo` cluster and run your initialization SQL once the cluster has started. You can verify that your SQL has been run by checking the `databaseInitSQL` status on your Postgres cluster. While the status is set, your init SQL will not be run again. You can check cluster status with the `kubectl describe` command:

```
kubectl -n postgres-operator describe postgresclusters.postgres-operator.crunchydata.com hippo
```

Warning

In some cases, due to how Kubernetes treats PostgresCluster status, PGO may run your SQL commands more than once. Please ensure that the commands defined in your init SQL are idempotent.

Now that `databaseInitSQL` is defined in your cluster status, verify database objects have been created as expected. After verifying, we recommend removing the `spec.databaseInitSQL` field from your spec. Removing the field from the spec will also remove `databaseInitSQL` from the cluster status.

PSQL Usage

PGO uses the `psql` interactive terminal to execute SQL statements in your database. Statements are passed in using standard input and the filename flag (e.g. `psql -f`).

SQL statements are executed as superuser in the default maintenance database. This means you have full control to create database objects, extensions, or run any SQL statements that you might need.

Integration with User and Database Management

If you are creating users or databases, please see the User/Database Management documentation. Databases created through the user management section of the spec can be referenced in your initialization sql. For example, if a database `zoo` is defined:

```
spec:
  users:
    - name: hippo
  databases:
    - "zoo"
```

You can connect to `zoo` by adding the following `psql` meta-command to your SQL:

```
\c zoo
create table t_zoo as select s, md5(random()::text) from generate_Series(1,5) s;
```

Transaction support

By default, `psql` commits each SQL command as it completes. To combine multiple commands into a single [transaction](#), use the `BEGIN` and `COMMIT` commands.

```
BEGIN;
create table t_random as select s, md5(random()::text) from generate_Series(1,5) s;
COMMIT;
```

PSQL Exit Code and Database Init SQL Status

The exit code from `psql` will determine when the `databaseInitSQL` status is set. When `psql` returns `0` the status will be set and SQL will not be run again. When `psql` returns with an error exit code the status will not be set. PGO will continue attempting to execute the SQL as part of its reconcile loop until `psql` returns normally. If `psql` exits with a failure, you will need to edit the file in your ConfigMap to ensure your SQL statements will lead to a successful `psql` return. The easiest way to make live changes to your ConfigMap is to use the following `kubectl edit` command:

```
kubectl -n postgres-operator edit configmap hippo-init-sql
```

Be sure to transfer any changes back over to your local file. Another option is to make changes in your local file and use `kubectl --dry-run` to create a template and pipe the output into `kubectl apply`

```
kubectl create -n postgres-operator configmap hippo-init-sql --from-file=init.sql=/path/to/init.sql --dry-run=client -o yaml | kubectl apply -f -
```

Hint

If you edit your ConfigMap and your changes aren't showing up, you may be waiting for PGO to reconcile your cluster. After some time, PGO will automatically reconcile the cluster or you can trigger reconciliation by applying any change to your cluster (e.g. with `kubectl apply -k kustomize/postgres`)

To ensure that `psql` returns a failure exit code when your SQL commands fail, set the `ON_ERROR_STOP` [variable](#) as part of your SQL file:

```
\set ON_ERROR_STOP
\echo Any error will lead to exit code 3
create table t_random as select s, md5(random()::text) from generate_series(1,5) s;
```

Troubleshooting

Changes Not Applied

If your Postgres configuration settings are not present, ensure that you are using the syntax that Postgres expects. You can see this in the [Postgres configuration documentation](#).

Next Steps

You've now seen how you can further customize your Postgres cluster. Let's move on to some administrative tasks you might need to complete while maintaining your Postgres database.

Cluster Management

Managing the lifecycle of your Postgres cluster means keeping components up-to-date with the latest bug-fixes and security patches, rotating your TLS certificates, and resizing memory and CPU as your resource needs ebb and flow. A production-grade Postgres cluster has a lot of moving pieces that need to be periodically refreshed. Crunchy Postgres for Kubernetes makes it easy with rolling updates and fine-grained controls for administering your Postgres cluster.

Administrative Tasks

Manually Restarting PostgreSQL

There are times when you might need to manually restart PostgreSQL. This can be done by adding or updating a custom annotation to the cluster's `spec.metadata.annotations` section. PGO will notice the change and perform a rolling restart.

For example, if you have a cluster named **hippo** in the namespace **postgres-operator**, all you need to do is patch the hippo PostgresCluster. In Bash, you can use the following:

```
kubectl patch postgrescluster/hippo -n postgres-operator --type merge --patch '{"spec":{"metadata":{"annotations":{"restarted":"'$(date)'"}}}}'
```

In Powershell, you would use:

```
kubectl patch postgresclusters/hippo -n postgres-operator --type merge --patch '{"spec":{"metadata":{"annotations":{"restarted":"'$(date)'"}}}}'
```

Watch your hippo cluster: you will see the rolling update has been triggered and the restart has begun.

Shutdown

You can shut down a Postgres cluster by setting the **spec.shutdown** attribute to **true**. You can do this by editing the manifest, or, in the case of the **hippo** cluster, executing a command like the below:

```
kubectl patch postgrescluster/hippo -n postgres-operator --type merge --patch '{"spec":{"shutdown": true}}'
```

In Powershell, you would execute:

```
kubectl patch postgrescluster/hippo -n postgres-operator --type merge --patch '{"spec":{"shutdown": true}}'
```

The effect of this is that all the Kubernetes workloads for this cluster are scaled to 0. You can verify this with the following command:

```
kubectl get deploy,sts,cronjob --selector=postgres-operator.crunchydata.com/cluster=hippo
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/hippo-pgbouncer	0/0	0	0	1h

NAME	READY	AGE
statefulset.apps/hippo-00-lwgx	0/0	1h

NAME	SCHEDULE	SUSPEND	ACTIVE
cronjob.batch/hippo-repo1-full	@daily	True	0

To turn a Postgres cluster that is shut down back on, you can set **spec.shutdown** to **false**.

Pausing Reconciliation and Rollout

You can pause the Postgres cluster reconciliation process by setting the **spec.paused** attribute to **true**. You can do this by editing the manifest, or, in the case of the **hippo** cluster, executing a command like the below:

```
kubectl patch postgrescluster/hippo -n postgres-operator --type merge --patch '{"spec":{"paused": true}}'
```

In Powershell environments, you would execute:

```
kubectl patch postgrescluster/hippo -n postgres-operator --type merge --patch '{"spec":{"paused": true}}'
```

Pausing a cluster will suspend any changes to the cluster's current state until reconciliation is resumed. This allows you to fully control when changes to the PostgresCluster spec are rolled out to the Postgres cluster. While paused, no statuses are updated other than the "Progressing" condition.

To resume reconciliation of a Postgres cluster, you can either set `spec.paused` to `false` or remove the setting from your manifest.

Rotating TLS Certificates

Credentials should be invalidated and replaced (rotated) as often as possible to minimize the risk of their misuse. Unlike passwords, every TLS certificate has an expiration, so replacing them is inevitable.

In fact, PGO automatically rotates the client certificates that it manages *before* the expiration date on the certificate. A new client certificate will be generated after 2/3rds of its working duration; so, for instance, a PGO-created certificate with an expiration date 12 months in the future will be replaced by PGO around the eight month mark. This is done so that you do not have to worry about running into problems or interruptions of service with an expired certificate.

Triggering a Certificate Rotation

If you want to rotate a single client certificate, you can regenerate the certificate of an existing cluster by deleting the `tls.key` field from its certificate Secret.

Is it time to rotate your PGO root certificate? All you need to do is delete the `pgo-root-cacert` secret. PGO will regenerate it and roll it out seamlessly, ensuring your apps continue communicating with the Postgres cluster without having to update any configuration or deal with any downtime.

```
kubectl delete secret pgo-root-cacert
```

Info

PGO only updates secrets containing the generated root certificate. It does not touch custom certificates.

Rotating Custom TLS Certificates

When you use your own TLS certificates with PGO, you are responsible for replacing them appropriately. Here's how.

PGO automatically detects and loads changes to the contents of PostgreSQL server and replication Secrets without downtime. You or your certificate manager need only replace the values in the Secret referenced by `spec.customTLSSecret`.

If instead you change `spec.customTLSSecret` to refer to a new Secret or new fields, PGO will perform a rolling restart.

Info

When changing the PostgreSQL certificate authority, make sure to update **customReplicationTLSSecret** as well.

PGO automatically notifies PgBouncer when there are changes to the contents of PgBouncer certificate Secrets. Recent PgBouncer versions load those changes without downtime, but versions prior to 1.16.0 need to be restarted manually. There are a few ways to restart an older version PgBouncer to reload Secrets:

- Store the new certificates in a new Secret. Edit the PostgresCluster object to refer to the new Secret, and PGO will perform a rolling restart of PgBouncer.**spec:**

```
proxy:
  pgBouncer:
    customTLSSecret:
      name: hippo.pgouncer.new.tls
```

or

- Replace the old certificates in the current Secret. PGO doesn't notice when the contents of your Secret change, so you need to trigger a rolling restart of PgBouncer. Edit the PostgresCluster object to add a unique annotation. The name and value are up to you, so long as the value differs from the previous value.**spec:**

```
proxy:
  pgBouncer:
    metadata:
      annotations:
        restarted: Q1-certs
```

This **kubectl patch** command uses your local date and time. In Bash:

```
kubectl patch postgrescluster/hippo -n postgres-operator --type merge --patch '{"spec":{"proxy":{"pgBouncer":{"metadata":{"annotations":{"restarted":"'$(date)'"}}}}}}'
```

In Powershell:

```
kubectl patch postgrescluster/hippo -n postgres-operator --type merge --patch '{"spec":{"proxy":{"pgBouncer":{"metadata":{"annotations":{"restarted":"'$(date)'"}}}}}}'
```

Changing the Primary

There may be times when you want to change the primary in your HA cluster. This can be done using the **patroni.switchover** section of the PostgresCluster spec. It allows you to enable switchovers in your PostgresClusters, target a specific instance as the new primary, and run a failover if your PostgresCluster has entered a bad state.

Let's go through the process of performing a switchover!

First you need to update your spec to prepare your cluster to change the primary. Edit your spec to have the following fields:

```
spec:
  patroni:
```

```
switchover:
  enabled: true
```

After you apply this change, PGO will be looking for the trigger to perform a switchover in your cluster. You will trigger the switchover by adding the `postgres-operator.crunchydata.com/trigger-switchover` annotation to your custom resource. The best way to set this annotation is with a timestamp, so you know when you initiated the change.

For example, for our `hippo` cluster, we can run the following command to trigger the switchover:

```
kubectl annotate -n postgres-operator postgrescluster hippo postgres-operator.crunchydata.com/trigger-switchover="$(date)"
```

Hint

If you want to perform another switchover you can re-run the annotation command and add the `--overwrite` flag:

```
kubectl annotate -n postgres-operator postgrescluster hippo --overwrite postgres-operator.crunchydata.com/trigger-switchover="$(date)"
```

PGO will detect this annotation and use the Patroni API to request a change to the current primary!

The roles on your database instance Pods will start changing as Patroni works. The new primary will have the `master` role label, and the old primary will be updated to `replica`.

The status of the switch will be tracked using the `status.patroni.switchover` field. This will be set to the value defined in your trigger annotation. If you use a timestamp as the annotation this is another way to determine when the switchover was requested.

After the instance Pod labels have been updated and `status.patroni.switchover` has been set, the primary has been changed on your cluster!

Info

After changing the primary, we recommend that you disable switchovers by setting `spec.patroni.switchover.enabled` to false or remove the field from your spec entirely. If the field is removed the corresponding status will also be removed from the PostgresCluster.

Targeting an instance

Another option you have when switching the primary is providing a target instance as the new primary. This target instance will be used as the candidate when performing the switchover. The `spec.patroni.switchover.targetInstance` field takes the name of the instance that you are switching to.

This name can be found in a couple different places; one is as the name of the StatefulSet and another is on the database Pod as the `postgres-operator.crunchydata.com/instance` label. The following commands can help you determine who is the current primary and what name to use as the `targetInstance`:

```
kubectl get pods -l postgres-operator.crunchydata.com/cluster=hippo -L postgres-operator.crunchydata.com/instance -L postgres-operator.crunchydata.com/role
```

NAME	READY	STATUS	RESTARTS	AGE	INSTANCE	ROLE
hippo-instance1-jdb5-0	3/3	Running	0	2m47s	hippo-instance1-jdb5	master
hippo-instance1-wm5p-0	3/3	Running	0	2m47s	hippo-instance1-wm5p	replica

In our example cluster **hippo-instance1-jdb5** is currently the primary meaning we want to target **hippo-instance1-wm5p** in the switchover. Now that you know which instance is currently the primary and how to find your **targetInstance**, let's update your cluster spec:

```
spec:
  patroni:
    switchover:
      enabled: true
      targetInstance: hippo-instance1-wm5p
```

After applying this change you will once again need to trigger the switchover by annotating the PostgresCluster (see above commands). You can verify the switchover has completed by checking the Pod role labels and **status.patroni.switchover**.

Failover

Finally, we have the option to failover when your cluster has entered an unhealthy state. The only spec change necessary to accomplish this is updating the **spec.patroni.switchover.type** field to the **Failover** type. One note with this is that a **targetInstance** is required when performing a failover. Based on the example cluster above, assuming **hippo-instance1-wm5p** is still a replica, we can update the spec:

```
spec:
  patroni:
    switchover:
      enabled: true
      targetInstance: hippo-instance1-wm5p
      type: Failover
```

Apply this spec change and your PostgresCluster will be prepared to perform the failover. Again you will need to trigger the switchover by annotating the PostgresCluster (see above commands) and verify that the Pod role labels and **status.patroni.switchover** are updated accordingly.

Warning

Errors encountered in the switchover process can leave your cluster in a bad state. If you encounter issues, found in the operator logs, you can update the spec to fix the issues and apply the change. Once the change has been applied, PGO will attempt to perform the switchover again.

Next Steps

We've covered a lot in terms of building, maintaining, scaling, customizing, and restarting our Postgres cluster. However, there may come a time where we need to resize our Postgres cluster. How do we do that?

Resize a Postgres Cluster

You did it -- the application is a success! Traffic is booming, so much so that you need to add more resources to your Postgres cluster. However, you're worried that any resize operation may cause downtime and create a poor experience for your end users.

This is where PGO comes in: PGO will help orchestrate rolling out any potentially disruptive changes to your cluster to minimize or eliminate and downtime for your application. To do so, we will assume that you have deployed a high availability Postgres cluster as described in the Day Two Tasks tutorial.

Let's dive in.

Resize Memory and CPU

Memory and CPU resources are an important component for vertically scaling your Postgres cluster. Coupled with tweaks to your Postgres configuration file, allocating more memory and CPU to your cluster can help it to perform better under load.

It's important for instances in the same high availability set to have the same resources. PGO lets you adjust CPU and memory within the **resources** sections of the **postgresclusters.postgres-operator.crunchydata.com** custom resource. These include:

- **spec.instances.resources** section, which sets the resource values for the PostgreSQL container, as well as any init containers in the associated pod and containers created by the **pgDataVolume** and **pgWALVolume** data migration jobs.
- **spec.instances.sidecars.replicaCertCopy.resources** section, which sets the resources for the **replica-cert-copy** sidecar container.
- **spec.monitoring.pgmonitor.exporter.resources** section, which sets the resources for the **exporter** sidecar container.
- **spec.backups.pgbackrest.repoHost.resources** section, which sets the resources for the pgBackRest repo host container, as well as any init containers in the associated pod and containers created by the **pgBackRestVolume** data migration job.
- **spec.backups.pgbackrest.sidecars.pgbackrest.resources** section, which sets the resources for the **pg-backrest** sidecar container.
- **spec.backups.pgbackrest.sidecars.pgbackrestConfig.resources** section, which sets the resources for the **pgbackrest-config** sidecar container.
- **spec.backups.pgbackrest.jobs.resources** section, which sets the resources for any pgBackRest backup job.
- **spec.backups.pgbackrest.restore.resources** section, which sets the resources for manual pgBackRest restore jobs.
- **spec.dataSource.postgresCluster.resources** section, which sets the resources for pgBackRest restore jobs created during the cloning process.
- **spec.proxy.pgBouncer.resources** section, which sets the resources for the **pgbouncer** container.

- `spec.proxy.pgBouncer.sidecars.pgBouncerConfig.resources` section, which sets the resources for the `pgbouncer-config` sidecar container.

The layout of these `resources` sections should be familiar: they follow the same pattern as the standard Kubernetes structure for setting [container resources](#). Note that these settings also allow for the configuration of [QoS classes](#).

For example, using the `spec.instances.resources` section, let's say we want to update our `hippo` Postgres cluster so that each instance has a limit of `2.0` CPUs and `4Gi` of memory. We can make the following changes to the manifest:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  postgresVersion: 17
  instances:
    - name: instance1
      replicas: 2
      resources:
        limits:
          cpu: 2.0
          memory: 4Gi
      dataVolumeClaimSpec:
        accessModes:
          - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - "ReadWriteOnce"
              resources:
                requests:
                  storage: 1Gi
```

In particular, we added the following to `spec.instances`:

```
resources:
  limits:
    cpu: 2.0
    memory: 4Gi
```

Apply these updates to your Postgres cluster with the following command:

```
kubectl apply -k kustomize/postgres
```

Now, let's watch how the rollout happens. In Bash, you can use a command like the following:

```
watch "kubectl -n postgres-operator get pods --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instance -o=json-path='{range .items[*]}{.metadata.name}{ \"\\t\" }{.metadata.labels.postgres-opera-
```

```
tor\.crunchydata\.com/role}}{\t}}{.status.phase}}{\t}}{.spec.containers[].re-
sources.limits}}{\n}}{end}'"
```

In Powershell, you can use a command like:

```
kubectl -n postgres-operator get pods --watch --selector=postgres-operator.crunchyda-
ta.com/cluster=hippo,postgres-operator.crunchydata.com/instance -o=json-
path="{range .items[*]}{.metadata.name}{'\t'}{.metadata.labels.postgres-opera-
tor\.crunchydata\.com/role}{'\t'}{.status.phase}{'\t'}{.spec.containers[].re-
sources.limits}}{\n}'"
```

Observe how each Pod is terminated one-at-a-time. This is part of a "rolling update". Because updating the resources of a Pod is a destructive action, PGO first applies the CPU and memory changes to the replicas. PGO ensures that the changes are successfully applied to a replica instance before moving on to the next replica.

Once all of the changes are applied, PGO will perform a "controlled switchover": it will promote a replica to become a primary, and apply the changes to the final Postgres instance.

By rolling out the changes in this way, PGO ensures there is minimal to zero disruption to your application: you are able to successfully roll out updates and your users may not even notice!

Resize PVC

Your application is a success! Your data continues to grow, and it's becoming apparent that you need more disk.

That's great: you can resize your PVC directly on your `postgresclusters.postgres-operator.crunchydata.com` custom resource with minimal to zero downtime.

PVC resizing, also known as [volume expansion](#), is a function of your storage class: it must support volume resizing. Additionally, PVCs can only be **sized up**: you cannot shrink the size of a PVC.

You can adjust PVC sizes on all of the managed storage instances in a Postgres instance that are using Kubernetes storage. These include:

- `spec.instances.dataVolumeClaimSpec.resources.requests.storage`: The Postgres data directory (aka your database).
- `spec.backups.pgbackrest.repos.volume.volumeClaimSpec.resources.requests.storage`: The pg-BackRest repository when using "volume" storage

The above should be familiar: it follows the same pattern as the standard [Kubernetes PVC](#) structure.

For example, let's say we want to update our `hippo` Postgres cluster so that each instance now uses a `10Gi` PVC and our backup repository uses a `20Gi` PVC. We can do so with the following markup:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  postgresVersion: 17
  instances:
    - name: instance1
      replicas: 2
      resources:
        limits:
```

```

      cpu: 2.0
      memory: 4Gi
    dataVolumeClaimSpec:
      accessModes:
      - "ReadWriteOnce"
      resources:
        requests:
          storage: 10Gi
    backups:
      pgbackrest:
        repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
              - "ReadWriteOnce"
              resources:
                requests:
                  storage: 20Gi

```

In particular, we added the following to `spec.instances`:

```

dataVolumeClaimSpec:
  resources:
    requests:
      storage: 10Gi

```

and added the following to `spec.backups.pgbackrest.repos.volume`:

```

volumeClaimSpec:
  accessModes:
  - "ReadWriteOnce"
  resources:
    requests:
      storage: 20Gi

```

Apply these updates to your Postgres cluster with the following command:

```
kubectl apply -k kustomize/postgres
```

Resize PVCs With StorageClass That Does Not Allow Expansion

Not all Kubernetes Storage Classes allow for [volume expansion](#). However, with PGO, you can still resize your Postgres cluster data volumes even if your storage class does not allow it!

Let's go back to the previous example:

```

apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  postgresVersion: 17
  instances:
  - name: instance1
    replicas: 2
    resources:
      limits:

```

```

      cpu: 2.0
      memory: 4Gi
    dataVolumeClaimSpec:
      accessModes:
      - 'ReadWriteOnce'
    resources:
      requests:
        storage: 1Gi
  backups:
    pgbackrest:
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - 'ReadWriteOnce'
            resources:
              requests:
                storage: 20Gi

```

First, create a new instance that has the larger volume size. Call this instance **instance2**. The manifest would look like this:

```

apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  postgresVersion: 17
  instances:
  - name: instance1
    replicas: 2
    resources:
      limits:
        cpu: 2.0
        memory: 4Gi
    dataVolumeClaimSpec:
      accessModes:
      - 'ReadWriteOnce'
    resources:
      requests:
        storage: 1Gi
  - name: instance2
    replicas: 2
    resources:
      limits:
        cpu: 2.0
        memory: 4Gi
    dataVolumeClaimSpec:
      accessModes:
      - 'ReadWriteOnce'
    resources:
      requests:
        storage: 10Gi
  backups:
    pgbackrest:
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - 'ReadWriteOnce'

```

```
resources:
  requests:
    storage: 20Gi
```

Take note of the block that contains **instance2**:

```
-name: instance2
  replicas: 2
  resources:
    limits:
      cpu: 2.0
      memory: 4Gi
  dataVolumeClaimSpec:
    accessModes:
      - 'ReadWriteOnce'
    resources:
      requests:
        storage: 10Gi
```

This creates a second set of two Postgres instances, both of which come up as replicas, that have a larger PVC.

Once this new instance set is available and they are caught to the primary, you can then apply the following manifest:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  postgresVersion: 17
  instances:
    - name: instance2
      replicas: 2
      resources:
        limits:
          cpu: 2.0
          memory: 4Gi
      dataVolumeClaimSpec:
        accessModes:
          - 'ReadWriteOnce'
      resources:
        requests:
          storage: 10Gi
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - 'ReadWriteOnce'
              resources:
                requests:
                  storage: 20Gi
```

This will promote one of the instances with the larger PVC to be the new primary and remove the instances with the smaller PVCs!

This method can also be used to shrink PVCs to use a smaller amount.

Troubleshooting

Postgres Pod Can't Be Scheduled

There are many reasons why a PostgreSQL Pod may not be scheduled:

- **Resources are unavailable.** Ensure that you have a Kubernetes [Node](#) with enough resources to satisfy your memory or CPU Request.
- **PVC cannot be provisioned.** Ensure that you request a PVC size that is available, or that your PVC storage class is set up correctly.

PVCs Do Not Resize

Ensure that your storage class supports PVC resizing. You can check that by inspecting the `allowVolumeExpansion` attribute:

```
kubectl get sc
```

If the storage class does not support PVC resizing, you can use the technique described above to resize PVCs using a second instance set.

Next Steps

Now that we know how to resize our Postgres clusters, let's look at how PGO handles software updates!

Apply Software Updates

Did you know that Postgres releases bug fixes [once every three months](#)? Additionally, we periodically refresh the container images to ensure the base images have the latest software that may fix some CVEs.

It's generally good practice to keep your software up-to-date for stability and security purposes, so let's learn how Crunchy Postgres for Kubernetes helps to you accept low risk, "patch" type updates.

Please note that you do not need to immediately update your Postgres and component container images following a Crunchy Postgres for Kubernetes upgrade, although it is recommended that you update your images as soon as possible to ensure you have the

latest security updates and bug fixes. This lets you choose when you want to apply updates to each of your Postgres clusters, so you can update it on your own schedule. And if you have a high availability Postgres cluster, Crunchy Postgres for Kubernetes uses a rolling update to minimize or eliminate any downtime for your application.

To find the Postgres and component images that correspond with your Crunchy Postgres for Kubernetes installation, you can

[browse the containers page](#)

in the [Crunchy Data Developer Portal](#).

Warning

The component image tagging strategy changed, starting with the v5.8.0 and v5.7.5 releases. Please see the [Components and Compatibility](#) page for more details.

Applying Minor Postgres Updates

The Postgres image is referenced using the `spec.image` and looks similar to the below (though the base image would be `ubi8` for versions below CPK v5.8.0):

```
spec:
  image: registry.developers.crunchydata.com/crunchydata/crunchy-postgres:ubi9-17.4-2513
```

Diving into the tag a bit further, you will notice the `17.4-2513` portion. This represents the Postgres minor version (`17.4`) and the patch number of the release `2513`. If the patch number is incremented (e.g. `2516`), this means that the container is rebuilt, but there are no changes to the Postgres version. If the minor version is incremented (e.g. `17.5-2316`), this means that there is a newer bug fix release of Postgres within the container.

To update the image, you just need to modify the `spec.image` field with the new image reference, e.g.

```
spec:
  image: registry.developers.crunchydata.com/crunchydata/crunchy-postgres:ubi9-17.5-2516
```

You can apply the changes using `kubectl apply`. Similar to the rolling update example when we resized the cluster, the update is first applied to the Postgres replicas, then a controlled switchover occurs, and the final instance is updated.

For the `hippo` cluster, you can see the status of the rollout by running the command below.

Bash:

```
kubectl -n postgres-operator get pods --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instance -o=json-
path='{range .items[*]}{.metadata.name}{"\t"}{.metadata.labels.postgres-opera-
tor\.crunchydata\.com/role}{"\t"}{.status.phase}{"\t"}{.spec.containers[].im-
age}{"\n"}{end}'
```

Powershell:

```
kubectl -n postgres-operator get pods --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instance -o=json-
path="{range .items[*]}{.metadata.name}{'\t'}{.metadata.labels.postgres-opera-
tor\.crunchydata\.com/role}{'\t'}{.status.phase}{'\t'}{.spec.containers[.im-
age]}{'\n'}{end}"
```

Or, by running a watch:

Bash:

```
watch "kubectl -n postgres-operator get pods --selector=postgres-operator.crunchyda-
ta.com/cluster=hippo,postgres-operator.crunchydata.com/instance -o=json-
path='{range .items[*]}{.metadata.name}{"\t"}{.metadata.labels.postgres-opera-
```

```
tor\.crunchydata\.com/role}}{\t}}{.status.phase}}{\t}}{.spec.containers[].image}}{\n}}{end}'"
```

Powershell:

```
kubectl -n postgres-operator get pods --watch --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instance -o=json-path="{range .items[*]}{.metadata.name}{\t'}{.metadata.labels.postgres-operator\.crunchydata\.com/role}{\t'}{.status.phase}{\t'}{.spec.containers[].image}}{\n}'"
```

Rolling Back Minor Postgres Updates

This methodology also allows you to rollback changes from minor Postgres updates. You can change the `spec.image` field to your desired container image. Crunchy Postgres for Kubernetes will then ensure each Postgres instance in the cluster rolls back to the desired image.

Applying Other Component Updates

There are other components that go into a Crunchy Postgres for Kubernetes Postgres cluster. These include pgBackRest, PgBouncer and others. Each one of these components has its own image: for example, you can find a reference to the pgBackRest image in the `spec.backups.pgbackrest.image` attribute.

Applying software updates for the other components in a Postgres cluster works similarly to the above. As pgBackRest and PgBouncer are Kubernetes [Deployments](#), Kubernetes will help manage the rolling update to minimize disruption.

Changing Base Images

Warning

Changing the Postgres base image from UBI 8 to UBI 9 can lead to corrupt indexes and other potential problems with your data.

A full backup is recommended prior to upgrading your base image, and you should thoroughly check and verify your data once the update is complete.

UBI9 images are only supported on CPK v5.8.0 and up.

Postgres uses locale information provided by the operating system for sorting text.

Changes to that information can lead to erroneous query results and other incorrect behavior.

Postgres is able to detect those changes and emit warnings like the following:

```
WARNING: collation "my-custom" has version mismatch
DETAIL: The collation in the database was created using version 2.34, but the operating system provides version 2.28.
HINT: Rebuild all objects affected by this collation and run ALTER COLLATION "my-custom" REFREEZE.
WARNING: database "postgres" has a collation version mismatch
WARNING: template database "template1" has a collation version mismatch
DETAIL: The database was created using collation version 2.34, but the operating system provides version 2.28.
```

```
WARNING: database "postgres" has a collation version mismatch
WARNING: template database "template1" has a collation version mismatch
DETAIL: The database was created using collation version 2.34, but the operating system provides version 2.28.
```

HINT: Rebuild all objects in this database that use the default collation and run ALTER DATABASE "plate1" REFRESH COLLATION VERSION

These warnings indicate you should **REINDEX** your databases to avoid any **possibility of data corruption**.

At a minimum, in every database in an affected cluster, run **REINDEX DATABASE** followed by **ALTER DATABASE**

The Postgres documentation has more detailed instructions for [custom collations](#).

You can always verify your indexes using the included [amcheck extension](#).

Guides

This section contains guides on handling various scenarios when managing Postgres clusters using PGO, the Postgres Operator. These include step-by-step instructions for situations such as migrating data to a PGO managed Postgres cluster or upgrading from an older version of PGO.

These guides are in no particular order: choose the guide that is most applicable to your situation.

If you are looking for how to manage most day-to-day Postgres scenarios, we recommend first going through the Tutorial.

Auto-Growable Disk

You may be nearing your disk space limit and not know it. Once you hit that limit, you're looking at downtime. Monitoring and a scalable storage class are great tools to avoid disk-full errors. But sometimes, the best solution is not having to think about it. Enabling auto-grow will let Crunchy Postgres for Kubernetes do the work for you. Auto-grow will watch your data directory and grow your disk. You set the limit on growth and Crunchy Postgres for Kubernetes does the rest.

Prerequisites

To use this feature, you'll need a storage provider that supports dynamic scaling. To see if your volume can expand, run the following

command on your [storage class](#) and see if the **allowVolumeExpansion** field is set to **true**:

Bash:

```
#Check whether your storage classes are expandable
kubectl describe storageclass | grep -e Name -e Expansion
```

Powershell:

```
kubectl describe storageclass | Select-String -Pattern @("Name", "Expansion") -CaseSensitive
```

Enabling Auto-Grow

To enable Crunchy Postgres for Kubernetes' auto-grow feature, you need to activate the Autogrow feature gate. PGO feature gates are enabled by setting the

PGO_FEATURE_GATES environment variable on the PGO Deployment.

```
PGO_FEATURE_GATES="AutoGrowVolumes=true"
```

Please note that it is possible to enable more than one feature at a time as this variable accepts a comma delimited list. For example, to enable multiple features, you would set **PGO_FEATURE_GATES** like so:

```
PGO_FEATURE_GATES="FeatureName=true, FeatureName2=true, FeatureName3=true..."
```

Additionally, you will need to set a limit for volume expansion to prevent the volume from growing beyond a specified size. Don't worry if you need to up the limit. Just change the limit field in your spec and re-apply. For example you could define the following in your spec:

```
spec:
  postgresVersion: 17
  instances:
    - name: instance1
      dataVolumeClaimSpec:
        accessModes:
          - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
          limits:
            storage: 5Gi # Set the limit on disk growth.
```

Warning

- Once auto-grow has expanded your volume request, **requests.storage** in your manifest will no longer be accurate.

Examine the pgdata PVC for instance1 and update your manifest, if you want to re-apply your manifest.

Nothing bad will happen if you don't update **requests.storage**, though you will likely receive a warning.

- Some storage services may place a limit on the number of volume expansions you can perform within some period of time. With that in mind, it remains a good idea to start with a resource request of what you think you'll actually need.

How It Works

After enabling the feature gate and setting the growth limit, Crunchy Postgres for Kubernetes will monitor your disk usage. When the disk is 75% full,

a request will be sent to expand your disk by 50%. In processing this request, Kubernetes will likely round the figure up to the nearest Gi.

Info

When scaling up your PVC, Crunchy Postgres for Kubernetes will make a precise request in Mi. But, your storage solution may round up that request to the nearest Gi.

An event will be logged when the disk starts growing. Look out for notifications indicating "expansion requested" and check your PVC status for completion.

You can grow up to the limit. Beyond that, you'll see an event alerting you that the volume can't be expanded beyond the limit.

Downsizing

In the event that your volume has grown larger than what you need, you can scale down to a smaller disk allocation by adding

a second instance set with a smaller storage request. The steps we'll follow are similar to what we describe in our tutorial [Resize PVC](#), which you may want to review for further background.

Let's assume that you've defined a **PostgresCluster** similar to what was described earlier:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  postgresVersion: 17
  instances:
    - name: instance1
      dataVolumeClaimSpec:
        accessModes:
          - "ReadWriteOnce"
        resources:
          requests:
            storage: 2Gi # Assume this number has been set correctly, following disk expansion
          limit:
            storage: 5Gi
```

Imagine that your volumes have grown to 2Gi and you want to downsize to 1Gi. You'll want to be sure that 1Gi is enough space and that

you won't have to scale up immediately after downsizing. If you exec into your instance **Pod**, you can use a tool like **df** to check usage in the **/pgdata** directory. Once you're confident in your estimate, add the following to the list of instances in your

spec, but do not remove the existing instance set.

```
name: instance2
dataVolumeClaimSpec:
  accessModes:
    - "ReadWriteOnce"
  resources:
    requests:
      storage: 1Gi # Set an appropriate, smaller request here.
```

Notice that **resources.limit** has not been set. By leaving **resources.limit** unset, you have disengaged auto-grow for this instance set.

Apply your manifest and confirm that your data has replicated to the new instance set. Once your data is synced, you can remove instance1 from the list of instances and apply again to remove the old instance set from your cluster.

Creating and Managing a Bridge Postgres Cluster

Overview

The **CrunchyBridgeCluster** API introduces a Kubernetes-native method for provisioning [Crunchy Bridge](#) clusters with Crunchy Postgres for Kubernetes (CPK). This integration allows you to use familiar Kubernetes tools such as **kubectl**, **kustomize**, ArgoCD, and more, streamlining the provisioning process for Crunchy Bridge clusters.

A key distinction of the **CrunchyBridgeCluster** API, compared to the **PostgresCluster** API, is that Crunchy Bridge is fully managed and takes care of all PostgreSQL workloads. As a result, you won't see running Pods as you would with **PostgresClusters**, rendering traditional **kubectl** commands for pod monitoring less relevant.

To ensure you maintain clear visibility into your Crunchy Bridge clusters, the API emphasizes providing detailed status and condition information within the **CrunchyBridgeCluster** custom resource. This allows for comprehensive monitoring and management through Kubernetes-native tools.

Getting Started

Using the **CrunchyBridgeCluster** API is straightforward and involves a few key steps:

- **Setting up your Crunchy Bridge account:**
- **Account:** You will need a [Crunchy Bridge account](#) to get started.
- **Teams:** You may want to create a [team](#) for collaborating with others. You will need to know either your personal or group team id. You can find the team id in the URL after selecting the team that you wish to use, or via a curl to the [Crunchy Bridge API](#).
- **Payment:** You will need an [active payment method](#). This can be created from My Account > Billing > Invoices. Crunchy Bridge bills prorated fees for database services, prorated down to the second like other cloud resources.
- **API:** You will need to create an [API key](#).
- **Docs:** See the [Crunchy Bridge documentation](#) to understand the service and features.
- **Install the Operator:** If you do not have Crunchy Postgres for Kubernetes running, simply follow the standard installation process for CPK to set up the operator in your Kubernetes cluster.
- **Create a Kubernetes Secret:** You will need to create a secret that contains your API key and Team ID from Crunchy Bridge. Ensure this Secret is in the same namespace where CPK is installed. You can see an example of a secret here: `kubectl create secret generic crunchy-bridge-api-key -n postgres-operator --from-literal=key=<your Crunchy Bridge API key here> --from-literal=team=<your Crunchy Bridge Team ID here>`
- **Provision Crunchy Bridge Clusters:** With the Secret in place, you can begin provisioning Crunchy Bridge clusters using the CrunchyBridgeCluster API. The Postgres workload management is fully handled by Crunchy Bridge, simplifying your Kubernetes database operations.

Configuring a Crunchy Bridge cluster

When you are ready to provision a Crunchy Bridge cluster, the following spec can serve as a starting point. Each option in the spec should be reviewed and customized to fit your specific requirements. You can pre-plan machine sizing, pricing, and regions from our [Crunchy Bridge cost calculator](#).

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: CrunchyBridgeCluster
metadata:
  name: my-test-cluster
  namespace: postgres-operator
spec:
  isHa: false
  clusterName: my-test-bridge-cluster
  plan: standard-4
  majorVersion: 16
  provider: aws
  region: us-west-2
  secret: crunchy-bridge-api-key
  storage: 10Gi
```

Configuration notes:

- [High Availability](#): Set **isHa** to true if high availability is required for your workload.
- [Cluster Plan](#): Choose a plan (**standard-4** in this example) that matches your performance needs.
- [Postgres Version](#): Specify the major version of Postgres that you want to provision.
- [Provider and Region](#): Select the cloud provider and region that best suits your latency and compliance requirements.
- [Storage](#): Select the amount of storage that you require in 1 GB increments.
- **Secrets Management**: Ensure the secret is correctly configured with your API key and team id.

Updates to your Crunchy Bridge provision

Changes to the Crunchy Bridge cluster can be made by editing the spec in your manifest and re-applying it using kubectl. This will send a resize or update request to the Crunchy Bridge platform. Crunchy Bridge will stage a new machine and failover to the updated machine during your selected [maintenance](#) window. Note that provider and region cannot be changed currently. You can also delete any Crunchy Bridge cluster by deleting the crunchybridgecluster manifest from the kubernetes cluster using kubectl.

Getting Crunchy Bridge support

We are here to help you make the most out of Crunchy Bridge. Support tickets can be generated from inside your Crunchy Bridge dashboard.

The full CRD documentation for The **CrunchyBridgeCluster** API is located [here](#).

Configuring Cluster Images

Crunchy Postgres for Kubernetes installers provide default images to use in your Postgres clusters. These defaults make a patch update to your cluster as easy as upgrading your version of Crunchy Postgres for Kubernetes. To see how this works, let's take a look at how Crunchy Postgres for Kubernetes determines the images you want to use and how to configure PGO's defaults when you want to change them.

Specifying a Crunchy Postgres Version

All Crunchy Postgres for Kubernetes installers come with default images defined in PGO's Pod spec. You can either rely on these defaults or override them by setting **image** fields manually.

To tell PGO which major version of Crunchy Postgres you want installed in your cluster, you can use a manifest with **spec.postgresVersion** set to the major version and PGO will use its defaults to fulfill your request, like this:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  postgresVersion: 15
  instances:
    - name: instance1
      dataVolumeClaimSpec:
        accessModes:
          - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: registry.developers.crunchydata.com/crunchydata/crunchy-pgback-
rest:ubi8-2.45-2512
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - "ReadWriteOnce"
              resources:
                requests:
                  storage: 1Gi
```

In this case, **spec.postgresVersion** is set to Postgres major version 15. But how does PGO know which version 15 image to pull? PGO knows because its installer provides environment variables during the installation process. A typical installer will include configuration like this:

```
spec:
  containers:
    - name: operator
      image: postgres-operator
      env:
        - name: RELATED_IMAGE_POSTGRES_15
          value: "registry.developers.crunchydata.com/crunchydata/crunchy-post-
gres:ubi8-15.3-2512"
```

The `spec.postgresVersion` field declares a major version which PGO can satisfy by looking at `RELATED_IMAGE_POSTGRES_15`. So long as the version set by `spec.postgresVersion` has a corresponding related image, PGO will know what to do. If a required image has not been set, PGO's functionality will be limited and you can expect to see a `MissingRequiredImages` event.

All PGO installers come with related images for the supported images you can run in your cluster, but you aren't required to use them. Notice that `backups.pgbackrest` has an `image` field explicitly set. Setting the `image` field overrides the related image. To override the default for Crunchy Postgres, you would set `spec.image` to the specific container image you want, like this:

```
spec:
  image: registry.developers.crunchydata.com/crunchydata/crunchy-postgres:ubi8-15.3-2512
  postgresVersion: 15
```

In the above case, the `image` field is set to run Crunchy Postgres 15.3, built on the Red Hat 8 Universal Base Image.

The Postgres minor version, the 3 in 15.3, will increment when security patches and other improvements are added to Postgres. Crunchy Postgres for Kubernetes makes it easy to update your cluster to the latest minor version by giving you the latest supported images in each installer release. Just upgrade Crunchy Postgres for Kubernetes and you'll update to the latest supported minor versions of your cluster components.

Configuring Installers

While PGO installers ship with preset related image references, you can also customize those settings to point at images of your choosing. Related images can be customized for all installer types, including [Kustomize](#) (via `manager.yaml`), [Helm](#) (via `values.yaml`) and OperatorHub (via `spec.config.env` in the Subscription).

Configuring the Kustomize Installer's Related Images

To configure the image references in your Kustomize installer, look for `kustomize/install/manager/manager.yaml` to find the related images the operator's environment variables.

Configuring the Helm Installer's Related Images

To configure the image references in your Helm installer, look for `helm/install/values.yaml`.

Configuring the OperatorHub Installer's Related Images

After Crunchy Postgres for Kubernetes has been installed from OperatorHub, you can edit image references by clicking on Installed Operators and selecting Crunchy Postgres for Kubernetes. From there, select Subscription and from the Actions dropdown menu select Edit Subscription. Scroll to the spec section and you can create a config block to set environment variables like this:

```
spec:
  config:
    env:
      - name: RELATED_IMAGE_POSTGRES_15
```

```
value: 'registry.developers.crunchydata.com/crunchydata/crunchy-postgres:ubi8-15.3-2512'
```

By specifying a value for `RELATED_IMAGE_POSTGRES_15` in the above, we've overridden the value that comes from the OperatorHub installation package. After you've adjusted the Subscription to meet your needs, save it and observe that the environment variables in your PGO pod have updated.

Special Considerations for Upgrades on OperatorHub

Crunchy Postgres for Kubernetes (CPK) provides an OperatorHub experience with seamless updates for Crunchy Postgres minor versions. Automatic updates to minor Postgres versions are made possible through the list of related images packaged with the Crunchy Postgres for Kubernetes installer. Installing a new version of Crunchy Postgres for Kubernetes will trigger these updates.

Upgrading to a new major version of Crunchy Postgres is not automatic, but related images are still involved. When it's time to upgrade the Crunchy Postgres major version, PGO will run the image defined under `RELATED_IMAGE_PGUPGRADE` to do the work. The upgrade container holds binaries for different versions of Crunchy Postgres. Successful upgrades depend on the upgrade container holding a binary for the version of Crunchy Postgres you're presently running, as well as the version of Crunchy Postgres targeted in your upgrade. When you upgrade your installation of Crunchy Postgres for Kubernetes, the newer package will include the latest supported versions of Crunchy Postgres and will not include versions no longer supported.

Note that minor and major version upgrades are only possible **for as long as your major version of Crunchy Postgres is supported**. This makes it important to perform major upgrades in a timely fashion. If the latest upgrade image does not include your current major version of Crunchy Postgres, a Postgres upgrade might be difficult.

Logical Replication

[Logical replication](#) is a Postgres feature that provides a convenient way for moving data between databases, particularly Postgres clusters that are in an active state. To apply logical replication, we'll first enable the feature in our cluster, then we'll create a publication in one cluster and a subscription to that publication in another cluster. With this pub-sub relationship established, we'll observe data created in one cluster flowing into another.

Before getting started, you may want to create the `postgres-operator` namespace if you haven't already, `kubectl create ns postgres-operator`. Just as we did in the Quickstart and Tutorials, we're going to create a Postgres cluster named `hippo`. You may want to delete the existing `hippo` cluster, if you have one left over. Finally, you'll need a running installation of Crunchy Postgres for Kubernetes.

Enable Logical Replication

This example creates two separate Postgres clusters named `hippo` and `rhino`. We will logically replicate data from `rhino` to `hippo`. We can create these two Postgres clusters by creating a file called `replication-example.yaml` and pasting in the manifests below:

```
---
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
```

```

name: hippo
namespace: postgres-operator
spec:
  postgresVersion: 17
  instances:
    - dataVolumeClaimSpec:
        accessModes:
          - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - "ReadWriteOnce"
              resources:
                requests:
                  storage: 1Gi
  ---
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: rhino
  namespace: postgres-operator
spec:
  postgresVersion: 17
  instances:
    - dataVolumeClaimSpec:
        accessModes:
          - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - "ReadWriteOnce"
              resources:
                requests:
                  storage: 1Gi
  users:
    - name: logic
      databases:
        - zoo
      options: "REPLICATION"

```

The key difference between the two Postgres clusters is this section in the **rhino** manifest:

```

users:
  - name: logic
    databases:
      - zoo
    options: "REPLICATION"

```

This creates a database called **zoo** and a user named **logic** with **REPLICATION** privileges. This will allow for replicating data logically to the **hippo** Postgres cluster.

Create these two Postgres clusters with the command **kubectl apply -f replication-example.yaml**

Create a Publication

For convenience, you can use the **kubectl exec** method to log into the zoo database in **rhino**:

```
kubectl exec -it -n postgres-operator -c database $(kubectl get pods -n postgres-operator --selector='postgres-operator.crunchydata.com/cluster=rhino,postgres-operator.crunchydata.com/role=master' -o name) -- psql zoo
```

Let's create a simple table called **abc** that contains just integer data. We will also populate this table:

```
CREATE TABLE abc (id int PRIMARY KEY);
INSERT INTO abc SELECT * FROM generate_series(1,10);
```

We need to grant **SELECT** privileges to the **logic** user in order for it to perform an initial data synchronization during logical replication. You can do so with the following command:

```
GRANT SELECT ON abc TO logic;
```

Finally, create a [publication](#) that allows for the replication of data from **abc**:

```
CREATE PUBLICATION zoo FOR ALL TABLES;
```

Quit out of the **rhino** Postgres cluster with **\q**.

Create a Subscription

For the next step, you will need to get the connection information for how to connect as the **logic** user to the **rhino** Postgres database. You can get the key information from the following commands, which return the hostname, username, and password:

```
kubectl -n postgres-operator get secrets rhino-pguser-logic -o go-template='{{.data.host | base64decode}}'
kubectl -n postgres-operator get secrets rhino-pguser-logic -o go-template='{{.data.user | base64decode }}'
kubectl -n postgres-operator get secrets rhino-pguser-logic -o go-template='{{.data.password | base64decode }}'
```

The host will be something like **rhino-primary.postgres-operator.svc** and the user will be **logic**. Further down, the guide references the password as **\$LOGIC_PASSWORD**. You can substitute the actual password there.

Log into the **hippo** Postgres cluster. Note that we are logging into the **postgres** database within the **hippo** cluster:

```
kubectl exec -it -n postgres-operator -c database $(kubectl get pods -n postgres-operator --selector='postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/role=master' -o name) -- psql
```

Create a table called **abc** that is identical to the table in the **rhino** database:

```
CREATE TABLE abc (id int PRIMARY KEY);
```

Finally, create a [subscription](#) that will manage the data replication from **rhino** into **hippo**:

```
CREATE SUBSCRIPTION zoo
  CONNECTION 'host=rhino-primary.postgres-operator.svc user=logic dbname=zoo password=$LOG-
IC_PASSWORD'
  PUBLICATION zoo;
```

In a few moments, you should see the data replicated into your table:

```
TABLE abc;
```

which yields:

```
id
----
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
(10 rows)
```

You can further test that logical replication is working by modifying the data on **rhino** in the **abc** table, and then verifying that it is replicated into **hippo**.

Postgres Major Version Upgrade

You can perform a PostgreSQL major version upgrade declaratively using Crunchy Postgres for Kubernetes! The below guide will show you how you can upgrade Postgres to a newer major version. For minor updates, i.e. applying a bug fix release, you can follow the applying software updates guide in the tutorial.

Note that major version upgrades are **permanent**: you cannot roll back a major version upgrade through declarative management at this time. If this is an issue, we recommend keeping a copy of your Postgres cluster running your previous version of Postgres.

Warning

Please note the following prior to performing a PostgreSQL major version upgrade:

- If you used OperatorHub to install Crunchy Postgres for Kubernetes, you will not be able to complete a Postgres major version upgrade without first obtaining a registration token.

- Any Postgres cluster being upgraded must be in a healthy state in order for the upgrade to complete successfully. If the cluster is experiencing issues such as Pods that are not running properly, or any other similar problems, those issues must be addressed before proceeding.
- Major PostgreSQL version upgrades of PostGIS clusters are not currently supported.
- Major PostgreSQL version upgrades of Standby clusters are not currently supported. To upgrade an environment with a Standby cluster, delete the Standby before upgrading the Primary cluster to avoid archive mismatch errors. Then, rebuild the Standby with the new PostgreSQL version after the Primary cluster has been upgraded.

The following guide assumes that you have a running installation of Crunchy Postgres for Kubernetes as well as a running Postgres cluster with Postgres version 14 deployed. For tips on installation, see the Basic Setup Tutorial. To install Postgres 14, follow the steps in Create a Postgres Cluster, being sure to change `postgresVersion: 15` to `postgresVersion: 14`

Step 1: Take a Full Backup

Before starting your major upgrade, you should take a new full backup of your data. This adds another layer of protection in cases where the upgrade process does not complete as expected.

At this point, your running cluster is ready for the major upgrade.

Step 2: Configure the Upgrade Parameters through a PGUpgrade object

The next step is to create a `PGUpgrade` resource. This is the resource that tells the PGO-Upgrade controller which cluster to upgrade, what version to upgrade from, and what version to upgrade to. There are other optional fields to fill in as well, such as `Resources` and `Tolerations`; to learn more about these optional fields, check out the Upgrade CRD API.

For instance, if you have a Postgres cluster named `hippo` running PG 16 but want to upgrade it to PG 17, the corresponding `PGUpgrade` manifest would look like this:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PGUpgrade
metadata:
  name: hippo-upgrade
spec:
  postgresClusterName: hippo
  fromPostgresVersion: 16
  toPostgresVersion: 17
```

The `postgresClusterName` gives the name of the target Postgres cluster to upgrade and `toPostgresVersion` gives the version to update to. It may seem unnecessary to include the `fromPostgresVersion`, but that is one of the safety checks we have built into the upgrade process: in order to successfully upgrade a Postgres cluster, you have to know what version you mean to be upgrading from.

One very important thing to note: upgrade objects should be made in the same namespace as the Postgres cluster that you mean to upgrade. For security, the PGO-Upgrade controller does not allow for cross-namespace processes.

If you look at the status of the **PGUpgrade** object at this point, you should see a condition saying this:

```
type: "progressing",
status: "false",
reason: "PGClusterNotShutdown",
message: "PostgresCluster instances still running",
```

What that means is that the upgrade process is blocked because the cluster is not yet shutdown. We are stuck ("progressing" is false) until we shutdown the cluster. So let's go ahead and do that now.

Step 3: Shutdown and Annotate the Cluster

In order to kick off the upgrade process, you need to shutdown the cluster and add an annotation to the cluster signalling which PGUpgrade to run.

Why do we need to add an annotation to the cluster if the PGUpgrade already has the cluster's name? This is another security mechanism--think of it as a two-key nuclear system: the **PGUpgrade** has to know which Postgres cluster to upgrade; and the Postgres cluster has to allow this upgrade to work on it.

The annotation to add is **postgres-operator.crunchydata.com/allow-upgrade**, with the name of the **PGUpgrade** object as the value. So for our example above with a Postgres cluster named **hippo** and a **PGUpgrade** object named **hippo-upgrade**, we could annotate the cluster with the command

```
kubectl -n postgres-operator annotate postgrescluster hippo postgres-operator.crunchydata.com/allow-upgrade="hippo-upgrade"
```

To shutdown the cluster, edit the **spec.shutdown** field to true and reapply the spec with **kubectl**. For example, if you used the tutorial to create your Postgres cluster, you would run the following command:

```
kubectl -n postgres-operator apply -k customize/postgres
```

(Note: you could also change the annotation at the same time as you shutdown the cluster; the purpose of demonstrating how to annotate was primarily to show what the label would look like.)

Step 4: Watch and wait

When the last Postgres Pod is terminated, the PGO-Upgrade process will kick into action, upgrading the primary database and preparing the replicas. If you are watching the namespace, you will see the PGUpgrade controller start Pods for each of those actions. But you don't have to watch the namespace to keep track of the upgrade process.

To keep track of the process and see when it finishes, you can look at the **status.conditions** field of the **PGUpgrade** object. If the upgrade process encounters any blockers preventing it from finishing, the **status.conditions** field will report on those blockers. When it finishes upgrading the cluster, it will show the status conditions:

```
type: "Progressing"
status: "false"
reason: "PGUpgradeCompleted"

type: "Succeeded" status: "true"
reason: "PGUpgradeSucceeded"
```

You can also check the Postgres cluster itself to see when the upgrade has completed. When the upgrade is complete, the cluster will show the new version in its `status.postgresVersion` field.

If the process encounters any errors, the upgrade process will stop to prevent further data loss; and the `PGUpgrade` object will report the failure in its status. For more specifics about the failure, you can check the logs of the individual Pods that were doing the upgrade jobs.

Step 5: Restart your Postgres cluster with the new version

Once the upgrade process is complete, you can erase the `PGUpgrade` object, which will clean up any Jobs and Pods that were created during the upgrade. But as long as the process completed successfully, that `PGUpgrade` object will remain inert. If you find yourself needing to upgrade the cluster again, you will not be able to edit the existing `PGUpgrade` object with the new versions, but will have to create a new `PGUpgrade` object. Again, this is a safety mechanism to make sure that any `PGUpgrade` can only be run once.

Likewise, you may remove the annotation on the Postgres cluster as part of the cleanup. While not necessary, it is recommended to leave your cluster without unnecessary annotations.

To restart your newly upgraded Postgres cluster, you will have to update the `spec.postgresVersion` to the new version. You may also have to update the `spec.image` value to reflect the image you plan to use if that field is already filled in. Turn `spec.shutdown` to false, and PGO will restart your cluster:

```
spec:
  shutdown: false
  postgresVersion: 17
```

Warning

Setting and applying the `postgresVersion` or `image` values before the upgrade will result in the upgrade process being rejected.

Step 6: Complete the Post-Upgrade Tasks

After the upgrade Job has completed, there will be some amount of post-upgrade processing that needs to be done. During the upgrade process, the upgrade Job, via `pg_upgrade`, will issue warnings and possibly create scripts to perform post-upgrade tasks. You can see the full output of the upgrade Job by running a command similar to this:

```
kubectl -n postgres-operator logs hippo-pgupgrade-abcd
```

While the scripts are placed on the Postgres data PVC, you may not have access to them. The below information describes what each script does and how you can execute them.

In Postgres 13 and older, `pg_upgrade` creates a script called `analyze_new_cluster.sh` to perform a post-upgrade analyze using `vacuumdb` on the database.

The script provides two ways of doing so:

```
vacuumdb --all --analyze-in-stages
```

or

```
vacuumdb --all --analyze-only
```

Note that these commands need to be run as a Postgres superuser (e.g. **postgres**). For more information on the difference between the options, please see the documentation for [vacuumdb](#).

If you are unable to exec into the Pod, you can run **ANALYZE** directly on each of your databases.

pg_upgrade may also create a script called **delete_old_cluster.sh**, which contains the equivalent of

```
rm -rf '/pgdata/pg16'
```

When you are satisfied with the upgrade, you can execute this command to remove the old data directory. Do so at your discretion.

Note that the **delete_old_cluster.sh** script does not delete the old WAL files. These are typically found in **/pgdata-ta/pg16_wal**, although they can be stored elsewhere. If you would like to delete these files, this must be done manually.

If you have extensions installed you may need to upgrade those as well. For example, for the **pgaudit** extension we recommend running the following to upgrade:

```
DROP EXTENSION pgaudit;  
CREATE EXTENSION pgaudit;
```

pg_upgrade may also create a file called **update_extensions.sql** to facilitate extension upgrades. Be aware some of the recommended ways to upgrade may be outdated.

Please carefully review the **update_extensions.sql** file before you run it, and if you want to upgrade **pgaudit** via this file, update the file with the above commands for **pgaudit** prior to execution. We recommend verifying all extension updates from this file with the appropriate extension documentation and their recommendation for upgrading the extension prior to execution. After you update the file, you can execute this script using **kubectl exec**.

```
kubectl -n postgres-operator exec -it -c database $(kubectl -n postgres-operator get pods --selector='postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/role=master' -o name) -- psql -f /pgdata/update_extensions.sql
```

If you cannot exec into your Pod, you can also manually run these commands as a Postgres superuser.

Ensure the execution of this and any other SQL scripts completes successfully, otherwise your data may be unavailable.

Once this is done, your major upgrade is complete! Enjoy using your newer version of Postgres!

Large Clusters

The **PGUpgrade** resource runs **pg_upgrade** optimally for any size cluster, but you can also override its choices.

It uses multiple CPU cores when **spec.resources.*.cpu** is three or greater. You can change this in **spec.jobs**.

If you have a specialized storage provider, the **--clone** or **--copy-file-range** option may be beneficial. You can change this in **spec.transferMethod**.

Migrate Data Volumes to New Clusters

There are certain cases where you may want to migrate existing volumes to a new cluster. If so, read on for an in depth look at the steps required.

Prerequisites

While your existing Postgres instance is still running, confirm that the following 3 conditions hold:

- Your volume has its `persistentVolumeReclaimPolicy` set to `Retain`.
- The [postgres superuser](#) exists in your Postgres instance.
- Your volume's data directory is owned by the operating system's `postgres` user, with user ID 26.

Warning

If your PVC's reclaim policy isn't set to `Retain`, **your data will be lost**. If you don't have a postgres database user, or if the data directory isn't owned by a `postgres` operating system user, the bootstrap process will fail.

Once all three of these conditions have been met, consider performing a test run to familiarize yourself with the process and identify pain points unique to your system and configuration.

Configure your PostgresCluster

In order to use existing pgData, pg_wal or pgBackRest repo volumes in a new PostgresCluster, you will need to configure the `spec.dataSource.volumes` section of your PostgresCluster manifest. As shown below, there are three possible volumes you may configure: `pgDataVolume`, `pgWALVolume` and `pgBackRestVolume`. Under each, you must define the PVC name to use in the new cluster. A directory may also be defined, as needed, for cases where the existing directory name does not match the v5 directory.

To help explain how these fields are used, we will consider a `pgcluster` named "oldhippo" from PGO v4. We will assume that the `pgcluster` has been deleted and only the PVCs have been left in place.

Info

Any differences in configuration or other datasources will alter this procedure significantly. Certain storage options require additional steps (see [Considerations](#)).

In a standard PGO v4.7 cluster, a primary database pod with a separate pg_wal PVC will mount its pgData PVC, named "oldhippo", at `/pgdata` and its pg_wal PVC, named "oldhippo-wal", at `/pgwal` within the pod's file system. In this pod, the standard pgData directory will be `/pgdata/oldhippo` and the standard pg_wal directory will be `/pgwal/oldhippo-wal`. The pgBackRest repo pod will mount its PVC at `/backrestrepo` and the repo directory will be `/backrestrepo/oldhippo-backrest-shared-repo`.

With the above in mind, we need to reference the three PVCs we wish to migrate in the `dataSource.volumes` portion of the `PostgresCluster` spec. Additionally, to accommodate the PGO v5 file structure, we must also reference the `pgData` and `pgBackRest` repo directories. Note that the `pg_wal` directory does not need to be moved when migrating from v4 to v5!

Now, we just need to populate our CRD with the information described above:

```
spec:
  dataSource:
    volumes:
      pgDataVolume:
        pvcName: oldhippo
        directory: oldhippo
      pgWALVolume:
        pvcName: oldhippo-wal
      pgBackRestVolume:
        pvcName: oldhippo-pgbr-repo
        directory: oldhippo-backrest-shared-repo
```

To understand how to set `pgDataVolume.directory`, think of subtracting the mount path of your volume from the `PGDATA` path.

If your volume is mounted at `"/data"`, and `PGDATA` is set to `"/data/pg15/oldhippo"`, you'll set `pgDataVolume.directory` to `"pg15/oldhippo"`.

Lastly, it is very important that the PostgreSQL version and storage configuration in your `PostgresCluster` match *exactly* the existing volumes being used.

If the volumes were used with PostgreSQL 13, the `spec.postgresVersion` value should be `13` and the associated `spec.image` value should refer to a PostgreSQL 13 image.

Similarly, the configured data volume definitions in your `PostgresCluster` spec should match your existing volumes. For example, if the existing `pgData` PVC has a `RWO` access mode and is 1 Gigabyte, the relevant `dataVolumeClaimSpec` should be configured as

```
dataVolumeClaimSpec:
  accessModes:
    - "ReadWriteOnce"
  resources:
    requests:
      storage: 1G
```

With the above configuration in place, your existing PVC will be used when creating your `PostgresCluster`. They will be given appropriate Labels and ownership references, and the necessary directory updates will be made so that your cluster is able to find the existing directories.

Considerations

Removing PGO v4 labels

When migrating data volumes from v4 to v5, PGO relabels all volumes for PGO v5, but **will not remove existing PGO v4 labels**. This results in PVCs that are labeled for both PGO v4 and v5, which can lead to unintended behavior.

To avoid that, you must manually remove the `pg-cluster` and `vendor` labels, which you can do with a `kubectl` command. For instance, given a cluster named `hippo` with a dedicated pgBackRest repo, the PVC will be `hippo-pgbr-repo`, and the PGO v4 labels can be removed with the below command:

```
kubectl label pvc hippo-pgbr-repo pg-cluster- vendor-
```

Proper file permissions for certain storage options

Additional steps are required to set proper file permissions when using certain storage options, such as NFS and HostPath storage due to a known issue with how fsGroups are applied.

When migrating from PGO v4, this will require the user to manually set the group value of the pgBackRest repo directory, and all subdirectories, to `26` to match the `postgres` group used in PGO v5. Please see [this example](#) for more information.

Additional Considerations

- An existing `pg_wal` volume is not required when the `pg_wal` directory is located on the same PVC as the `pgData` directory.
- When using existing `pg_wal` volumes, an existing `pgData` volume **must** also be defined to ensure consistent naming and proper bootstrapping.
- When migrating from PGO v4 volumes, it is recommended to use the most recently available version of PGO v4.
- As there are many factors that may impact this procedure, it is strongly recommended that a test run be completed beforehand to ensure successful operation.

Putting it all together

Now that we've identified all of our volumes and required directories, we're ready to create our new cluster!

Below is a complete `PostgresCluster` that includes everything we've talked about. After your `PostgresCluster` is created, you should remove the `spec.dataSource.volumes` section.

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: oldhippo
spec:
  postgresVersion: 17
  dataSource:
    volumes:
      pgDataVolume:
        pvcName: oldhippo
        directory: oldhippo
      pgWALVolume:
        pvcName: oldhippo-wal
      pgBackRestVolume:
        pvcName: oldhippo-pgbr-repo
        directory: oldhippo-backrest-shared-repo
  instances:
    - name: instance1
      dataVolumeClaimSpec:
        accessModes:
          - "ReadWriteOnce"
```

```

resources:
  requests:
    storage: 1G
walVolumeClaimSpec:
  accessModes:
  - "ReadWriteOnce"
  resources:
    requests:
      storage: 1G
backups:
  pgbackrest:
    repos:
    - name: repo1
      volume:
        volumeClaimSpec:
          accessModes:
          - "ReadWriteOnce"
          resources:
            requests:
              storage: 1G

```

Exporter Configuration

The Crunchy Postgres for Kubernetes Monitoring stack relies on either the Crunchy Postgres Exporter sidecar or the OpenTelemetry Collector sidecar to collect real-time metrics about a PostgreSQL database.

In this guide, we cover how to configure the Crunchy Postgres Exporter to use a custom password, tls encryption, and custom queries to fit your needs.

Setting a custom ccp_monitoring password

The `postgres_exporter` process will use the `ccp_monitoring` username and password to gather metrics from Postgres. Considering these credentials are only used within a cluster, they can normally be generated by Crunchy Postgres for Kubernetes without user intervention. There are some cases, like standby monitoring, where a user might need to manually configure the `ccp_monitoring` password.

To update the `ccp_monitoring` password for a PostgresCluster, you will need to edit the `$CLUSTER_NAME-monitoring` secret. The following command will open up an editor with the contents of the monitoring secret:

```
kubectl edit secret $CLUSTER_NAME-monitoring
```

The editor will look something like this:

```

apiVersion: v1
kind: Secret
metadata:
  name: $CLUSTER_NAME-monitoring
  labels:
    postgres-operator.crunchydata.com/cluster: $CLUSTER_NAME
    postgres-operator.crunchydata.com/role: monitoring
data:
  password: cGFzc3dvcmQ=
  verifier: $sha

```

To set a password you can remove the entire **data** section (including both the **password** and **verifier** fields) and replace it with the **stringData** field:

```
stringData:
  password: $NEW_PASSWORD
```

Note: The **stringData** field is a Kubernetes feature that allows you to provide a plain-text field to a secret that is then encoded like the **data** field. This field is describe in the [Kubernetes documentation](#).

By saving this change, the secret will be updated and the change will make its way into the pod. The new secret files will be updated in the file system and the **postgres_exporter** process will be restarted, which may take a minute or two. Once the process has restarted, the **postgres_exporter** will query the database using the updated password.

Configuring TLS Encryption for the Exporter

Crunchy Postgres for Kubernetes allows you to configure the exporter sidecar to use TLS encryption. If you provide a custom TLS Secret via the exporter spec:

```
monitoring:
  pgmonitor:
    exporter:
      customTLSSecret:
        name: hippo.tls
```

Like other custom TLS Secrets that can be configured with Crunchy Postgres for Kubernetes, the Secret will need to be created in the same Namespace as your PostgresCluster. It should also contain the TLS key (**tls.key**) and TLS certificate (**tls.crt**) needed to enable encryption.

```
data:
  tls.crt: $VALUE
  tls.key: $VALUE
```

After you configure TLS for the exporter, you will need to update your Prometheus deployment to use TLS, and your connection to the exporter will be encrypted. Check out the [Prometheus](#) documentation for more information on configuring TLS for [Prometheus](#).

Custom Queries for the Exporter

Out of the box, the exporter is set up with default queries that will provide you with valuable information about your PostgresClusters. However, sometimes, you want to provide your own custom queries to retrieve metrics not in the defaults. Luckily, Crunchy Postgres for Kubernetes has you covered.

The first thing you will need to figure out when implementing your own custom queries is whether you want to completely swap out the default queries or add your queries to the defaults that Crunchy Data provides.

Using Your Own Custom Set

If you wish to completely swap out the Crunchy-provided default queries with your own set, you will need to start by putting all of the queries that you wish to run in a YAML file named **queries.yml**. You can use the query files found in the

[pgMonitor repo](#) as guidance for the proper format. This file should then be placed in a ConfigMap. For example, we could run the following command:

```
kubectl create configmap my-custom-queries --from-file=path/to/file/queries.yml -n postgres-operator
```

This will create a ConfigMap named **my-custom-queries** in the **postgres-operator** namespace, and it will hold the **queries.yml** file found at the relative path of **path/to/file**.

Once the ConfigMap is created, you simply need to tell Crunchy Postgres for Kubernetes the name of the ConfigMap by editing your PostgresCluster Spec:

```
monitoring:
  pgmonitor:
    exporter:
      configuration:
        - configMap:
            name: my-custom-queries
```

Once the spec is applied, the exporter will be restarted and your new metrics will be available. If you later make a change to the custom queries in the ConfigMap, the exporter process will again be restarted and the new queries used once a difference is detected in the ConfigMap.

Append Your Custom Queries to the Defaults

Starting with Postgres Operator 5.5, you can easily append custom queries to the Crunchy Data defaults! To do this, the setup has the same three easy steps that we just went through:

- Put your desired queries in a YAML file named **queries.yml**.
- Create a ConfigMap that holds the **queries.yml** file.
- Tell Crunchy Postgres for Kubernetes the name of your ConfigMap using the **monitoring.pgmonitor.exporter.configuration** spec.

The additional step that tells Crunchy Postgres for Kubernetes to append the queries rather than swapping them out is to turn on the **AppendCustomQueries** feature gate.

Crunchy Postgres for Kubernetes feature gates are enabled by setting the **PGO_FEATURE_GATES** environment variable on the Crunchy Postgres for Kubernetes Deployment. To enable the appending of the custom queries, you would want to set:

```
PGO_FEATURE_GATES="AppendCustomQueries=true"
```

Please note that it is possible to enable more than one feature at a time as this variable accepts a comma delimited list. For example, to enable multiple features, you would set **PGO_FEATURE_GATES** like so:

```
PGO_FEATURE_GATES="FeatureName=true, FeatureName2=true, FeatureName3=true..."
```

OpenTelemetry Logging

For an overview of the full observability architecture within CPK, including details about the architecture for OpenTelemetry logging, please see the Database Observability Architecture page.

This section will provide steps for enabling OpenTelemetry logging, along with examples for configuring your PostgresClusters to send logging data to a variety of different OpenTelemetry-compatible services and backends.

Enabling OpenTelemetry Logging

In order to use OpenTelemetry logging, the **OpenTelemetryLogs** feature gate must first be enabled in your CPK installation.

Please see the Feature Gate Installation Guide for guidance on how to properly enable this feature gate within your installation.

Once the feature gate is enabled, you will be able to create PostgresClusters and PGAdmins with OpenTelemetry logging. To do that, add an **instrumentation** block to your PostgresCluster or PGAdmin spec, like so, for a default-only configuration:

```
spec:
  instrumentation: {}
```

Once applied, you will see OpenTelemetry collector sidecars deployed alongside the various components comprising your PostgresCluster and/or PGAdmin. Additionally, CPK will automatically configure the various components within your PostgresCluster and/or PGAdmin for file-base logging.

Configuration Defaults for OpenTelemetry Logging

When OpenTelemetry logging is enabled, CPK ensures certain logging configurations are set and changes some of the default behavior for the associated components. For instance, to process logs correctly, each component is configured to log to files.

Postgres

When **OpenTelemetryLogs** is enabled for a PostgresCluster, the following configurations are set by CPK:

```
logging_collector = 'on'
log_directory = '/pgdata/logs/postgres'
log_destination = 'jsonlog' # Set for Postgres 16 and higher, but set to 'csvlog' for 15 and 14
log_rotation_size = '0'
log_truncate_on_rotation = 'on'
log_timezone = 'UTC'
```

CPK sets those parameters and will override any user attempt to set them.

NOTE: By turning on OpenTelemetry logging, the location of the postgres logs will move from **/pgdata/pg##/log** to **/pgdata/logs/postgres**.

pgBackRest

In order to parse the logs correctly, the OpenTelemetry collector expects the default timestamp format for pgBackRest. To make sure that the timestamp is in the correct form, you shouldn't disable or adjust the timestamp by setting the `no-log-timestamp` or `log-timestamp=n` configurations.

Note: When pgBackRest logging is turned up to `debug` or higher, Kubernetes may [rotate the files](#) storing OTel collector console output, obscuring it from `kubectl logs`. This may result in `kubectl logs` returning a blank result from the `collector` container until another pgBackRest process (such as a manual backup) is triggered.

pgBouncer

When `OpenTelemetryLogs` is enabled for a PostgresCluster with pgBouncer, CPK sets the logfile:

```
logfile = "/tmp/pgbouncer.log"
```

Note that pgBouncer logs to a file in the ephemeral `/tmp` directory, so any restart of the pgBouncer pod will wipe out previous logs.

pgAdmin4

When `OpenTelemetryLogs` is enabled for a pgAdmin custom resource with a `spec.instrumentation` block, CPK makes the following adjustments to the pgAdmin configuration:

```
DATA_DIR = '/var/lib/pgadmin'
LOG_FILE = '/var/lib/pgadmin/logs/pgadmin.log'
JSON_LOGGER = True
CONSOLE_LOG_LEVEL = logging.WARNING
FILE_LOG_LEVEL = logging.INFO
FILE_LOG_FORMAT_JSON = {'time': 'created', 'name': 'name', 'level': 'levelname', 'message': 'message'}
```

CPK also makes changes to the Gunicorn settings to ensure that it logs to file at `/var/lib/pgadmin/logs/gunicorn.log`, and, like pgAdmin above, logs in json format.

NOTE: Only pgAdmins that are deployed using the PGAdmin custom resource can use the OpenTelemetry features. Logs from pgAdmins that are deployed via the PostgresCluster's `spec.userInterface` configuration will not be collected.

Log Rotation & Retention

When OpenTelemetry logging is enabled for a PostgresCluster or PGAdmin, CPK will ensure that the components log to file. To ensure that these log files don't become unmanageable, CPK also manages log rotation and retention where possible.

To configure log retention for your PostgresCluster or PGAdmin, fill in the `spec.instrumentation.logs.retention-Period` field on your spec:

```
spec:
  instrumentation:
    logs:
      retentionPeriod: 2d
```

This `retentionPeriod` field can be an RFC 3339 duration or a number and unit; the minimum unit is an hour, and the maximum unit is a week.

The different components of a PostgresCluster or PGAdmin manage their rotation differently, so this setting is approximate. CPK will always retain *at least* the specified amount, but sometimes more will be retained.

NOTE: Patroni logs are not rotated by age, but by size. This can be set independently in the `spec.patroni.logging.storageLimit` field. If that field is left blank, CPK will default to 25M, which is the minimum limit for Patroni log storage. See our [guide to customizing Postgres instance logs](#) for more detail on this field.

Configuring Exporters

When you first turn on OpenTelemetry logging in CPK with no additional configuration, the logs that are collected are sent to the [Debug Exporter](#), which outputs the logs to the console. Since the collector is running in a sidecar container in a Kubernetes Pod, that console output is added to the container logs which you can retrieve with the `kubectl logs` command. If you were running a logging-enabled PostgresCluster named `hippo` in the `postgres-operator` namespace and wanted to see your `postgres`, `patroni`, and `pgbackrest` logs from the primary instance pod, the commands to retrieve those logs would look like this:

```
PG_CLUSTER_PRIMARY_POD=$(kubectl get pod -n postgres-operator -o name -l postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/role=master)
kubectl -n postgres-operator logs "${PG_CLUSTER_PRIMARY_POD}" -c collector
```

However, this output is not the easiest to read and is not well organized or easily filtered or searched. You will therefore almost certainly want to export your logs to a dedicated backend or service of some kind where you can more easily search and read through your logs.

Luckily, the [OpenTelemetry Collector](#) that we use has a plethora of [exporters](#) built into it that should satisfy most needs.

To use an exporter, you define it in the `instrumentation.config.exporters` section. Fields in this section should follow the `type[/name]` ["component identifier"](#) format where the `type` is the exporter you want to use and `name` is the optional name you want to give this configuration. The optional name allows you to define multiple configurations of the same exporter type. For example, you could have two configurations of the `otlp` exporter where one is called `otlp` and the second is called `otlp/2`. The value for each field is the configuration for that exporter. For example:

```
spec:
  instrumentation:
    config:
      exporters:
        otlp:
          endpoint: a-standalone-collector:4317
          tls:
            insecure: true
```

```
otlp/2:
  endpoint: another-collector:4317
```

The configuration you define will differ depending on the exporter you are using. Please follow the documentation for your chosen exporter to determine what configuration to provide. OpenTelemetry keeps a [list of exporters](#) that are specific to the "contrib" collector, along with their documentation. There are also [exporters that have their documentation in the base collector repo](#), but are also available in the "contrib" collector that we use.

Once the exporter is configured, you lastly need to tell the collector to use the exporter in the logging pipeline by adding the name of the exporter to the `instrumentation.logs.exporters` array. For example:

```
spec:
  instrumentation:
    config:
      exporters:
        otlp/1:
          endpoint: a-standalone-collector:4317
          tls:
            insecure: true
      logs:
        exporters: ['otlp/1']
```

You will find more examples of exporter configurations for commonly used logging backends in the [Example Exporter Configurations](#) section below.

Files

Some exporters might require that configuration be provided via files, such as separate config files, certificates, etc. This can be done via the `instrumentation.config.files` section, which allows you to project files that are in Kubernetes Secrets, ConfigMaps, etc., into the volume that is mounted into the collector container. For example, creating a Secret with the following command:

```
kubectl -n postgres-operator create secret tls some-otel-exporter-certs --cert=server.crt --key=server.key
```

And then adding the following to your instrumentation spec:

```
spec:
  instrumentation:
    config:
      files:
        - secret:
            name: some-otel-exporter-certs
```

Will result in the `server.crt` and `server.key` files being mounted in the `/etc/otel-collector` directory of the collector container.

Batch Size

In between the collection of logs via the receiver components and the exporting of the logs via the exporter components, OpenTelemetry allows for transformation of the data via "processor" components. One of the processors that we use in

our implementation is the [Batch Processor](#), which compresses the data and reduces the number of network connections needed to export the data.

The size of the batches and how often they are sent is determined by three different settings:

- `maxDelay` - The maximum time to wait before exporting a batch, regardless of the batch's size. Higher numbers allow more records to be deduplicated and compressed before export.
- `maxRecords` - The maximum number of records to include in an exported batch. When present, batches this size are sent without any further delay.
- `minRecords` - The number of records to wait for before exporting a batch. Higher numbers allow more records to be deduplicated and compressed before export.

By default, `maxRecords` is not set and the other two settings are configured as such:

- `maxDelay` = 200ms
- `minRecords` = 8192

You can configure these settings to your liking via the `spec.instrumentation.logs.batches` section. For example:

```
spec:
  instrumentation:
    logs:
      batches:
        maxDelay: 1s
        maxRecords: 16384
        minRecords: 8192
```

If you wish to turn batching off entirely, you must set both `maxDelay` and `minRecords` to zero:

```
spec:
  instrumentation:
    logs:
      batches:
        maxDelay: 0s
        minRecords: 0
```

Resource Detection

Another processor that we incorporate into our logs pipelines is the [Resource Detection Processor](#), which can detect resource information from the host and add it as metadata to log records. The full list of supported detectors can be found in the [processor's documentation](#).

You can configure one or more detectors via the `spec.instrumentation.config.detectors` array, where each entry has a `name` field that indicates which detector to use, and an optional `attributes` field, where you can specify particular attributes that you wish to turn on or off. For example, if you were running CPK in an Azure Kubernetes Service cluster, you might configure this section like so:

```
spec:
  instrumentation:
    config:
      detectors:
        - name: aks
```

```
attributes:
  k8s.cluster.name: true
```

See the [Resource Detection Processor documentation](#) for more details on the various detectors and their particular attributes.

Example Exporter Configurations

This section provides example configurations for a variety of different OpenTelemetry-compatible logging services and backends.

Google Cloud

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: otel-hippo
  namespace: postgres-operator
spec:
  instrumentation:
    config:
      detectors:
        - name: gcp
      exporters:
        # https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/exporter/googlecloudexporter#configuration-reference
        googlecloud:
          log:
            default_log_name: "collector-exported-log"
            resource_filters:
              - prefix: "k8s"
              - prefix: "db"
  logs:
    exporters: ['googlecloud']
```

OTLP

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: otel-hippo
  namespace: postgres-operator
spec:
  instrumentation:
    config:
      exporters:
        # https://github.com/open-telemetry/opentelemetry-collector/tree/main/exporter/otlpexporter#getting-started
        otlp: # for exporting to another collector
          endpoint: "otel-collector:4317"
          tls:
            insecure: true
  logs:
    exporters: ['otlp']
```

OpenTelemetry Metrics

For an overview of the full observability architecture within CPK, including details about the architecture for OpenTelemetry metrics, please see the Database Observability Architecture page.

This section will provide steps for enabling OpenTelemetry metrics.

Enabling OpenTelemetry Metrics

In order to use OpenTelemetry metrics, the **OpenTelemetryMetrics** feature gate must first be enabled in your CPK installation.

Please see the Feature Gate Installation Guide for guidance on how to properly enable this feature gate within your installation.

Once the feature gate is enabled, you will be able to create PostgresClusters with OpenTelemetry metrics. To do that, add an **instrumentation** block to your PostgresCluster spec, like so for a default-only configuration:

```
spec:
  instrumentation: {}
```

You will see OpenTelemetry collector sidecars deployed alongside the various components comprising your PostgresCluster that are enabled to export metrics: currently Postgres and pgBouncer.

By default, CPK will setup a Prometheus exporter that will expose these metrics on port **9187** at the **/metrics** endpoint. If you are using the CPK Monitoring installer, then you are good to go as it is already set up to scrape this endpoint for metrics. If you are using a custom metric aggregation method, just point that towards the Postgres and/or pgBouncer pods at the **9187/metrics** endpoint.

Customizing Postgres Queries

While the metrics that are provided out of the box will cover most users' Postgres observability needs, some users will want to add their own metrics and/or remove some of the built-in metrics. Luckily, we have made both customizations very easy.

Adding Custom Metrics

Since we use the [OTel SqlQueryReceiver](#) to run our queries and collect the metrics, in order to add your own custom metrics you will first need to put them in a YAML file in a format that the SqlQueryReceiver accepts. This should be a list or array of entries that each have an **sql** field that holds the SQL statement that will run on the database and a **metrics** section that holds a list of the different metrics that are associated with that SQL statement. Each metric entry must have:

- a **metric_name**, which holds the desired name of the metric
- a **value_column**, which holds the column name in the returned dataset that is used to set the value of the metric's datapoint

As an example of formatting, see the following query and its respective metrics from our built-in queries:

```
sql: >
  SELECT datname AS dbname
```

```

    , checksum_failures AS count
    , coalesce(extract(epoch from (clock_timestamp() - checksum_last_failure)), 0) AS time_since_last_failure_seconds
FROM pg_catalog.pg_stat_database
WHERE pg_stat_database.datname IS NOT NULL;
metrics:
- metric_name: ccp_data_checksum_failure_count
  value_column: count
  attribute_columns: ["dbname"]
  description: Total number of checksum failures on this database
  static_attributes:
    server: "localhost:5432"
- metric_name: ccp_data_checksum_failure_time_since_last_failure_seconds
  value_column: time_since_last_failure_seconds
  value_type: double
  attribute_columns: ["dbname"]
  description: Time interval in seconds since the last checksum failure was encountered
  static_attributes:
    server: "localhost:5432"

```

As you can see there are other optional fields that can be set on each metric. See the [SqlQueryReceiver](#) documentation for details on all of the various settings.

Once you have your custom queries and metrics in an SqlQueryReceiver compliant yaml file, you will want to put it in a ConfigMap. If we had two queries files, one that contains queries we plan to run very frequently and one that contains queries we plan to run less frequently, we might create our ConfigMap using a command similar to the below:

```

kubectl create configmap my-custom-queries --from-file=path/to/file/my-slow-custom-queries.yaml --from-file=path/to/file/my-fast-custom-queries.yaml -n postgres-operator

```

Once the ConfigMap is created, you need to edit your spec to tell the operator to use the new queries. There is an **instrumentation.metrics.customQueries.add** section which can hold a list of entries, where each entry has a **name**, a **queries** section where you specify the ConfigMap and the file within, and the optional **collectionInterval** which tells the receiver how often to run the queries. For example:

```

spec:
  instrumentation:
    metrics:
      customQueries:
        add:
          - name: slow-custom-queries
            queries:
              name: my-custom-queries
              key: my-slow-custom-queries.yaml
              collectionInterval: 300s
          - name: 2fast2furious
            queries:
              name: my-custom-queries
              key: my-fast-custom-queries.yaml

```

In this example, we are adding two sets of queries, both of which come from files in our **my-custom-queries** ConfigMap. The first set uses the **my-slow-custom-queries.yaml** file and is named **slow-custom-queries**. This set of queries will be run every 5 minutes. The second set uses the **my-fast-custom-queries.yaml** file and is named **2fast2furious**. It doesn't have a **collectionInterval** set, so it will use the default setting, which is 5 seconds.

Removing Built-in Metrics

Removing default metrics is even more straight forward; simply add the names of the metrics that you no longer want to the `spec.instrumentation.metrics.customQueries.remove` list. For example:

```
spec:
  instrumentation:
    metrics:
      customQueries:
        remove:
          - ccp_pg_stat_activity_count
          - ccp_stat_user_tables_autoanalyze_count
```

If all metrics for a given SQL query are removed, the SQL query will no longer be run.

Storage Retention

PGO uses [persistent volumes](#) to store Postgres data and, based on your configuration, data for backups, archives, etc. There are cases where you may want to retain your volumes for later use.

The below guide shows how to configure your persistent volumes (PVs) to remain after a Postgres cluster managed by PGO is deleted and to deploy the retained PVs to a new Postgres cluster.

For the purposes of this exercise, we will use a Postgres cluster named `hippo`.

Modify Persistent Volume Retention

Retention of persistent volumes is set using a [reclaim policy](#). By default, more persistent volumes have a policy of `Delete`, which removes any data on a persistent volume once there are no more persistent volume claims (PVCs) associated with it.

To retain a persistent volume you will need to set the reclaim policy to `Retain`. Note that persistent volumes are cluster-wide objects, so you will need the appropriate permissions to be able to modify a persistent volume.

To retain the persistent volume associated with your Postgres database, you must first determine which persistent volume is associated with the persistent volume claim for your database. First, locate the persistent volume claim. For example, with the `hippo` cluster that you would have created in the Quickstart, you can do so with the following command:

```
kubectl get pvc -n postgres-operator --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/data=postgres
```

This will yield something similar to the below, which are the PVCs associated with any Postgres instance:

NAME	STATUS	VOLUME		CAPACITY	ACCESS MODES	STORAGECLASS
hippo-instance1-dard	x9vq-6m53s	pgdata	Bound	pvc-aef7ee64-4495-4813-b896-8a67edc53e58	1Gi	R

The `VOLUME` column contains the name of the persistent volume. You can inspect it using `kubectl get pv`.g.:

```
kubectl get pv pvc-aef7ee64-4495-4813-b896-8a67edc53e58
```

which should yield:

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM		
pvc-aef7ee64-4495-4813-b896-8a67edc53e58	1Gi			RWO		Delete	Bound
stance1-x9vq-pgdata		standard		8m10s			

To set the reclaim policy to **Retain**, you can run a command similar to this:

Bash:

```
kubectl patch pv pvc-aef7ee64-4495-4813-b896-8a67edc53e58 -p '{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'
```

Powershell:

```
kubectl patch pv pvc-aef7ee64-4495-4813-b896-8a67edc53e58 -p '{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'
```

Verify that the change occurred:

```
kubectl get pv pvc-aef7ee64-4495-4813-b896-8a67edc53e58
```

should show that **Retain** is set in the **RECLAIM POLICY** column:

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM		
pvc-aef7ee64-4495-4813-b896-8a67edc53e58	1Gi			RWO		Retain	Bound
stance1-x9vq-pgdata		standard		9m53s			

Delete Postgres Cluster, Retain Volume

Warning

This is a potentially destructive action. Please be sure that your volume retention is set correctly and/or you have backups in place to restore your data.

Delete your Postgres cluster. You can delete it using the manifest or with a command similar to:

```
kubectl -n postgres-operator delete postgrescluster hippo
```

Wait for the Postgres cluster to finish deleting. You should then verify that the persistent volume is still there:

```
kubectl get pv pvc-aef7ee64-4495-4813-b896-8a67edc53e58
```

should yield:

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM		
pvc-aef7ee64-4495-4813-b896-8a67edc53e58	1Gi			RWO		Retain	Released
stance1-x9vq-pgdata		standard		21m			

Create Postgres Cluster With Retained Volume

You can now create a new Postgres cluster with the retained volume. First, to aid the process, you will want to provide a label that is unique for your persistent volumes so we can identify it in the manifest. For example:

```
kubectl label pv pvc-aef7ee64-4495-4813-b896-8a67edc53e58 pgo-postgres-cluster=postgres-operator-hippo
```

(This label uses the format `$NAMESPACE-<clusterName>`).

Next, you will need to reference this persistent volume in your Postgres cluster manifest. For example:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  postgresVersion: 17
  instances:
    - name: instance1
      dataVolumeClaimSpec:
        accessModes:
          - 'ReadWriteOnce'
        resources:
          requests:
            storage: 1Gi
        selector:
          matchLabels:
            pgo-postgres-cluster: postgres-operator-hippo
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - 'ReadWriteOnce'
              resources:
                requests:
                  storage: 1Gi
```

Wait for the Pods to come up. You may see the Postgres Pod is in a **Pending** state. You will need to go in and clear the claim on the persistent volume that you want to use for this Postgres cluster, e.g.:

Bash:

```
kubectl patch pv pvc-aef7ee64-4495-4813-b896-8a67edc53e58 -p '{"spec":{"claimRef": null}}'
```

Powershell:

```
kubectl patch pv pvc-aef7ee64-4495-4813-b896-8a67edc53e58 -p '{"spec":{"claimRef": null}}'
```

After that, your Postgres cluster will come up and will be using the previously used persistent volume!

If you ultimately want the volume to be deleted, you will need to revert the reclaim policy to **Delete**, e.g.:

Bash:

```
kubectl patch pv pvc-aef7ee64-4495-4813-b896-8a67edc53e58 -p '{"spec":{"persistentVolumeReclaimPolicy":"Delete"}}'
```

Powershell:

```
kubectl patch pv pvc-aef7ee64-4495-4813-b896-8a67edc53e58 -p '{"spec":{"persistentVolumeReclaimPolicy":"Delete"}}'
```

After doing that, the next time you delete your Postgres cluster, the volume and your data will be deleted.

Additional Notes on Storage Retention

Systems using "hostpath" storage or a storage class that does not support label selectors may not be able to use the label selector method for using a retained volume volume. You would have to specify the **volumeName** directly, e.g.:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  postgresVersion: 17
  instances:
    - name: instance1
      dataVolumeClaimSpec:
        accessModes:
          - 'ReadWriteOnce'
        resources:
          requests:
            storage: 1Gi
        volumeName: 'pvc-aef7ee64-4495-4813-b896-8a67edc53e58'
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              accessModes:
                - 'ReadWriteOnce'
              resources:
                requests:
                  storage: 1Gi
```

Additionally, to add additional replicas to your Postgres cluster, you will have to make changes to your spec. You can do one of the following:

- Remove the volume-specific configuration from the volume claim spec (e.g. delete **spec.instances.selector** or **spec.instances.volumeName**)
- Add a new instance set specifically for your replicas, e.g.:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  postgresVersion: 17
```

```

instances:
- name: instance1
  dataVolumeClaimSpec:
    accessModes:
      - 'ReadWriteOnce'
    resources:
      requests:
        storage: 1Gi
  selector:
    matchLabels:
      pgo-postgres-cluster: postgres-operator-hippo
- name: instance2
  dataVolumeClaimSpec:
    accessModes:
      - 'ReadWriteOnce'
    resources:
      requests:
        storage: 1Gi
backups:
  pgbackrest:
    repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
              - 'ReadWriteOnce'
            resources:
              requests:
                storage: 1Gi

```

Optional Backups

Info

FEATURE AVAILABILITY: *Available in v5.7.0 and above*

Because Crunchy Postgres for Kubernetes (CPK) was originally designed for production use, disaster recovery was built-in from day one. This was achieved largely through required backups.

However, there are use-cases where you may not want backups. For instance, you might want to start up a temporary **PostgresCluster** for testing purposes and not want to dedicate resources to backups.

For this use-case and others, CPK v5.7+ allows backups to be turned on or off for each **PostgresCluster**.

Running without backups: a few considerations

Running a **PostgresCluster** without backups means some features are no longer available.

First, and most importantly: without backups, there is no practical recovery mechanism. If you run a cluster with backups and accidentally drop an important table, you can restore an older backup and recover that table. If you don't have backups, you don't have that recovery option. For this reason, we really do not recommend running a cluster without backups outside of a few use-cases (temporary test clusters, etc.).

Second, for replicas, a **PostgresCluster** without backups will use **pg_basebackup** to initially create the replica and stream additional changes from the primary. Because of this, when starting a replica, it may speed up the process to run **checkpoint** on the primary first.

Third, you cannot clone a cluster with no backups, since cloning relies on backups. But you can still delete a cluster and retain the pgdata volume and re-use that volume as described in our Data Migration guide.

Fourth, when setting up a standby cluster, you cannot use any repo-based streaming, but you can stream from the primary as described in our streaming tutorial.

Fifth, when monitoring a **PostgresCluster** without backups, the **pgbackrest**-related metrics will be blank, as expected.

Optional Backups: a user guide

Starting a PostgresCluster without backups

With CPK v5.7+, nothing has changed about starting a cluster *with* backups: you need to have a defined **spec . backups** section in your cluster spec.

In order to start a cluster without backups, you can simply remove the **spec . backups** section.

The **spec . backups** section used to be required, and if you are running CPK v5.6 or older, you will get an error from the Kubernetes API saying that the spec is invalid.

However, if you are running CPK v5.7+, a **PostgresCluster** without a **spec . backups** field is valid, and will result in a **PostgresCluster** being created without backups.

Turning on backups

In order to turn on backups when a cluster doesn't have them, you simply need to fill in the **spec . backups** section with your requirements.

To learn more about backup options, see our tutorial on configuring backups for your Postgres cluster.

Once the **spec . backups** section is filled in, CPK will start reconciling the required Kubernetes objects for regular backups.

Turning off backups

Starting a cluster without backups only requires that you remove or leave blank the **spec . backups** section. But turning off backups requires an additional annotation be added to the **PostgresCluster**.

Why? Because turning off backups means removing that backup data; and acts that remove data require additional confirmation.

In this case, to confirm that you want your backups removed, add this annotation to your cluster:

```
postgres-operator.crunchydata.com/authorizeBackupRemoval="true"
```

A sample command to add this annotation is

```
kubectl annotate postgrescluster \<CLUSTER_NAME\> postgres-operator.crunchydata.com/authorizeBackupRemoval="true"
```

Adding that annotation to your cluster will remove the backups and all associated Kubernetes objects: the **PersistentVolume** that held the data, the **StatefulSet** that represented the repo-host, the RBAC Kubernetes objects that allowed the expected access, etc.

Note: CPK will only remove Kubernetes-local data. If you are using cloud-based backups for a **PostgresCluster** and you turn off backups for that cluster, CPK will stop backing up to the cloud--but we do not remove cloud-based backups. You are responsible for cleaning the, e.g., S3 buckets in that case if you want to remove them.

If you remove the **spec.backups** section from a cluster that previously had backups BUT have not yet added the annotation, CPK will pause reconciling that cluster. You can check for this in the cluster status, which will have a message saying that CPK has paused progress on that cluster because the annotation is missing. At this point, you can either add the annotation to remove backups or re-add the **spec.backups** section.

Note: After the backups are removed, it is a best practice to remove the annotation. That way, if you turn on and then off backups at a later date, you will have the opportunity to confirm that you want the backups removed.

How we achieve this

In order to make backups optional, we made two changes to the operator and the **PostgresCluster** CRD:

- We made the **spec.backups** section optional in the CRD.
- CPK now manages the **archive_command** depending on whether the **spec.backups** section is present.

By making **spec.backups** optional, CPK can now add or remove the Kubernetes objects related to backups, just like CPK does with monitoring or other features. (That said, see Turning off backups above for the case where CPK requires additional confirmation to reconcile and remove Kubernetes objects.)

If **spec.backups** is present, CPK sets the **archive_command** to the usual **pgbackrest** command that we use to archive backups. But if **spec.backups** is not present, CPK sets the **archive_command** to a command that automatically returns true. Since Postgres will attempt to archive the backup as usual and then drop the backup if it receives a true command, this means that Postgres will drop those backups as soon as they are archived.

We made the decision to change archiving behavior through setting the **archive_command** since this setting can be changed without restarting the Postgres process. For more on **archive_command**, see the [Postgres docs](#).

Huge Pages

Overview

Huge Pages, a.k.a. "Super Pages" or "Large Pages", are larger chunks of memory that can speed up your system. Normally, the chunks of memory, or "pages", used by the CPU are 4kB in size. The more memory a process needs, the more pages the CPU needs to manage. By using larger pages, the CPU can manage fewer pages and increase its efficiency. For this reason, it is generally recommended to use Huge Pages with your Postgres databases.

Configuring Huge Pages with PGO

To turn Huge Pages on with PGO, you first need to have Huge Pages turned on at the OS level. This means having them enabled, and a specific number of pages preallocated, on the node(s) where you plan to schedule your pods. All processes that run on a given node and request Huge pages will be sharing this pool of pages, so it is important to allocate enough pages for all the different processes to get what they need. This system/kube-level configuration is outside the scope of this document, since the way that Huge Pages are configured at the OS/node level is dependent on your Kube environment. Consult your Kube environment documentation and any IT support you have for assistance with this step.

When you enable Huge Pages in your Kube cluster, it is important to keep a few things in mind during the rest of the configuration process: 1. What size of Huge Pages are enabled? If there are multiple sizes enabled, which one is the default? Which one do you want Postgres to use? 2. How many pages were preallocated? Are there any other applications or processes that will be using these pages? 3. Which nodes have Huge Pages enabled? Is it possible that more nodes will be added to the cluster? If so, will they also have Huge Pages enabled?

Once Huge Pages are enabled on one or more nodes in your Kubernetes cluster, you can tell Postgres to start using them by adding some configuration to your PostgresCluster spec (Warning: setting/changing this setting will cause your database to restart):

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  postgresVersion: 17
  instances:
    - name: instance1
      resources:
        limits:
          hugepages-2Mi: 16Mi
          memory: 4Gi
```

This is where it is important to know the size and the number of Huge Pages available. In the spec above, the **hugepages-2Mi** line indicates that we want to use 2MiB sized pages. If your system only has 1GiB sized pages available, then you will want to use **hugepages-1Gi** as the setting instead. The value after it, **16Mi** in our example, determines the amount of pages to be allocated to this Postgres instance. If you have multiple instances, you will need to enable/allocate Huge Pages on an instance by instance basis. Keep in mind that if you have a "Highly Available" cluster, meaning you have multiple replicas, each replica will also request Huge Pages. You therefore need to be cognizant of the total amount of Huge Pages available on the node(s) and the amount your cluster is requesting. If you request more pages than are available, you might see some replicas/instances fail to start.

Note: In the **instances.#.resources** spec, there are **limits** and **requests**. If a request value is not specified (like in the example above), it is presumed to be equal to the limit value. For Huge Pages, the request value must always be equal to the limit value, therefore, it is perfectly acceptable to just specify it in the **limits** section.

Note: Postgres uses the system default size by default. This means that if there are multiple sizes of Huge Pages available on the node(s) and you attempt to use a size in your PostgresCluster that is not the system default, it will fail. To use a non-default size you will need to tell Postgres the size to use with the **huge_page_size** parameter. (Warning: setting or changing this will cause your database to restart):

```
spec:
  config:
    parameters:
      huge_page_size: 1GB
```

The Kubernetes Issue

There is an issue in Kubernetes where essentially, if Huge Pages are available on a node, it will tell the processes running in the pods on that node that it has Huge Pages available even if the pod has not actually requested any Huge Pages. This is an issue because by default, Postgres is set to "try" to use Huge Pages. When Postgres is led to believe that Huge Pages are available and it attempts to use Huge Pages only to find that the pod doesn't actually have any Huge Pages allocated since they were never requested, Postgres will fail.

We have worked around this issue by setting **huge_pages = off** in our newest Crunchy Postgres images. PGO will automatically turn **huge_pages** back to **try** whenever Huge Pages are requested in the resources spec. Those who were already happily using Huge Pages will be unaffected, and those who were not using Huge Pages, but were attempting to run their Postgres containers on nodes that have Huge Pages enabled, will no longer see their databases crash.

The only dilemma that remains is that those whose PostgresClusters are not using Huge Pages, but are running on nodes that have Huge Pages enabled, will see their **shared_buffers** set to their lowest possible setting. This is due to the way that Postgres' **initdb** works when bootstrapping a database. There are few ways to work around this issue:

- Use Huge Pages! You're already running your Postgres containers on nodes that have Huge Pages enabled, why not use them in Postgres?
- Create nodes in your Kubernetes cluster that don't have Huge Pages enabled, and put your Postgres containers on those nodes.
- If for some reason you cannot use Huge Pages in Postgres, but you must run your Postgres containers on nodes that have Huge Pages enabled, you can manually set the **shared_buffers** parameter back to a good setting (Warning: setting or changing this will cause your database to restart):

```
spec:
  config:
    parameters:
      shared_buffers: 128MB
```

Tablespaces

Warning

PGO tablespaces currently requires enabling the **TablespaceVolumes** feature gate and may interfere with other features. (See below for more details.)

A [Tablespace](#) is a Postgres feature that is used to store data on a different volume than the primary data directory. While most workloads do not require tablespaces, they can be helpful for larger data sets or utilizing particular hardware to optimize performance on a particular Postgres object (a table, index, etc.). Some examples of use cases for tablespaces include:

- Partitioning larger data sets across different volumes
- Putting data onto archival systems
- Utilizing faster/more performant hardware (or a storage class) for a particular database
- Storing sensitive data on a volume that supports transparent data-encryption (TDE)

and others.

In order to use Postgres tablespaces properly in a highly-available, distributed system, there are several considerations to ensure proper operations:

- Each tablespace must have its own volume; this means that every tablespace for every replica in a system must have its own volume;
- The available filesystem paths must be consistent on each Postgres pod in a Postgres cluster;
- The backup & disaster recovery management system must be able to safely backup and restore data to tablespaces.

Additionally, a tablespace is a critical piece of a Postgres instance: if Postgres expects a tablespace to exist and the tablespace volume is unavailable, this could trigger a downtime scenario.

While there are certain challenges with creating a Postgres cluster with high-availability along with tablespaces in a Kubernetes-based environment, the Postgres Operator adds many conveniences to make it easier to use tablespaces.

Enabling TablespaceVolumes in PGO v5

In PGO v5, tablespace support is currently feature-gated. If you want to use this experimental feature, you will need to enable the feature via the PGO **TablespaceVolumes** [feature gate](#).

PGO feature gates are enabled by setting the **PGO_FEATURE_GATES** environment variable on the PGO Deployment. To enable tablespaces, you would want to set

```
PGO_FEATURE_GATES="TablespaceVolumes=true"
```

Please note that it is possible to enable more than one feature at a time as this variable accepts a comma delimited list. For example, to enable multiple features, you would set **PGO_FEATURE_GATES** like so:

```
PGO_FEATURE_GATES="FeatureName=true, FeatureName2=true, FeatureName3=true..."
```

Adding TablespaceVolumes to a postgrescluster in PGO v5

Once you have enabled **TablespaceVolumes** on your PGO deployment, you can add volumes to a new or existing cluster by adding volumes to the **spec.instances.tablespaceVolumes** field.

A **TablespaceVolume** object has two fields: a name (which is required and used to set the path) and a **dataVolumeClaimSpec**, which describes the storage that your Postgres instance will use for this volume. This field behaves identically to the **dataVolumeClaimSpec** in the **instances** list. For example, you could use the following to create a **postgrescluster**:

```
spec:
  instances:
    - name: instance1
```

```

dataVolumeClaimSpec:
  accessModes:
    - 'ReadWriteOnce'
  resources:
    requests:
      storage: 1Gi
tablespaceVolumes:
- name: user
  dataVolumeClaimSpec:
    accessModes:
      - 'ReadWriteOnce'
    resources:
      requests:
        storage: 1Gi

```

In this case, the `postgrescluster` will have 1Gi for the database volume and 1Gi for the tablespace volume, and both will be provisioned by PGO.

But if you were attempting to migrate data from one `postgrescluster` to another, you could re-use pre-existing volumes by passing in some label selector or the `volumeName` into the `tablespaceVolumes.dataVolumeClaimSpec` the same way you would pass that information into the `instances.dataVolumeClaimSpec` field:

```

spec:
instances:
- name: instance1
  dataVolumeClaimSpec:
    volumeName: pvc-1001c17d-c137-4f78-8505-be4b26136924 # A preexisting vol-
ume you want to reuse for PGDATA
    accessModes:
      - 'ReadWriteOnce'
    resources:
      requests:
        storage: 1Gi
  tablespaceVolumes:
- name: user
  dataVolumeClaimSpec:
    accessModes:
      - 'ReadWriteOnce'
    resources:
      requests:
        storage: 1Gi
        volumeName: pvc-3fea1531-617a-4fff-9032-6487206ce644 # A preexisting vol-
ume you want to use for this tablespace

```

Note: the `name` of the `tablespaceVolume` needs to be

- unique in the instance since that name becomes part of the mount path for that volume; * valid as part of a path name, label, and part of a volume name.

There is validation on the CRD for these requirements.

Once you request those `tablespaceVolumes`, PGO takes care of creating (or reusing) those volumes, including mounting them to the pod at a known path (`/tablespaces/NAME`) and adding them to the necessary containers.

How to use Postgres Tablespaces in PGO v5

After PGO has mounted the volumes at the requested locations, the startup container makes sure that those locations have the appropriate owner and permissions. This behavior mimics the startup behavior behind the **PGDATA** directory, so that when you connect to your cluster, you should be able to start using those tablespaces.

In order to use those tablespaces in Postgres, you will first need to create the tablespace, including the location. As noted above, PGO mounts the requested volumes at **/tablespaces/NAME**. So if you request tablespaces with the names **books** and **authors**, the two volumes will be mounted at **/tablespaces/books** and **/tablespaces/authors**.

However, in order to make sure that the directory has the appropriate ownership so that Postgres can use it, we create a subdirectory called **data** in each volume.

To create a tablespace in Postgres, you will issue a command of the form

```
CREATE TABLESPACE name LOCATION '/path/to/dir';
```

So to create a tablespace called **books** in the new **books** volume, your command might look like

```
CREATE TABLESPACE books LOCATION '/tablespaces/books/data';
```

To break that path down: **tablespaces** is the mount point for all tablespace volumes; **books** is the name of the volume in the spec; and **data** is a directory created with the appropriate ownership by the startup script.

Once you have

- enabled the **TablespaceVolumes** feature gate, * added **tablespaceVolumes** to your cluster spec,
- and created the tablespace in Postgres,

then you are ready to use tablespaces in your cluster. For example, if you wanted to create a table called **books** on the **books** tablespace, you could execute the following SQL:

```
CREATE TABLE books (  
  book_id VARCHAR2(20),  
  title VARCHAR2(50)  
  author_last_name VARCHAR2(30)  
)  
TABLESPACE books;
```

Considerations

Only one pod per volume

As stated above, it is important to ensure that every tablespace has its own volume (i.e. its own [persistent volume claim](#)). This is especially true for any replicas in a cluster: you don't want multiple Postgres instances writing to the same volume.

So if you have a single named volume in your spec (for either the main PGDATA directory or for tablespaces), you should not raise the **spec.instances.replicas** field above 1, because if you did, multiple pods would try to use the same volume.

Too-long names?

Different Kubernetes objects have different limits about the length of their names. For example, services follow the DNS label conventions: 63 characters or less, lowercase, and alphanumeric with hyphens U+002D allowed in between.

Occasionally some PGO-managed objects will go over the limit set for that object type because of the user-set cluster or instance name.

We do not anticipate this being a problem with the **PersistentVolumeClaim** created for a tablespace. The name for a **PersistentVolumeClaim** created by PGO for a tablespace will potentially be long since the name is a combination of the cluster, the instance, the tablespace, and the **-tablespace** suffix. However, a **PersistentVolumeClaim** name can be up to 253 characters in length.

Same tablespace volume names across replicas

We want to make sure that every pod has a consistent filesystem because Postgres expects the same path on each replica.

For instance, imagine on your primary Postgres, you add a tablespace with the location **/tablespaces/kafka/data**. If you have a replica attached to that primary, it will likewise try to add a tablespace at the location **/tablespaces/kafka/data**; and if that location doesn't exist on the replica's filesystem, Postgres will rightly complain.

Therefore, if you expand your **postgrescluster** with multiple instances, you will need to make sure that the multiple instances have **tablespaceVolumes** with the *same names*, like so:

```
spec:
  instances:
    - name: instance1
      dataVolumeClaimSpec:
        accessModes:
          - 'ReadWriteOnce'
        resources:
          requests:
            storage: 1Gi
      tablespaceVolumes:
        - name: user
          dataVolumeClaimSpec:
            accessModes:
              - 'ReadWriteOnce'
            resources:
              requests:
                storage: 1Gi
    - name: instance2
      dataVolumeClaimSpec:
        accessModes:
          - 'ReadWriteOnce'
        resources:
          requests:
            storage: 1Gi
      tablespaceVolumes:
        - name: user
          dataVolumeClaimSpec:
            accessModes:
              - 'ReadWriteOnce'
            resources:
              requests:
                storage: 1Gi
```

Tablespace backups

PGO uses `pgBackRest` as our backup solution, and `pgBackRest` is built to work with tablespaces natively. That is, `pgBackRest` should back up the entire database, including tablespaces, without any additional work on your part.

Note: `pgBackRest` does not itself use tablespaces, so all the backups will go to a single volume. One of the primary uses of tablespaces is to relieve disk pressure by separating the database among multiple volumes, but if you are running out of room on your `pgBackRest` persistent volume, tablespaces will not help, and you should first solve your backup space problem.

Adding tablespaces to existing clusters

As with other changes made to the definition of a Postgres pod, adding `tablespaceVolumes` to an existing cluster may cause downtime. The act of mounting a new PVC to a Kubernetes Deployment causes the Pods in the deployment to restart.

Restoring from a cluster with tablespaces

This functionality has not been fully tested.

Removing tablespaces

Removing a tablespace is a nontrivial operation. Postgres does not provide a `DROP TABLESPACE . . CASCADE` command that would drop any associated objects with a tablespace. Additionally, the Postgres documentation covering the `DROP TABLESPACE` command goes on to note:

A tablespace can only be dropped by its owner or a superuser. The tablespace > must be empty of all database objects before it can be dropped. It is possible that objects in other databases might still reside in the tablespace even if no objects in the current database are using the tablespace. Also, if the tablespace is listed in the `temp_tablespaces` setting of any active session, the DROP might fail due to temporary files residing in the tablespace.

Because of this, and to avoid a situation where a Postgres cluster is left in an inconsistent state due to trying to remove a tablespace, PGO does not provide any means to remove tablespaces automatically. If you need to remove a tablespace from a Postgres deployment, we recommend following this procedure:

- As a database administrator: 1. Log into the primary instance of your cluster.
- Drop any objects (tables, indexes, etc) that reside within the tablespace you wish to delete.
- Delete this tablespace from the Postgres cluster using the `DROP TABLESPACE` command.
- As a Kubernetes user who can modify `postgrescluster` specs
- Remove the `tablespaceVolumes` entries for the tablespaces you wish to remove.

More Information

For more information on how tablespaces work in Postgres please refer to the [Postgres manual](#).

Volume Snapshots

Info

FEATURE AVAILABILITY: *Available in v5.7.0 and above*

Volume snapshots are a convenient way to create a copy of a volume's contents without having to create a new **PersistentVolume**. Taking a volume snapshot can be much faster than creating a traditional full backup. Restoring from a snapshot can also be much faster.

Despite the promise of volume snapshots, they also have notable limitations:

- The accessibility of snapshots across zones and regions will vary with your platform. A snapshot created in zone B may require additional work to be made available in zone C.
- Restoring from a naive snapshot can leave you with a corrupted database.

To keep your data safe, Crunchy Postgres for Kubernetes takes the additional steps to make sure that snapshot capture is properly handled in coordination with a traditional backup strategy. This strategy provides the dependability of traditional backups with the benefits of snapshot-based storage.

Prepare your environment

To use the volume snapshot feature, you will first need to know if your Kubernetes cluster has the necessary CRDs and controller to take snapshots. You can check your CRDs by running:

```
kubectl get crd volumesnapshotclasses.snapshot.storage.k8s.io
kubectl get crd volumesnapshotcontents.snapshot.storage.k8s.io
kubectl get crd volumesnapshots.snapshot.storage.k8s.io
```

If you don't have the correct CRDs installed, install them from the [external-snapshotter Github repo](#). You may also need to deploy the snapshot-controller.

Create a **VolumeSnapshotClass**

Now that you've ensured the **VolumeSnapshot** CRDs are installed, the next step is to check that you have a usable **VolumeSnapshotClass** in place. Some Kubernetes clusters will already have a **VolumeSnapshotClass** available and in some

cases you will need to create one yourself. See your platform's documentation for details.

If you already have a **VolumeSnapshotClass** installed, you should be able to find it with:

```
kubectl get volumesnapshotclasses
```

Enable the feature gate

To enable Crunchy Postgres for Kubernetes' volume snapshot feature, activate the **VolumeSnapshot** feature gate. Feature gates are enabled by setting the **PGO_FEATURE_GATES** environment variable on the Crunchy Postgres for Kubernetes Deployment.

```
PGO_FEATURE_GATES="VolumeSnapshots=true"
```

To enable more than one feature at a time, use a comma delimited list. For example, to enable multiple features, you would set **PGO_FEATURE_GATES** like so:

```
PGO_FEATURE_GATES="FeatureName=true, FeatureName2=true, FeatureName3=true..."
```

Enable VolumeSnapshots for your postgrescluster

To enable the automatic capturing of volume snapshots for a given **PostgresCluster**, add the following to your spec:

```
spec:
  backups:
    snapshots:
      volumeSnapshotClassName: <name of the snapshot class>
```

Now, every time you take a manual backup or a scheduled backup runs, the backup will be used to build a consistent snapshot. With every new snapshot taken, the old snapshot will be deleted, ensuring that you do not need to manage snapshots on your own.

Cloning from a snapshot

Once you enable snapshots on the cluster you want to clone, your steps to create the clone are the same as they've always been (see [Clone a Postgres Cluster](#)), though you'll notice some difference in what happens under the hood. Crunchy Postgres for Kubernetes will automatically look for the source cluster's snapshot. If a snapshot is found, a new persistent volume will be populated with the data in the snapshot.

Early tests with data sets up to 100 GB show that turning on snapshots can decrease the time it takes to create a clone by 60%.

Extension Management

[Extensions](#) combine functions, data types, casts, etc. -- everything you need to add some new feature to PostgreSQL in an easy to install package. How easy to install? For many extensions, like the **fuzzystrmatch** extension, it's as easy as connecting to the database and running a command like this:

```
CREATE EXTENSION fuzzystrmatch;
```

However, in other cases, an extension might require additional configuration management. PGO lets you add those configurations to the **PostgresCluster** spec easily.

PGO also allows you to add a custom database initialization script in case you would like to automate how and where the extension is installed.

This guide will walk through adding custom configuration for an extension and automating installation, using the example of Crunchy Data's own **pgnodemx** extension.

pgnodemx

pgnodemx is a PostgreSQL extension that is able to pull container-specific metrics (e.g. CPU utilization, memory consumption) from the container itself via SQL queries.

In order to do this, **pgnodemx** requires information from the Kubernetes [DownwardAPI](#) to be mounted on the PostgreSQL pods. Please see the **pgnodemx and the DownwardAPI** section of the backup architecture page for more information on where and how the DownwardAPI is mounted.

pgnodemx Configuration

To enable the **pgnodemx** extension, we need to set certain configurations. Luckily, this can all be done directly through the spec:

```
spec:
  config:
    parameters:
      shared_preload_libraries: pgnodemx
      pgnodemx.kdapi_enabled: on
      pgnodemx.kdapi_path: /etc/database-containerinfo
```

Those three settings will

- load **pgnodemx** at start; * enable the **kdapi** functions (which are specific to the capture of Kubernetes DownwardAPI information);
- tell **pgnodemx** where those DownwardAPI files are mounted (at the **/etc/dabatase-containerinfo** path).

If you create a **PostgresCluster** with those configurations, you will be able to connect, create the extension in a database, and run the functions installed by that extension:

```
CREATE EXTENSION pgnodemx;
SELECT * FROM proc_diskstats();
```

Automating pgnodemx Creation

Now that you know how to configure **pgnodemx**, let's say you want to automate the creation of the extension in a particular database, or in all databases. We can do that through a custom database initialization.

First, we have to create a ConfigMap with the initialization SQL. Let's start with the case where we want **pgnodemx** created for us in the **hippo** database. Our initialization SQL file might be named **init.sql** and look like this:

```
\c hippo\
CREATE EXTENSION pgnodemx;
```

Now we create the ConfigMap from that file in the same namespace as our PostgresCluster will be created:

```
kubectl create configmap hippo-init-sql -n postgres-operator --from-file=init.sql=path/to/init.sql
```

You can check that the ConfigMap was created and has the right information:

```
kubectl get configmap -n postgres-operator hippo-init-sql -o yaml

apiVersion: v1 data:
  init.sql: |-
    \c hippo\
    CREATE EXTENSION pgnodev;
kind: ConfigMap
metadata:
  name: hippo-init-sql
  namespace: postgres-operator
```

Now, in addition to the spec changes we made above to allow `pgnodev` to run, we add that ConfigMap's information to the PostgresCluster spec: the name of the ConfigMap (`hippo-init-sql`) and the key for the data (`init.sql`):

```
spec:
  databaseInitSQL:
    key: init.sql
    name: hippo-init-sql
```

Apply that spec to a new or existing PostgresCluster, and the pods should spin up with `pgnodev` already installed in the `hippo` database.

Locale and Encoding Settings

By default, CPK clusters are created with the locale `en_US` and `UTF-8` encoding. This is set when the database is initialized and cannot be changed. However, it is possible to create a new database with different locale and encoding settings, provided they are available in the container.

Our containers are built with ICU support which uses the external ICU library. This offers many different locale and language options and should meet most of your needs. CPK also offers LIBC support, but because LIBC uses the locales provided by the operating system, the locales available in the database container may differ from the ones on your operating system. You can use the following command to check the available locales in the database container, replacing `$POD-NAME` with the pod name in your environment:

```
kubectl -n postgres-operator exec -c database $POD-NAME -- locale -a
```

Hint

For a full list of locale options available in the database, use the query `SELECT * FROM pg_collation` the command `\dos+` in psql.

Configuration Methods

There are two methods you can use to create a new database in your CPK cluster: the SQL command **CREATE DATABASE** or the **createdb** utility. These methods are effectively the same, except that the **createdb** utility will call **psql** for us, which some users find more convenient. If you have ever created a new database in PostgreSQL before, then you are already familiar with at least one of these methods.

For both methods, you'll first need to identify the primary pod so you can execute your commands against the database. To make things easier, you can store this information in an environment variable. For example, using a cluster named **hippo**:

Bash:

```
PRIMARY_POD=$(kubectl -n postgres-operator get pods --selector='postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/role=master' -o json-path='{.items[*].metadata.labels.statefulset\.kubernetes\.io/pod-name}')
```

Powershell:

```
$env:PRIMARY_POD=(kubectl -n postgres-operator get pods --selector='postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/role=master' -o json-path='{.items[*].metadata.labels.statefulset\.kubernetes\.io/pod-name}')
```

Now, you can inspect the environment variable to see which Pod is the current primary:

Bash:

```
echo $PRIMARY_POD
```

Powershell:

```
echo $env:PRIMARY_POD
```

This should yield something similar to:

```
hippo-instance1-hltn-0
```

Now that your environment variable is set, let's create some databases!

Method #1: CREATE DATABASE

The first method you can use is the **CREATE DATABASE** command. For this example, you'll create a database named **rhino** using the ICU locale "Japanese" (**ja**) with **UTF8** encoding.

Info

For ICU locales, it is [recommended](#) to use Unicode encodings like UTF-8 whenever possible.

First, exec into the primary pod using your environment variable and connect to the database via **psql**:

Bash:

```
kubectl -n postgres-operator exec -it "$PRIMARY_POD" -- psql
```

Powershell:

```
kubectl -n postgres-operator exec -it "$env:PRIMARY_POD" -- psql
```

Next, run the **CREATE DATABASE** command to create the database **rhino** with your desired settings:

```
postgres=# CREATE DATABASE rhino LOCALE_PROVIDER 'icu' ICU_LOCALE 'ja' ENCODING 'UTF8' TEMPLATE 'template0' ;
CREATE DATABASE
```

Info

Notice that you are using **template0** to create the database instead of **template1**. This is because copying from **template0** allows you to choose different locale and encoding settings, whereas copying from **template1** will use the same parameters that were set when the database was initialized.

Once the database has been created, make sure the locale and encoding settings are correct. This information is stored in the system catalog **pg_database** which can be queried using the **\l** command:

```
postgres=# \l
```

List of databases						
Name	Owner	Encoding	Locale Provider	Collate	Ctype	ICU Locale
hippo	postgres	UTF8	libc	en_US.utf-8	en_US.utf-8	
postgres	postgres	UTF8	libc	en_US.utf-8	en_US.utf-8	
rhino	postgres	UTF8	icu	en_US.utf-8	en_US.utf-8	ja
template0	postgres	UTF8	libc	en_US.utf-8	en_US.utf-8	
template1	postgres	UTF8	libc	en_US.utf-8	en_US.utf-8	
(5 rows)						

Success! From this list, you can see that database **rhino** was created with the ICU locale **ja** and **UTF8** encoding. You can also see your other databases, **postgres** and **hippo**, which were created with the default settings when the cluster was initialized.

Method #2: createdb

With the second method, you'll use the **createdb** utility to create a new database **elephant** using a locale provided by LIBC. For this database, you'll set your locale to British English (**en_GB**) and change the encoding to **LATIN1**.

Using your environment variable from before, run the **createdb** command in the primary pod, setting the **--locale** and **--encoding** flags to reflect your choices. Remember, since you're changing the locale and encoding settings, you will use **template0** instead of **template1** to create the database:

Bash:

```
kubectl -n postgres-operator exec -it "$PRIMARY_POD" -- createdb -T template0 --lo-cale 'en_GB' --encoding 'LATIN1' elephant
```

Powershell:

```
kubectl -n postgres-operator exec -it "$env:PRIMARY_POD" -- createdb -T template0 --lo-cale 'en_GB' --encoding 'LATIN1' elephant
```

Now, check the system catalog `pg_database` to make sure the database was created with the correct settings:

Bash:

```
kubectl -n postgres-operator exec -it "$PRIMARY_POD" -- psql -c '\l'
```

Powershell:

```
kubectl -n postgres-operator exec -it "$env:PRIMARY_POD" -- psql -c '\l'
```

This will yield something similar to:

Defaulted container "database" out of: database, replication-cert-copy, pgbackrest, pgback-rest-config, postgres-startup (init), nss-wrapper-init (init)

				List of databases				
Name	Owner	Encoding	Locale Provider	Collate	Ctype	ICU	Locale	
elephant	postgres	LATIN1	libc	en_GB	en_GB			
hippo	postgres	UTF8	libc	en_US.utf-8	en_US.utf-8			
postgres	postgres	UTF8	libc	en_US.utf-8	en_US.utf-8			
rhino	postgres	UTF8	icu	en_US.utf-8	en_US.utf-8	ja		
template0	postgres	UTF8	libc	en_US.utf-8	en_US.utf-8			
template1	postgres	UTF8	libc	en_US.utf-8	en_US.utf-8			
(6 rows)								

Here, you can see that database `elephant` was created with the locale `en_GB` and `LATIN1` encoding.

Troubleshooting

If the locale and encoding settings you have chosen do not match, you will see an error message like the following:

```
$kubectl -n postgres-operator exec -it "$PRIMARY_POD" -- createdb -T template0 --lo-cale 'en_HK' lion
```

Defaulted container "database" out of: database, replication-cert-copy, pgbackrest, pgback-rest-config, postgres-startup (init), nss-wrapper-init (init)
createdb: error: database creation failed: ERROR: encoding "UTF8" does not match locale "en_HK"
DETAIL: The chosen LC_CTYPE setting requires encoding "LATIN1".
command terminated with exit code 1

Based on the error message, you can see that the locale `en_HK` requires `LATIN1` encoding instead of the default `UTF8` encoding. To resolve this error, add the appropriate encoding option to your command:

Bash:

```
kubectl -n postgres-operator exec -it "$PRIMARY_POD" -- createdb -T template0 --locale 'en_HK' --encoding 'LATIN1' lion
```

Powershell:

```
kubectl -n postgres-operator exec -it "$env:PRIMARY_POD" -- createdb -T template0 --locale 'en_HK' --encoding 'LATIN1' lion
```

This time, you do not see an error. Check the system catalog `pg_database` and make sure your database `lion` was created with the correct locale and encoding settings:

```
postgres=# \l
```

Name	Owner	Encoding	Locale Provider	List of databases			ICU Locale
				Collate	Ctype		
elephant	postgres	LATIN1	libc	en_GB	en_GB		
hippo	postgres	UTF8	libc	en_US.utf-8	en_US.utf-8		
lion	postgres	LATIN1	libc	en_HK	en_HK		
postgres	postgres	UTF8	libc	en_US.utf-8	en_US.utf-8		
rhino	postgres	UTF8	icu	en_US.utf-8	en_US.utf-8	ja	
template0	postgres	UTF8	libc	en_US.utf-8	en_US.utf-8		
template1	postgres	UTF8	libc	en_US.utf-8	en_US.utf-8		
(7 rows)							

Success! The database `lion` was created with your desired settings.

Considerations

Setting `--locale` is equivalent to specifying `--lc-collate`, `--lc-ctype`, and `--icu-locale` to the same value. Some locales are only valid for ICU and must be set with `--icu-locale`. [This table](#) in the Postgres documentation shows which character sets are only valid for ICU and must be set with `--icu-locale`.

The other locale settings `lc_messages`, `lc_monetary`, `lc_numeric`, and `lc_time` are not fixed per database and are not set by this command. If you want to make them the default for a specific database, you can use `ALTER DATABASE ... SET`

pgAdmin

Info

FEATURE AVAILABILITY: Available in v5.5.0 and above

Crunchy Postgres for Kubernetes (CPK) allows deploying pgAdmin either alongside or independently of PostgresClusters. This guide covers configuration options for the PGAdmin API, focusing on two primary setups: one pgAdmin instance per PostgresCluster (one-to-one) or one instance accessing all PostgresClusters in a namespace (one-to-many).

 Hint

The PGAdmin API currently supports all actively maintained versions of Postgres.

Verify your Installation

To ensure proper setup, verify the presence of the PGAdmin Custom Resource Definition (CRD) in your cluster using the following command:

```
kubectl get crd --selector postgres-operator.crunchydata.com/control-plane=postgres-operator
```

NAME	CREATED AT
pgadmins.postgres-operator.crunchydata.com	...

If the PGAdmins CRD is not present, upgrade to v5.5.0 or later.

Create a PGAdmin Deployment

Now that you have verified your installation, we can walk through an example deployment of the PGAdmin API. The first step is to create a Secret for your pgAdmin user password. The following command will create a Secret that contains the password for an example user (**rhino-user**):

```
kubectl create secret generic pgadmin-password-secret -n postgres-operator --from-literal=rhino-password=$RHINO_USER_PASSWORD
```

Where **\$RHINO_USER_PASSWORD** is the password for the user (**rhino-user**).

Once you have created the password Secret, you're ready to define your PGAdmin deployment. Much like a PostgresCluster, a PGAdmin deployment is defined as YAML:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PGAdmin
metadata:
  name: rhino
  namespace: postgres-operator
spec:
  users:
    - username: rhino@example.com
      role: Administrator
      passwordRef:
        name: pgadmin-password-secret
        key: rhino-password
  dataVolumeClaimSpec:
    accessModes:
      - "ReadWriteOnce"
  resources:
```

```
requests:
  storage: 1Gi
serverGroups:
- name: supply
  postgresClusterSelector: {}
```

This YAML defines a PGAdmin named **rhino** that will discover every PostgresCluster in the **postgres-operator** namespace.

Create this resource in your Kubernetes environment, typically by saving it as a file and using **kubectl apply -f pgadmin.yaml**, and CPK will create your pgAdmin deployment.

With your PGAdmin deployment created, you can start a [port-forward](#) to the Pod and log into pgAdmin with your user (**rhino-user**) and password (**\$RHINO_USER_PASSWORD**) at **localhost:5050**.

Once you are connected to pgAdmin, you can access the PostgresClusters that were discovered. Before you can see your Postgres data, you will need to provide your **pguser** password. With that you can use your pgAdmin interface to access your Postgres data.

Deleting a PGAdmin

When you are done using this PGAdmin deployment, you can delete the resource by name or by file:

```
#Delete by name
kubectl delete pgadmin rhino -n postgres-operator
# or Delete by file
kubectl delete -f pgadmin.yaml
```

Configuration

Configuration of the PGAdmin deployment is done using the **config** field in the PGAdmin manifest. This field is broken into a few fields that you might use depending on your environment. In this section we will walk through each of these fields and how you might use them.

pgAdmin settings

The **config.settings** field will be used to set any value that you would find in the [pgAdmin config.py file](#). Some of easiest values to describe are the **SHOW_GRAVATAR_IMAGE** and **DEBUG** settings. The following configuration will enable **DEBUG** mode and disable gravatars when your users log in:

```
spec:
  config:
    settings:
      SHOW_GRAVATAR_IMAGE: False
      DEBUG: True
```

The values provided in **config.settings** are stored in a ConfigMap that is mounted to the pgAdmin Pod. The mounted ConfigMap and its values are passed to pgAdmin through the [config_system.py configuration file](#).

It is worth noting that CPK will own some of the fields, and you won't be able to configure them. A good example of this is the **SERVER_MODE** setting. Since we want pgAdmin to run as a web server and not a desktop app, CPK will always set this value.

Hint

You can check the pgAdmin settings ConfigMap with the following command:

```
kubectl get cm -l postgres-operator.crunchydata.com/pgadmin=rhino -o yaml
```

Settings with Credentials

There are some pgAdmin settings that hold credentials or other sensitive data that you might not want stored as plain-text in your pgAdmin manifest. For some of these settings you can define a Secret reference in a separate field for that setting.

There are two settings that can be configured using a Secret key reference. The LDAP_BIND_PASSWORD setting was available in v5.5 and CONFIG_DATABASE_URI setting is configurable as of v5.6.

To configure these options, provide a Secret name and data key for the password. The following example shows how you can configure both options:

```
spec:
  config:
    ldapBindPassword:
      name: ldappass
      key: password
    configDatabaseURI:
      name: external-db-uri-secret
      key: uri
```

Providing these credential settings using a Secret helps to keep your sensitive data more secure.

Mounting files to the pgAdmin Pod

In some cases you may need to mount configuration files to the pgAdmin Pod. For example, if you want to [configure TLS connections](#) to pgAdmin, you will need to provide cert files. You can mount files by defining ConfigMaps or Secrets in the **config.files** field. The contents of the resources are mounted as [projected volumes](#) to the **/etc/pgadmin/conf.d** in the pgAdmin Pod. The following mounts **tls.crt** of Secret **mysecret** to **/etc/pgadmin/conf.d/tls.crt**:

```
spec:
  config:
    files:
      - secret:
          name: mysecret
          items:
            - key: tls.crt
```

Gunicorn Server Configuration

Info

FEATURE AVAILABILITY: *Available in v5.6.0 and above*

When pgAdmin is deployed through the PostgreSQL Operator, Gunicorn server is used to [run it in server mode](#). You can adjust some Gunicorn server [settings](#) through the `config.gunicorn` of your manifest file. For example, if you are [enabling TLS](#), you can follow these steps:

Create a TLS Secret pointing to your `cert` and `key` files:

```
kubectl create secret tls pgadmin-tls-certs --cert=server.crt --key=server.key
```

Configure your PGAdmin resource with the following `config.gunicorn` fields:

```
config:
  gunicorn:
    keyfile: /etc/pgadmin/conf.d/gunicorn-tls.key
    certfile: /etc/pgadmin/conf.d/gunicorn-tls.crt
  files:
    - secret:
        name: pgadmin-tls-certs
        items:
          - key: tls.crt
            path: gunicorn-tls.crt
          - key: tls.key
            path: gunicorn-tls.key
```

The `config.files` field, mounts the `tls.crt` and `tls.key` files in the `/etc/pgadmin/conf.d/` directory as `gunicorn-tls.crt` and `gunicorn-tls.key`, respectively. With those files in place, the `config.gunicorn` field sets the server's [keyfile](#) and [certfile](#) settings to point to those mounted files, enabling TLS.

Server Discovery

Crunchy Postgres for Kubernetes (CPK) is capable of discovering PostgresClusters so that any user who can sign in to that pgAdmin deployment can any discovered PostgresCluster. How does that work?

In this guide we will walk through two ways that dynamic discovery can discover clusters. These two discovery types can be used to support different deployment methods, notably one PGAdmin to one PostgresCluster and one PGAdmin to many PostgresClusters.

Discovery Types

CPK will use selectors that you provide through the `serverGroups` field to dynamically discover PostgresClusters. The field provides two ways that you can select PostgresClusters, by name and by labels.

The `serverGroups` field is a list type meaning you can configure a combination of discovery types, allowing for more flexibility. Take the following `serverGroup` example:

```
serverGroups:
- name: selector-discovery
```

```
postgresClusterSelector:
  matchLabels:
    environment: production
- name: name-discovery
  postgresClusterName: cluster-name
```

If you were to create a PGAdmin with this **serverGroup** definition, your pgAdmin deployment would discover the PostgresCluster named **cluster-name** and any PostgresCluster that has the **environment** label set to **production**.

If the **serverGroups** field is omitted or if the specified selectors do not match any PostgresClusters, then no servers will be found. In this case, users will need to manually manage ServerGroups and Servers.

Discovery By PostgresCluster Name

Discovery by name is fairly simple, you set the **postgresClusterName** field and provide the name of a PostgresCluster. This PostgresCluster should exist in the same Namespace as your PGAdmin. CPK will then add that PostgresCluster to your defined server group. This discovery type is helpful when you want to deploy one PGAdmin per PostgresCluster.

Discovery by selector

Discovery by Selector provides more options and is helpful when you want to deploy one PGAdmin instance that will monitor many PostgresClusters. The **postgresClusterSelector** field accepts a [Kubernetes Label Selector](#) and can be used in a few ways. Let's walk through the following **pgadmin** example to see how you can use Selectors.

```
spec:
  serverGroups:
    - name: demand
      postgresClusterSelector:
        matchLabels:
          owner: logistics
    - name: supply
      postgresClusterSelector: {}
    - name: maintenance
      postgresClusterSelector:
        matchExpressions:
          - { key: owner, operator: In, values: [logistics, transportation] }
```

Here we have defined three **serverGroups**, showing three separate ways to select on labels.

- The **demand** group has a **postgresClusterSelector** in the **matchLabels** form: any PostgresCluster that matches all of the labels here will be registered automatically.
- The **supply** group matches an empty **postgresClusterSelector**. This is a Kubernetes-specific idiom that will match **all** PostgresClusters in the namespace.
- The **maintenance** group uses the **matchExpressions** format to define what labels to match on.

To be clear, this example is meant to demonstrate several different ways you can define the **postgresClusterSelector**. If you had a PostgresCluster with the label **owner: logistics**, you should be able to log in to your pgAdmin instance and see that PostgresCluster in all three ServerGroups.

Discovered Servers

When a PostgresCluster has been discovered, that cluster will be registered in pgAdmin as a shared server.

Because the server is shared, any user who logs into this pgAdmin will be able to see that PostgresCluster, but will be unable to delete or rename it.

Info

Note: Once you log in to pgAdmin and see PostgresClusters, you will still need a valid Postgres user and credentials to access the Postgres database.

So if you want to deploy one pgAdmin to manage all the PostgresClusters in a namespace and share those servers with all the pgAdmin users, you can set your **pgadmin** deployment to register all those PostgresClusters automatically and skip manually importing them one-by-one!

Warning

If a server is added to any shared server groups, or if the pgAdmin Pod restarts for any other reason, the saved passwords for all servers in the shared server groups in the GUI will be lost and have to re-entered. This occurs because pgAdmin cannot export passwords or add servers without reloading the entire list of servers. This does not occur for any manually added servers or server groups.

User Management

Info

FEATURE AVAILABILITY: *Available in v5.6.0 and above*

In order to log into and use your pgAdmin, you need user credentials. Crunchy Postgres for Kubernetes (CPK) provides a way to manage internal pgAdmin users through the PGAdmin API.

Defining Users

Users are defined in the **users** field in the PGAdmin custom resource. Below is an example of a PGAdmin manifest with a single user in place:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PGAdmin
metadata:
  name: rhino
spec:
  dataVolumeClaimSpec:
    accessModes:
      - "ReadWriteOnce"
    resources:
      requests:
        storage: 1Gi
  serverGroups:
    - name: supply
```

```
postgresClusterSelector: {}
users:
- username: user@example.com
  role: User
  passwordRef:
    name: user-password-secret
    key: password
```

Info

Note: If a user already exists in pgAdmin, presumably added via the pgAdmin GUI, the operator will not be able to “take control” of that user if it is added to the **users** field. We therefore recommend committing to one user management method or the other: managing users via the PGAdmin spec or via the pgAdmin GUI.

User Properties

When defining a user, there are three properties that you can set:

- **username** (required) - The username for the user. The username for these internal users must be in [email format](#).
- **role** (optional) - The role that you want to give this user. The options are **Administrator** and **User**. If left unset, the role will default to **User**.
- **passwordRef** (required) - A reference to a Kubernetes Secret and key that hold the password for this user.
- **name** (required) - The name of the Secret.
- **key** (required) - The key inside the Secret that holds the password.

Given the sensitive nature of passwords, we require that the passwords be stored in Secrets and then obtained by referencing those Secrets in the user **spec**. Given that the **passwordRef** references not only the name of the Secret, but also the key that holds a particular password, you can choose to have a separate Secret for each user or you can store multiple passwords in one Secret.

Warning

pgAdmin allows users to set a minimum password length with the **PASSWORD_LENGTH_MIN** setting. The default minimum password length is 6 characters. If you adjust this setting, ensure your passwords meet the updated minimum.

Example

Let's say you wanted to add two users, where one has admin privileges and the other does not, and you want to store both passwords in one Secret. You would start by creating the Secret:

```
kubectl create secret generic pgadmin-password-secret -n postgres-operator --from-literal=hippo-password=$HIPPO_USER_PASSWORD --from-literal=elephant-password=$ELEPHANT_USER_PASSWORD
```

This creates a Secret called **pgadmin-password-secret** in the **postgres-operator** namespace, with two keys: **hippo-password** and **elephant-password**.

You would then update the `users` field in your PGAdmin manifest to create these two users and reference the Secret you just created to set the users' passwords.

```
spec:
  users:
  - username: hippo@example.com
    role: Administrator
    passwordRef:
      name: pgadmin-password-secret
      key: hippo-password
  - username: elephant@example.com
    passwordRef:
      name: pgadmin-password-secret
      key: elephant-password
```

Notice that the `role` setting was omitted for the `elephant@example.com` user and will therefore default to a `User` role.

If you change the password stored in your Secret, the operator will automatically update the password in pgAdmin. You can also change the `passwordRef` to point to a different Secret or key. Likewise, you can change a user's `role` and it will be updated in pgAdmin.

Warning

While the `passwordRef` and `role` properties can be changed, a user's `username` cannot be modified. If you “change” the `username` for a given user in the spec, the operator will perceive this as the removal of the old user and the creation of a new user. The “old” user will be removed from the Postgres Operator's local database of users; however, **the user will not be removed from pgAdmin**. If you wish to fully delete any user, you will need to remove the user from your `spec` and then delete the user via the pgAdmin GUI while logged in as a user with an Administrator role.

Retrieving and editing passwords

If you've forgotten a password, you can either retrieve it from the Secret or change the password altogether. Let's start by retrieving a password.

Following the example in the previous section, let's say we want to retrieve the password for the `hippo@example.com` user. We can do this with the following command:

Bash:

```
kubectl get secret/pgadmin-password-secret -n postgres-operator -o 'go-template={{index .data "hippo-password" | base64decode }}'
```

Powershell:

```
kubectl get secret/pgadmin-password-secret -n postgres-operator -o 'go-template={{index .data \"hippo-password\" | base64decode }}'
```

If we instead wanted to change this password, we could do that with the following command:

Bash:

```
kubectl patch secret/pgadmin-password-secret -n postgres-operator -p='{"stringData":{"hippo-password": "$NEW_PASSWORD" }}'
```

Powershell:

```
kubectl patch secret/pgadmin-password-secret -n postgres-operator -p='{\"stringData\":{ \"hippo-password\": \"\\\"$NEW_PASSWORD\\\"\" : \"\\\"$NEW_PASSWORD\\\"\" }}'
```

Where `"$NEW_PASSWORD"` is your new password.

Connectivity

Connecting to pgAdmin

There are a few ways to connect to your pgAdmin server. If you have access to `kubectl` in your Kubernetes environment, you can use [port-forward to access the pgAdmin Pod](#) directly. This works fine for testing, but for production deployments you might want to consider using a [Kubernetes Service](#).

We recommend looking to the [Kubernetes networking documentation](#) for specifics around networking. Kubernetes provides many ways to handle networking and connections to your Pod that won't be covered here. We will walk through some basic setup that will get you connected to your pgAdmin interface.

Connecting directly to the Pod

You can use `port-forward` to connect to directly to the pgAdmin Pod. This will give you access to pgAdmin on your local machine through a browser.

When starting a `port-forward` to the pgAdmin Pod, you need to determine the name of the Pod for your PGAdmin deployment. You can do this by using `kubectl get` and selecting the Pod with the `postgres-operator.crunchydata.com/pgadmin` label. You can save the Pod name to the variable `PGADMIN_POD` to make it easier to reuse:

Bash:

```
export PGADMIN_POD=$(kubectl get pod -n postgres-operator --selector="postgres-operator.crunchydata.com/pgadmin=rhino" -o name)
```

Powershell:

```
$env:PGADMIN_POD=$(kubectl get pod -n postgres-operator --selector="postgres-operator.crunchydata.com/pgadmin=rhino" -o name)
```

Once you've identified your pgAdmin Pod, you can `port-forward` to it directly:

Bash:

```
kubectl port-forward -n postgres-operator ${PGADMIN_POD} 5050:5050
```

Powershell:

```
kubectl port-forward -n postgres-operator ${env:PGADMIN_POD} 5050:5050
```

Once the connection is established, you can [connect over the port-forward](#).

Connecting through a Service

You also have the option to [create a Service](#) to connect. If you are using a Service, the easiest way to connect is to start a **port-forward** connection that points to that Service. In this case you only need to know the name of the Service.

```
kubectl port-forward service/$MY_SERVER 5050:5050
```

Where **\$MY_SERVER** is name of the Service.

Once the connection is established, you can [connect over the port-forward](#). This is a good way to test that your Service is working correctly.

However, it still might not be your preferred connection method in production. For alternative methods, reference the [Kubernetes documentation](#) or our [OpenShift Route documentation](#).

Connecting through an OpenShift Route

An OpenShift [Route](#) is one way to accomplish application hosting at a public URL when using OpenShift. While the possibilities for configuration are extensive, a simple HTTP connection can be accomplished with a few simple steps. First, assuming you have a Service defined named **my-service** (see [Creating a Service](#) for more details), you could define a Route as follows:

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: hello-pgadmin
spec:
  host: hello-pgadmin.$INGRESS_DOMAIN
  port:
    targetPort: pgadmin-port
  to:
    kind: Service
    name: my-service
```

where **\$INGRESS_DOMAIN** is the default Ingress domain name. One way to easily get that value is by using

```
oc get ingresses.config/cluster -o jsonpath='{.spec.domain}'
```

After creating this Route, in a web browser navigate to [http://hello-pgadmin.\\$INGRESS_DOMAIN](http://hello-pgadmin.$INGRESS_DOMAIN) and login to pgAdmin using a defined user.

Creating a Service

With the PGAdmin API you have two options for creating a Service. You can either provide a **ServiceName** in your PGAdmin manifest to create a [ClusterIP](#) Service or you can manually create a Service as part of your deployment.

Creating a ClusterIP Service with PGAdmin API

CPK provides the ability to create a **ClusterIP** Service that points to your pgAdmin Pod. You can configure this by providing a name in the **spec.ServiceName** field.

```
spec:
  serviceName: "my-service"
```

Warning

If the Service you provide through **serviceName** already exists in your environment and is not owned by CPK, CPK will not take ownership of that Service.

CPK will create a **ClusterIP** Service using the name that you provide. This Service will be configured to point to the pgAdmin web server and will be owned by your PGAdmin custom resource and labeled like any other PGAdmin resource.

After the Service is created, you can make some adjustments to the Service, like adding labels or annotations. If you need further adjustments, we recommend [manually creating a service](#) that meets your needs.

Creating a Service manually

If you need to modify your **ClusterIP** Service, or you require other Service types (like **LoadBalancer** or **NodePort** Services), you have the ability to create your own Service and point it at pgAdmin.

Whichever type of Service you create will need to point to the pgAdmin Pod and port. This is done by setting the **selector** and **port** fields on the Service.

In the example below we are pointing to the Pod for PGAdmin **my-pgadmin** using the **postgres-operator.crunchydata.com/pgadmin: my-pgadmin** label. We also configure the service to point to port **5050**, the default port for pgAdmin.

Additional configuration will depend on your Kubernetes environment and the available networking options. You can reference the [Kubernetes Service documentation](#) for information on types of Services.

In our example we will assume you have a Kubernetes cluster that supports the **NodePort** Service type and that **NodePort 30050** is [allowed in your cluster](#). You can create the following Service that will point to pgAdmin:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  ports:
    - name: pgadmin-port
      port: 5050
      protocol: TCP
      nodePort: 30050
  selector:
    postgres-operator.crunchydata.com/pgadmin: my-pgadmin
```

Once the **NodePort** Service is created you will be able to connect to pgAdmin on the node where your Kubernetes cluster is running.

Related documentation

Configuring TLS connections to pgAdmin

Migration from PostgresCluster API

The PGAdmin API is the new way to deploy pgAdmin with Crunchy Postgres for Kubernetes (CPK). In this guide, we walk through how to migrate your pgAdmin deployment from the PostgresCluster API to the PGAdmin API.

Info

FEATURE AVAILABILITY *This guide uses features that are available in CPK v5.6.0 and above.*

Why migrate?

Deploying pgAdmin through the PostgresCluster API limits your configuration options in quite a few ways:

- pgAdmin deployments are only compatible with PostgreSQL 14 and below.
- pgAdmin users and their passwords must be the same as specific Postgres users defined in the PostgresCluster.
- pgAdmin usernames will always have the **@pgo** suffix.

Besides the limitations just mentioned, the PGAdmin API has significant improvements over the PostgresCluster version:

- Added support for new versions of pgAdmin that include feature enhancements, bug fixes, and security patches
- New features like:• declarative user passwords• better configuration options• server discovery• connectivity options

In this section we will walk through how to configure your PGAdmin manifest to replicate your PostgresCluster API-based deployment. Some of these fields can be copied directly from your PostgresCluster manifest while others are either new or need to be configured differently. Before we talk about [how to migrate](#) your PostgresCluster API-based pgAdmin, let's walk through how the PGAdmin API is different.

How does the PGAdmin API compare?

What hasn't changed?

Configuration

Configuration of pgAdmin settings and mounting of files is the same between both APIs. These options are still configured through a **config** section of your manifest. You can define pgAdmin settings using **settings** and mount files to the pgAdmin Pod using **files**. If you have your pgAdmin connected to an LDAP server, you can use **ldapBindPassword** to securely provide your credential. You can copy these fields directly over from your PostgresCluster manifest. Your mounted files and settings will be applied to your new pgAdmin deployment in the same way as your existing deployment.

Hint

There are new sub-fields in the `config` field that relate to new features, look into those in our [pgAdmin configuration docs](#).

Generic Kubernetes Options

Some of the configuration options in the PostgresCluster API are basic Kubernetes configuration options that aren't specific to pgAdmin. For example, the `dataVolumeClaimSpec` is a standard Kubernetes field that defines the size of your pgAdmin PVC. Other examples are the `metadata`, `resources`, and `affinity` fields. These fields can also be copied from your PostgresCluster manifest and defined in the PGAdmin manifest.

What has changed?

User management

User management with pgAdmin has changed significantly with the PGAdmin API. With the PostgresCluster API, users and passwords were created based on the Postgres users that you defined in your manifest. With PGAdmin, users are still defined in the `users` section of your manifest but are unrelated to Postgres. This allows you to update users and rotate passwords separately from Postgres.

pgAdmin requires usernames to be in the [email format](#). With the PostgresCluster API, this condition was met by adding the `@pgo` suffix to your user. Now you have the ability to provide your own email as the username, meaning you can have `<user>@my.company.com`.

You can also declaratively define and rotate your password by providing a reference to a Kubernetes Secret. Before you create your PGAdmin resource, you will need to create this Secret.

Service Creation

The PostgresCluster API for pgAdmin contains a copy of the Kubernetes Service spec, configurable through the `service` field. This field allows you to effectively pass a Service definition through to Kubernetes. With the PGAdmin API, we have decided against providing this type of pass-through configuration of the pgAdmin Service. Instead, the `serviceName` field of the PGAdmin API produces a simple ClusterIP service.

Services and connections to your pgAdmin deployment will vary depending on your Kubernetes environment. We go into more detail about connecting to your pgAdmin and creating Services in our [connectivity documentation](#).

What's new?

Server Discovery

Any pgAdmin deployments created with the PGAdmin API are not tied to a specific PostgresCluster. This provides the flexibility to create a single pgAdmin that can manage multiple PostgresClusters. The PGAdmin API can be configured to discover servers in your Kubernetes namespace using the `serverGroups` field. More information about this can be found in the [server discovery documentation](#).

However, you can still easily create a pgAdmin deployment that can only access a single PostgresCluster. This is done by providing your PostgresCluster deployment name through the PGAdmin API. You even have the ability to define your own server group name in the pgAdmin interface!

Migrating to the PGAdmin API

What does a PostgresCluster pgAdmin manifest look like?

Consider a PostgresCluster with the following pgAdmin fields:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  users:
    - name: rhino
      databases:
        - zoo
  userInterface:
    pgAdmin:
      config:
        settings:
          SHOW_GRAVATAR_IMAGE: False
        files:
          - configMap:
              name: myconfigmap
              optional: false
      dataVolumeClaimSpec:
        accessModes:
          - 'ReadWriteOnce'
        resources:
          requests:
            storage: 1Gi
  ...
```

First, notice that this is not a complete PostgresCluster manifest. These are only fields that relate to pgAdmin in some way. Any other fields will be left in your PostgresCluster.

The user definition for **rhino** creates that user in Postgres and an associated non-administrator user in pgAdmin named **rhino@pgo**. You will use the same password, stored in the **hippo-pguser-rhino** Secret, to log in to both Postgres and pgAdmin.

Under the **config** section of the manifest, we have configuration for **files** and **settings**. The **SHOW_GRAVATAR_IMAGE** setting is disabled and we are mounting the contents of **myconfigmap** to **/etc/pgadmin/conf.d** in the Pod.

We define a **dataVolumeClaimSpec** of size 1GiB that pgAdmin will use to store persistent data, like the SQLite DB file.

Additionally, a [ClusterIP Service](#) named **hippo-pgadmin** is created by default.

With the above in mind, let's look at a similar configuration using the PGAdmin API.

Replicating your PostgresCluster pgAdmin

First, the PostgresCluster manifest is simplified leaving only the **users** section:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo2
spec:
  users:
    - name: rhino
      databases:
        - zoo
  ...
```

The **users** section of the PostgresCluster manifest is unchanged. We are still creating a Postgres user **rhino** and database **zoo**. You will still need the credentials of the **rhino** Postgres user when connecting pgAdmin to your database, after you have successfully logged in to pgAdmin.

However, the **userInterface** section and all pgAdmin specific configuration is now defined in the PGAdmin manifest. Let's consider a PGAdmin manifest:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PGAdmin
metadata:
  name: hippo2-pgadmin
spec:
  users:
    - username: "rhino@example.com"
      passwordRef:
        name: pgadmin-password
        key: password-data
  config:
    settings:
      SHOW_GRAVATAR_IMAGE: False
    files:
      - configMap:
          name: myconfigmap
          optional: false
  serviceName: hippo2-pgadmin # based on the PostgresCluster name
  dataVolumeClaimSpec:
    accessModes:
      - "ReadWriteOnce"
    resources:
      requests:
        storage: 1Gi
  serverGroups:
    - name: "Crunchy PostgreSQL Operator"
      postgresClusterName: hippo2
```

User Creation

In the PGAdmin manifest, we still have a **users** field, but the definition has different fields. The **username** field can be any string in an [email format](#) and the **passwordRef** field will point to a Secret that contains your password. You can create the Secret with the following command:

```
kubectl create secret generic pgadmin-password --from-literal=password-data=$YOUR_PASSWORD
```

In the example, we create the `rhino@example.com` pgAdmin user and set the password to the contents of the `pgadmin-password` secret. Learn more about user management in our user management docs.

Configuration

Like in the PostgresCluster example, we provide configuration options through the `config` field. The `settings` and `files` fields look exactly the same as from our PostgresCluster manifest and can be copied directly over. Your files will still be mounted at `/etc/pgadmin/conf.d` and your pgAdmin settings will be set. Learn more about configuration in the configuration docs.

Service

Unlike the PostgresCluster API, the PGAdmin API will not create a Service by default. If you are using the default Service, you can replicate this behavior by setting the `serviceName` field in your PGAdmin manifest. If you do not need a Service, you can simply leave out the `serviceName` field.

The `serviceName` field will create a [ClusterIP Service](#) with the same naming as the default PostgresCluster pgAdmin deployment (`<cluster-name>-pgadmin`). In our example, we set `serviceName` to `hippo2-pgadmin`. If you need a different type of Service, consult our connectivity docs.

Data Volume

We define a `dataVolumeClaimSpec` of size 1GiB that pgAdmin will use to store persistent data, like the SQLite DB file. You can copy this spec from your PostgresCluster manifest directly to your PGAdmin manifest at `dataVolumeClaimSpec`. You can also copy over other generic Kubernetes options, like the `affinity`, `metadata`, or `resources` fields.

Server Discovery

Finally, you will need to tell the PGAdmin API what PostgresCluster that it should discover. Since we are replicating a PostgresCluster deployment, where PGAdmin can only see one PostgresCluster, we will select the PostgresCluster by name.

In our example, we define a `serverGroup` named `Crunchy PostgreSQL Operator` and set it to discover a single PostgresCluster named `hippo2`. Learn more about server discovery in the server discovery docs.

Next steps

Using the PGAdmin manifest, you can configure a pgAdmin deployment to replicate a one PostgresCluster to one pgAdmin deployment. Similar manifests can be created for other PostgresClusters or you can deploy one pgAdmin that can discover many PostgresClusters. You have the flexibility to choose!

There are some configuration options that we did not cover in this guide. For example, you might be interested in and advanced configuration like LDAP. There is also new functionality that wasn't available through a PostgresCluster API-based deployment, notably TLS configuration using Unicorn.

If you don't see something in this guide, read through the PGAdmin API docs or feel free to reach out in Discord.

Advanced Configuration

This guide walks through different use cases that go beyond a basic deployment. These features require extra configuration that needs to be done outside of pgAdmin. For example, deploying an LDAP server or a PostgreSQL database to use as the pgAdmin settings database.

Authentication Sources

The **AUTHENTICATION_SOURCES** setting in pgAdmin allows you to adjust the ways in which users can authenticate. By default, pgAdmin is setup to only allow internal users, users that are stored in the pgAdmin settings database, to authenticate. By adding options to the **AUTHENTICATION_SOURCES** list, you can enable other sources.

If you wanted your pgAdmin users to be able to authenticate via LDAP, in addition to using internal authentication, you would need to include **ldap** option in the **AUTHENTICATION_SOURCES** setting array:

```
spec:
  config:
    settings:
      AUTHENTICATION_SOURCES: ['ldap', 'internal']
```

The first source in the list will have a higher priority, meaning you can use **ldap** as your first source and **internal** as a fallback in case **ldap** fails.

LDAP Configuration

The [pgAdmin config.py file](#) has configuration options to enable [LDAP authentication into pgAdmin](#). These settings will depend on your LDAP server. We will go through some simple examples here to show how you can connect to an LDAP server.

Basic connection

You will configure a majority of LDAP settings using the **config.settings** field. The first step to enabling LDAP is to update your **AUTHENTICATION_SOURCES** setting to include the new source. CPK requires that you enable the **LDAP_AUTO_CREATE_USER** setting so that pgAdmin will create a pgAdmin user for any LDAP user that successfully logs in.

```
spec:
  config:
    settings:
      AUTHENTICATION_SOURCES: ['ldap', 'internal']
      LDAP_AUTO_CREATE_USER: True # Required if using LDAP
```

This is also where you will configure your **LDAP_SERVER_URI** and other LDAP settings, like **LDAP_SEARCH_BASE_DN** or **LDAP_ANONYMOUS_BIND**. Reference the [pgAdmin LDAP documentation](#) for more information about LDAP settings.

LDAP Bind User and Password

Depending on your LDAP configuration, you might need to define a user and password that will bind pgAdmin to the LDAP server. These options are defined in [config.py](#) as **LDAP_BIND_USER** and **LDAP_BIND_PASSWORD**. You will define

the `LDAP_BIND_USER` like you would any other setting. However, the `LDAP_BIND_PASSWORD` is not something that we recommend storing in your PGAdmin spec. Instead, CPK provides the `ldapBindPassword` field that lets you point at a Secret:

```
spec:
  config:
    settings:
      LDAP_BIND_USER: $user
    ldapBindPassword:
      name: ldappass
      key: $password
```

This field is a Secret key reference that will be mounted to the pgAdmin Pod. CPK will configure pgAdmin to look in the mounted file instead of using the plaintext `LDAP_BIND_PASSWORD` setting. This helps to keep your password secure.

Connection to a TLS LDAP server

If you are connecting to a LDAP server using TLS, you will need to provide cert files to secure the connection. Like we talked about in the configuration docs, you will need to mount your cert files to the pgAdmin Pod. Once the files are available to pgAdmin, you will need to tell pgAdmin where to look for them. This is done using the `LDAP_CA_CERT_FILE`, `LDAP_CERT_FILE`, and `LDAP_KEY_FILE` settings. Your final spec should include something like this:

```
spec:
  config:
    settings:
      LDAP_SERVER_URI: ldaps://my.ds.example.com
      LDAP_CA_CERT_FILE: /etc/pgadmin/conf.d/certs/ca.crt
      LDAP_CERT_FILE: /etc/pgadmin/conf.d/certs/tls.crt
      LDAP_KEY_FILE: /etc/pgadmin/conf.d/certs/tls.key
    files:
      - secret:
          name: openldap
          items:
            - key: ca.crt
              path: certs/ca.crt
            - key: tls.crt
              path: certs/tls.crt
            - key: tls.key
              path: certs/tls.key
```

OAuth2 Configuration

The [pgAdmin config.py file](#) also has configuration options to enable [OAuth2 authentication for pgAdmin](#). These settings will depend on your OAuth2 server. As with LDAP, we will go through some simple examples here to show how you can connect to an OAuth2 server.

Example Configurations

You will configure the OAuth2 settings using the `config.settings` field. The first step to enabling OAuth2 is to update your `AUTHENTICATION_SOURCES` setting to include the new source. CPK requires that you enable the `OAUTH2_AUTO_CREATE_USER` setting so that pgAdmin will create a pgAdmin user for any OAuth2 user that successfully

 You must sign in to view this resource. 



pgAdmin

Login

[Forgotten your password?](#)

Login

Login with Google

Login with Github

External pgAdmin settings Database

Configuring an External database for pgAdmin user settings

By default, the pgAdmin user settings are stored in a local SQLite database. However, pgAdmin does provide a configuration setting for defining a database connection string to an external database. This setting is the [CONFIG DATABASE URI parameter](#). The expected parameter must be given in the following format:

```
dialect+driver://username:password@host:port/database
```

While it is possible to set this value directly in `config.settings` like other pgAdmin configuration settings, this connection string often contains sensitive information, so storage in a Secret is recommended. As a simple example, if you had a basic PostgresCluster named `hippo`, by default you could use a connection string similar to

```
postgresql://hippo:$MY_PASSWORD@hippo-primary.postgres-operator.svc:5432/hippo
```

where `$MY_PASSWORD` is updated to your user password. By default, a PostgresCluster named `hippo` would have a Secret named `hippo-pguser-hippo` that contains a URI similar to the one above. To use that value for your external database, you would configure your PGAdmin as follows:

```
spec:
  config:
```

```
configDatabaseURI:
  name: hippo-pguser-hippo
  key: uri
```

Just be sure to remember, when using Postgres 15+ you will need to verify your user has creation permissions in the default schema, as described in the quickstart). If the user does not have creation permissions, pgAdmin won't be able to create the needed tables!

In cases where you want to define a specific schema, you can also create your own Secret with more specific settings. For instance, if you wanted to use the connection string that specified a specific schema such as

```
postgresql://hippo:$MY_PASSWORD@hippo-primary.postgres-operator.svc:5432/hippo?options=-csearch_path=myschema
```

you could create a Secret as follows:

```
kubectl create secret generic config-db-uri-myschema --from-literal=uri="postgresql://hippo:$MY_PASSWORD@hippo-primary.postgres-operator.svc:5432/hippo?options=-csearch_path=myschema"
```

and then reference that Secret in your pgAdmin manifest

```
spec:
  config:
    configDatabaseURI:
      name: config-db-uri-myschema
      key: uri
```

Warning

When using external databases for pgAdmin, please be sure to configure distinct storage locations (schemas, databases, etc) when using multiple pgAdmin instances and remove old data when no longer needed. This will ensure you avoid potential data conflicts between different pgAdmins.

As with LDAP bind password, the `configDatabaseURI` parameter is a Secret key reference that will be mounted to the pgAdmin Pod allowing you to avoid storing credentials in plaintext. Using this information, your pgAdmin instance will be able to store its user settings in whichever location you define independently of the PGAdmin Pod.

pgAdmin v4.30

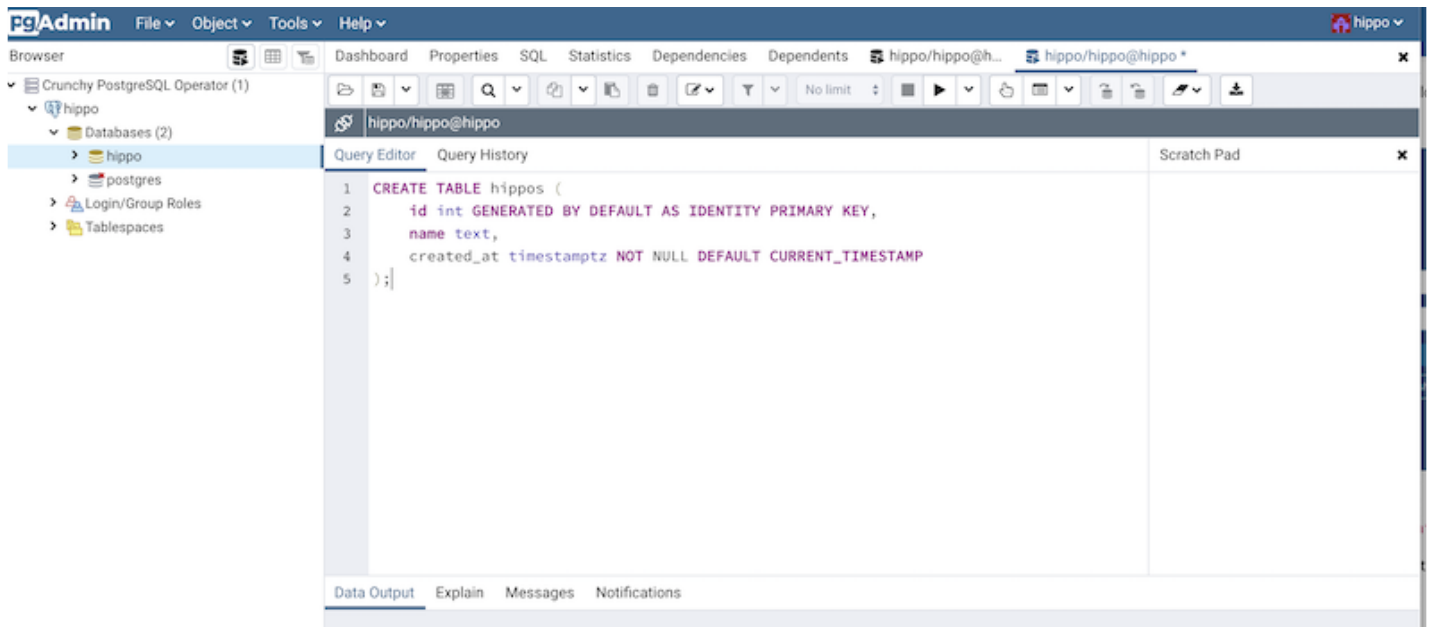
Warning

The information on this page pertains to pgAdmin v4.30 deployments that are created using the PostgresCluster API.

- PgAdmin v4.30 deployments are not compatible with PostgreSQL 15 and newer.
- Updates to PostgresCluster API based PgAdmin deployments have ceased.

- PgAdmin v4.30 is only available on UBI 8. Users interested in running UBI 9 images should migrate to the PgAdmin API.

Migrate to the PGAdmin API for the latest Postgres and pgAdmin versions. This API also includes the newest features and functionality.



[pgAdmin 4](#) is a popular graphical user interface that makes it easy to work with PostgreSQL databases from a web-based client. With its ability to manage and orchestrate changes for PostgreSQL users, the PostgreSQL Operator is a natural partner to keep a pgAdmin 4 environment synchronized with a PostgreSQL environment.

The PostgreSQL Operator lets you deploy pgAdmin 4 alongside a PostgreSQL cluster and keeps users' database credentials synchronized. You can simply log into pgAdmin 4 with your PostgreSQL username and password and immediately have access to your databases.

Deploying pgAdmin 4

If you've done the quickstart, add the following fields to the spec and reapply; if you don't have any Postgres clusters running, add the fields to a spec, and apply.

```
userInterface:
  pgAdmin:
    dataVolumeClaimSpec:
      accessModes:
        - 'ReadWriteOnce'
      resources:
        requests:
          storage: 1Gi
```

This creates a pgAdmin 4 deployment unique to this PostgreSQL cluster and synchronizes the PostgreSQL user information. To access pgAdmin 4, you can set up a port-forward to the Service, which follows the pattern **<cluster-Name>-pgadmin**, to port **5050**:

```
kubectl port-forward svc/hippo-pgadmin 5050:5050
```

Point your browser at **http://localhost:5050** and you will be prompted to log in. Use your database username with **@pgo** appended and your database password. In our case, the pgAdmin username is **hippo@pgo** and the password is found in the user secret, **hippo-pguser-hippo**:

Bash:

```
PG_CLUSTER_USER_SECRET_NAME=hippo-pguser-hippo
PGPASSWORD=$(kubectl get secrets -n postgres-operator "${PG_CLUSTER_USER_SECRET_NAME}" -o go-template='{{.data.password | base64decode}}')
PGUSER=$(kubectl get secrets -n postgres-operator "${PG_CLUSTER_USER_SECRET_NAME}" -o go-template='{{.data.user | base64decode}}')
```

Powershell:

```
$env:PG_CLUSTER_USER_SECRET_NAME="hippo-pguser-hippo"
$env:PGPASSWORD=(kubectl get secrets -n postgres-operator "${env:PG_CLUSTER_USER_SECRET_NAME}" -o go-template='{{.data.password | base64decode}}')
$env:PGUSER=(kubectl get secrets -n postgres-operator "${env:PG_CLUSTER_USER_SECRET_NAME}" -o go-template='{{.data.user | base64decode}}')
```



Hint

If your password does not appear to work, you can retry setting up the user by rotating the user password. Do this by deleting the **password** data field from the user secret (e.g. **hippo-pguser-hippo**).

Optionally, you can also set a custom password.

User Synchronization

The operator will synchronize users defined in the spec (e.g., in **spec.users**) with the pgAdmin 4 deployment. Any user created in the database without being defined in the spec will not be synchronized.

Custom Configuration

You can adjust some pgAdmin settings through the `userInterface.pgAdmin.config` field. For example, set `SHOW_GRAVATAR_IMAGE` to `False` to disable automatic profile pictures:

```
userInterface:
  pgAdmin:
    config:
      settings:
        SHOW_GRAVATAR_IMAGE: False
```

You can also mount files to `/etc/pgadmin/conf.d` inside the pgAdmin container using [projected volumes](#). The following mounts `useful.txt` of Secret `mysecret` to `/etc/pgadmin/conf.d/useful.txt`:

```
userInterface:
  pgAdmin:
    config:
      files:
        - secret:
            name: mysecret
            items:
              - key: useful.txt
        - configMap:
            name: myconfigmap
            optional: false
```

Kerberos Configuration

You can configure pgAdmin to [authenticate its users using Kerberos](#) SPNEGO. In addition to setting `AUTHENTICATION_SOURCES` and `KRB_APP_HOST_NAME`, you need to enable `KERBEROS_AUTO_CREATE_USER` and mount a `krb5.conf` and a keytab file:

```
userInterface:
  pgAdmin:
    config:
      settings:
        AUTHENTICATION_SOURCES: ['kerberos']
        KERBEROS_AUTO_CREATE_USER: True
        KRB_APP_HOST_NAME: my.service.principal.name.local # without HTTP class
        KRB_KTNAME: /etc/pgadmin/conf.d/krb5.keytab
      files:
        - secret:
            name: mysecret
            items:
              - key: krb5.conf
              - key: krb5.keytab
```

LDAP Configuration

You can configure pgAdmin to [authenticate its users using LDAP](#) passwords. In addition to setting `AUTHENTICATION_SOURCES` and `LDAP_SERVER_URI`, you need to enable `LDAP_AUTO_CREATE_USER`:

```
userInterface:
  pgAdmin:
    config:
```

```
settings:
  AUTHENTICATION_SOURCES: ['ldap']
  LDAP_AUTO_CREATE_USER: True
  LDAP_SERVER_URI: ldaps://my.ds.example.com
```

When using a dedicated user to bind, you can store the `LDAP_BIND_PASSWORD` setting in a Secret and reference it through the `ldapBindPassword` field:

```
userInterface:
  pgAdmin:
    config:
      ldapBindPassword:
        name: ldappass
        key: mypw
```

Deleting pgAdmin 4

You can remove the pgAdmin 4 deployment by removing the `userInterface` field from the spec.

Detailed Architecture

The goal of PGO, the Postgres Operator from Crunchy Data is to provide a means to quickly get your applications up and running on Postgres for both development and production environments. To understand how PGO does this, we want to give you a tour of its architecture, with explains both the architecture of the PostgreSQL Operator itself as well as recommended deployment models for PostgreSQL in production!

PGO Architecture

The Crunchy PostgreSQL Operator extends Kubernetes to provide a higher-level abstraction for rapid creation and management of PostgreSQL clusters. The Crunchy PostgreSQL Operator leverages a Kubernetes concept referred to as "[Custom Resources](#)" to create several [custom resource definitions \(CRDs\)](#) that allow for the management of PostgreSQL clusters.

The main custom resource definition is `postgresclusters.postgres-operator.crunchydata.com`. This allows you to control all the information about a Postgres cluster, including:

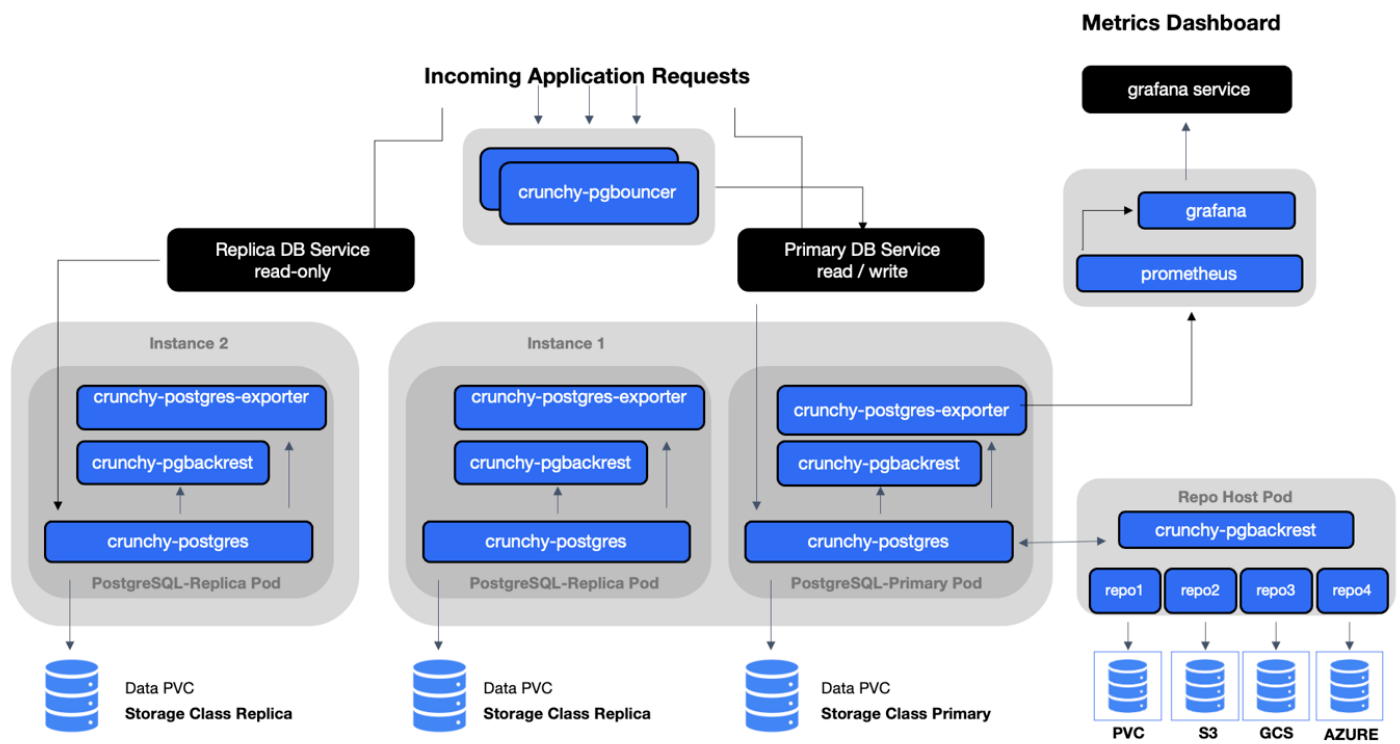
- General information
- Resource allocation
- High availability
- Backup management
- Where and how it is deployed (affinity, tolerations, topology spread constraints)
- Disaster Recovery / standby clusters
- Monitoring

and more.

PGO itself runs as a Deployment and is composed of a single container.

- **operator** (image: postgres-operator) - This is the heart of the PostgreSQL Operator. It contains a series of Kubernetes [controllers](#) that place watch events on a series of native Kubernetes resources (Jobs, Pods) as well as the Custom Resources that come with the PostgreSQL Operator (PostgresCluster, PGUpgrade)

The main purpose of PGO is to create and update information around the structure of a Postgres Cluster, and to relay information about the overall status and health of a PostgreSQL cluster. The goal is to also simplify this process as much as possible for users. For example, let's say we want to create a high-availability PostgreSQL cluster that has multiple replicas, supports having backups in both a local storage area and Amazon S3 and has built-in metrics and connection pooling, similar to:



This can be accomplished with a relatively simple manifest. Please refer to the tutorial for how to accomplish this, or see the [Postgres Operator examples](#) repo.

The Postgres Operator handles setting up all of the various StatefulSets, Deployments, Services and other Kubernetes objects.

You will also notice that **high-availability is enabled by default** if you deploy at least one Postgres replica. The Crunchy PostgreSQL Operator uses a distributed-consensus method for PostgreSQL cluster high-availability, and as such delegates the management of each cluster's availability to the clusters themselves. This removes the PostgreSQL Operator from being a single-point-of-failure, and has benefits such as faster recovery times for each PostgreSQL cluster. For a detailed discussion on high-availability, please see the High-Availability section.

Kubernetes StatefulSets: The PGO Deployment Model

PGO, the Postgres Operator from Crunchy Data, uses [Kubernetes StatefulSets](#) for running Postgres instances, and will use [Deployments](#) for more ephemeral services.

PGO deploys Kubernetes Statefulsets in a way to allow for creating both different Postgres instance groups and be able to support advanced operations such as rolling updates that minimize or eliminate Postgres downtime. Additional components in our PostgreSQL cluster, such as the pgBackRest repository or an optional PgBouncer, are deployed with Kubernetes Deployments.

With the PGO architecture, we can also leverage Statefulsets to apply affinity and toleration rules across every Postgres instance or individual ones. For instance, we may want to force one or more of our PostgreSQL replicas to run on Nodes in a different region than our primary PostgreSQL instances.

What's great about this is that PGO manages this for you so you don't have to worry! Being aware of this model can help you understand how the Postgres Operator gives you maximum flexibility for your PostgreSQL clusters while giving you the tools to troubleshoot issues in production.

The last piece of this model is the use of [Kubernetes Services](#) for accessing your PostgreSQL clusters and their various components. The PostgreSQL Operator puts services in front of each Deployment to ensure you have a known, consistent means of accessing your PostgreSQL components.

Note that in some production environments, there can be delays in accessing Services during transition events. The PostgreSQL Operator attempts to mitigate delays during critical operations (e.g. failover, restore, etc.) by directly accessing the Kubernetes Pods to perform given actions.

Additional Architecture Information

There is certainly a lot to unpack in the overall architecture of PGO. Understanding the architecture will help you to plan the deployment model that is best for your environment. For more information on the architectures of various components of the PostgreSQL Operator, please read onward!

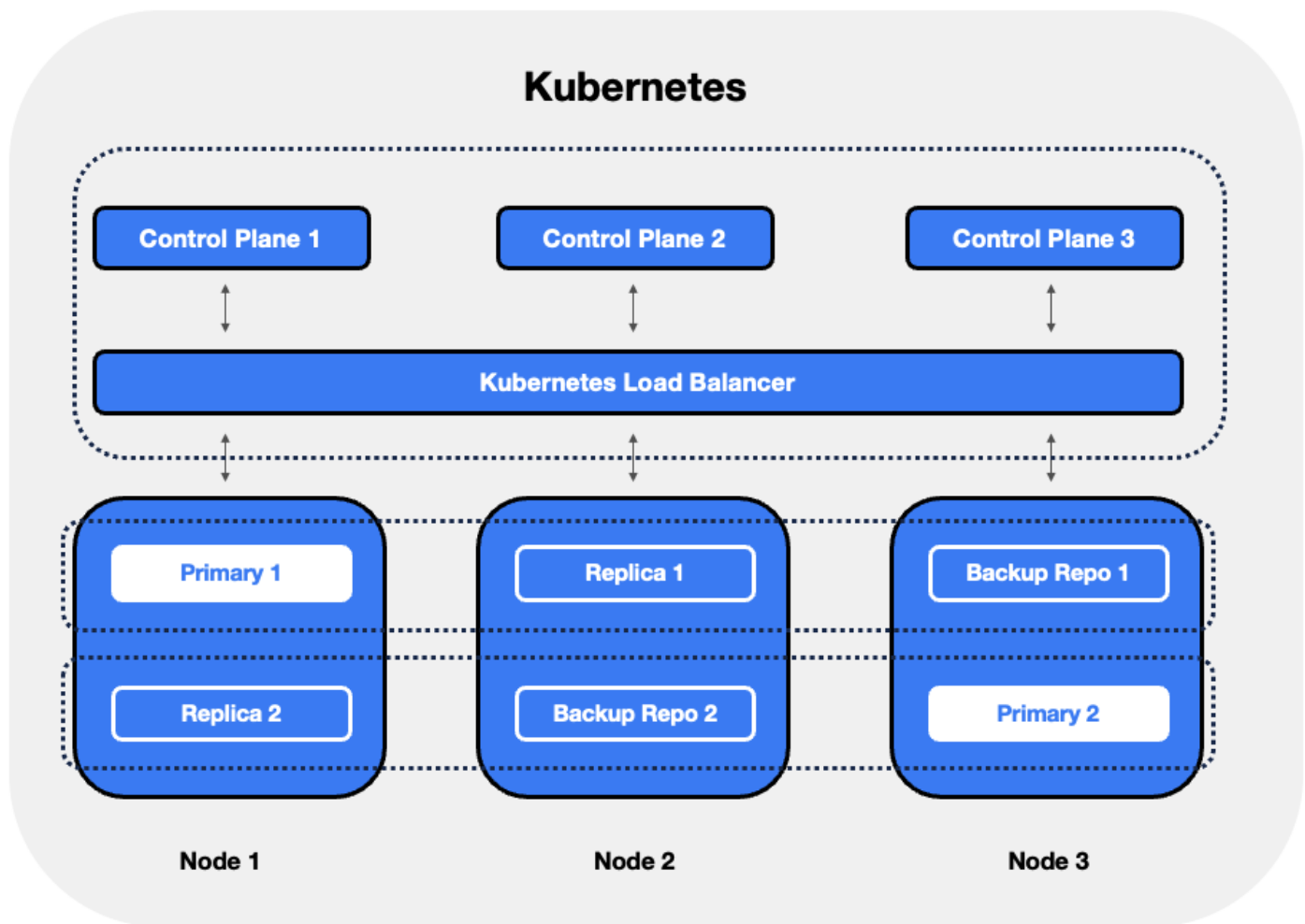
High Availability

One of the great things about PostgreSQL is its reliability: it is very stable and typically "just works." However, there are certain things that can happen in the environment that PostgreSQL is deployed in that can affect its uptime, including:

- The database storage disk fails or some other hardware failure occurs
- The network on which the database resides becomes unreachable
- The host operating system becomes unstable and crashes
- A key database file becomes corrupted
- A data center is lost

There may also be downtime events that are due to the normal case of operations, such as performing a minor upgrade, security patching of operating system, hardware upgrade, or other maintenance.

Fortunately, PGO, the Postgres Operator from Crunchy Data, is prepared for this.



The Crunchy PostgreSQL Operator supports a distributed-consensus based high availability (HA) system that keeps its managed PostgreSQL clusters up and running, even if the PostgreSQL Operator disappears. Additionally, it leverages Kubernetes specific features such as [Pod Anti-Affinity](#) to limit the surface area that could lead to a PostgreSQL cluster becoming unavailable. The PostgreSQL Operator also supports automatic healing of failed primaries and leverages the efficient pgBackRest "delta restore" method, which eliminates the need to fully reprovision a failed cluster!

The Crunchy PostgreSQL Operator also maintains high availability during a routine task such as a PostgreSQL minor version upgrade.

For workloads that are sensitive to transaction loss, PGO supports PostgreSQL synchronous replication.

The high availability backing for your PostgreSQL cluster is only as good as your high availability backing for Kubernetes. To learn more about creating a [high availability Kubernetes cluster](#), please review the [Kubernetes documentation](#) or consult your systems administrator.

The Crunchy Postgres Operator High Availability Algorithm

A critical aspect of any production-grade PostgreSQL deployment is a reliable and effective high availability (HA) solution. Organizations want to know that their PostgreSQL deployments can remain available despite various issues that have the potential to disrupt operations, including hardware failures, network outages, software errors, or even human mistakes.

The key portion of high availability that the PostgreSQL Operator provides is that it delegates the management of HA to the PostgreSQL clusters themselves. This ensures that the PostgreSQL Operator is not a single-point of failure for the availability of any of the PostgreSQL clusters that it manages, as the PostgreSQL Operator is only maintaining the definitions of what should be in the cluster (e.g. how many instances in the cluster, etc.).

Each HA PostgreSQL cluster maintains its availability by using Patroni to manage failover when the primary becomes compromised. Patroni stores the primary's ID in annotations on a Kubernetes **Endpoints** object which acts as a lease. The primary must periodically renew the lease to signal that it's healthy. If the primary misses its deadline, replicas compare their WAL positions to see who has the most up-to-date data. Instances with the latest data try to overwrite the ID on the lease. The first to succeed becomes the new primary, and all others follow the new primary.

How The Crunchy PostgreSQL Operator Uses Pod Anti-Affinity

Kubernetes has two types of Pod anti-affinity:

- Preferred: With preferred (**preferredDuringSchedulingIgnoredDuringExecution**) Pod anti-affinity, Kubernetes will make a best effort to schedule Pods matching the anti-affinity rules to different Nodes. However, if it is not possible to do so, then Kubernetes may schedule one or more Pods to the same Node.
- Required: With required (**requiredDuringSchedulingIgnoredDuringExecution**) Pod anti-affinity, Kubernetes mandates that each Pod matching the anti-affinity rules **must** be scheduled to different Nodes. However, a Pod may not be scheduled if Kubernetes cannot find a Node that does not contain a Pod matching the rules.

There is a tradeoff with these two types of pod anti-affinity: while "required" anti-affinity will ensure that all the matching Pods are scheduled on different Nodes, if Kubernetes cannot find an available Node, your Postgres instance may not be scheduled. Likewise, while "preferred" anti-affinity will make a best effort to scheduled your Pods on different Nodes, Kubernetes may compromise and schedule more than one Postgres instance of the same cluster on the same Node.

By understanding these tradeoffs, the makeup of your Kubernetes cluster, and your requirements, you can choose the method that makes the most sense for your Postgres deployment. We'll show examples of both methods below!

For an example for how pod anti-affinity works with PGO, please see the high availability tutorial.

Synchronous Replication: Guarding Against Transaction Loss

Clusters managed by the Crunchy PostgreSQL Operator can be deployed with synchronous replication, which is useful for workloads that are sensitive to losing transactions, as PostgreSQL will not consider a transaction to be committed until it is committed to all synchronous replicas connected to a primary. This provides a higher guarantee of data consistency and, when a healthy synchronous replica is present, a guarantee of the most up-to-date data during a failover event.

This comes at a cost of performance: PostgreSQL has to wait for a transaction to be committed on all synchronous replicas, and a connected client will have to wait longer than if the transaction only had to be committed on the primary (which is how asynchronous replication works). Additionally, there is a potential impact to availability: if a synchronous replica crashes, any writes to the primary will be blocked until a replica is promoted to become a new synchronous replica of the primary.

Node Affinity

Kubernetes [Node Affinity](#) can be used to schedule Pods to specific Nodes within a Kubernetes cluster. This can be useful when you want your PostgreSQL instances to take advantage of specific hardware (e.g. for geospatial applications) or if you want to have a replica instance deployed to a specific region within your Kubernetes cluster for high availability purposes.

For an example for how node affinity works with PGO, please see the high availability tutorial.

Tolerations

Kubernetes [Tolerations](#) can help with the scheduling of Pods to appropriate nodes. There are many reasons that a Kubernetes administrator may want to use tolerations, such as restricting the types of Pods that can be assigned to particular Nodes. Reasoning and strategy for using taints and tolerations is outside the scope of this documentation.

You can configure the tolerations for your Postgres instances on the `postgresclusters` custom resource.

Pod Topology Spread Constraints

Kubernetes [Pod Topology Spread Constraints](#) can also help you efficiently schedule your workloads by ensuring your Pods are not scheduled in only one portion of your Kubernetes cluster. By spreading your Pods across your Kubernetes cluster among your various failure-domains, such as regions, zones, nodes, and other user-defined topology domains, you can achieve high availability as well as efficient resource utilization.

For an example of how pod topology spread constraints work with PGO, please see the high availability tutorial.

Rolling Updates

Some changes to a running PostgreSQL cluster require a planned restart. Various PostgreSQL settings must be set "at server start," for example, like [shared buffers](#). Restarts can be disruptive in a high availability deployment, which is why many systems employ a ["rolling update" strategy](#) (a.k.a. a "rolling restart") to minimize or eliminate downtime.

The simple update strategies provided by Kubernetes do not work for stateful applications like PostgreSQL. Instead, the PostgreSQL Operator employs the following algorithm to ensure the cluster can accept reads and writes except for the short time it takes to perform a single switchover:

- Each replica is updated in turn as follows:
 - The replica is explicitly shut down to flush any outstanding changes to its disk.
 - If requested, the PostgreSQL Operator will apply any changes to the Pod.
 - The replica is brought back online. The PostgreSQL Operator waits for the replica to become available before it proceeds to the next replica.
- The above steps are repeated until all replicas are up-to-date.
- A controlled switchover is performed. The replicas collectively choose a new primary, and the former primary shuts down and follows a process similar to step 1.

PGO automatically detects when to apply a rolling update.

Pod Disruption Budgets

Pods in a Kubernetes cluster can experience [voluntary disruptions](#) as a result of actions initiated by the application owner or a Cluster Administrator. During these voluntary disruptions Pod Disruption Budgets (PDBs) can be used to ensure that

a minimum number of Pods will be running. The operator allows you to define a minimum number of Pods that should be available for instance sets and PgBouncer deployments in your postgrescluster. This minimum is configured in the postgrescluster spec and will be used to create PDBs associated to a resource defined in the spec. For example, the following spec will create two PDBs, one for **instance1** and one for the PgBouncer deployment:

```
spec:
  instances:
    - name: instance1
      replicas: 3
      minAvailable: 1
  proxy:
    pgBouncer:
      replicas: 3
      minAvailable: 1
```

Hint

The **minAvailable** field accepts number (**3**) or string percentage (**50%**) values.

For more information see [Specifying a PodDisruptionBudget](#).

If **minAvailable** is set to **0**, we will not reconcile a PDB for the resource and any existing PDBs will be removed. This will effectively disable Pod Disruption Budgets for the resource.

If **minAvailable** is not provided for an object, a default value will be defined based on the number of replicas defined for that object. If there is one replica, a PDB will not be created. If there is more than one replica defined, a minimum of one Pod will be used.

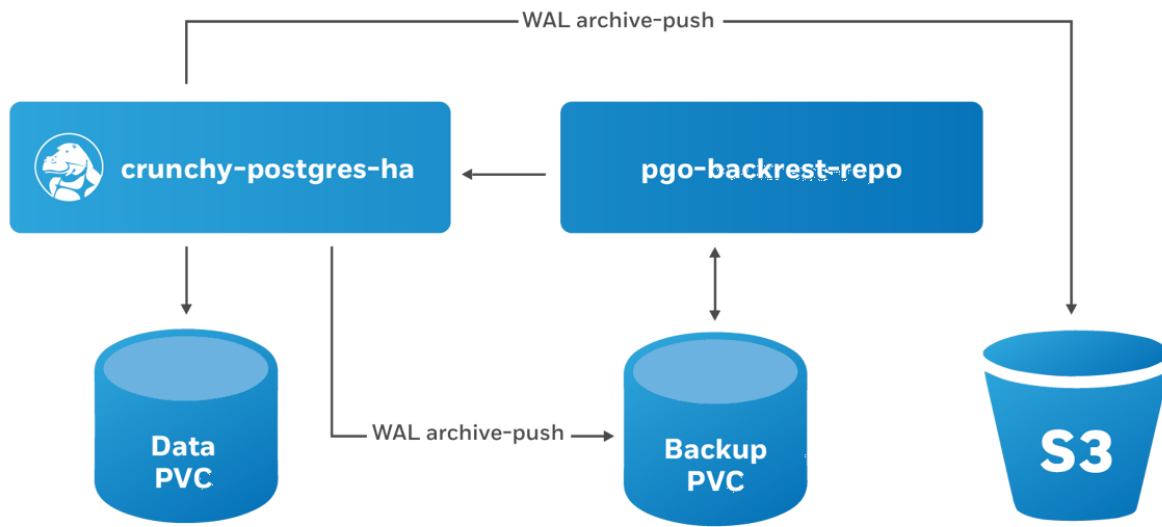
Backup Management

When using the PostgreSQL Operator, the answer to the question "do you take backups of your database" is automatically "yes!"

The PostgreSQL Operator uses the open source [pgBackRest](#) backup and restore utility that is designed for working with databases that are many terabytes in size. As described in the tutorial, pgBackRest is enabled by default as it permits the PostgreSQL Operator to automate some advanced as well as convenient behaviors, including:

- Efficient provisioning of new replicas that are added to the PostgreSQL cluster
- Preventing replicas from falling out of sync from the PostgreSQL primary by allowing them to replay old WAL logs
- Allowing failed primaries to automatically and efficiently heal using the "delta restore" feature
- Serving as the basis for the cluster cloning feature
- ...and of course, allowing for one to take full, differential, and incremental backups and perform full and point-in-time restores

Below is one example of how PGO manages backups with local storage and an Amazon S3 configuration.



The PostgreSQL Operator leverages a pgBackRest repository to facilitate the usage of the pgBackRest features in a PostgreSQL cluster. When a new PostgreSQL cluster is created, it simultaneously creates a pgBackRest repository.

You can store your pgBackRest backups in up to four different locations and using four different storage types:

- Any Kubernetes storage class
- Amazon S3 (or S3 equivalents like MinIO)
- Google Cloud Storage (GCS)
- Azure Blob Storage

PostgreSQL is automatically configured to use the **pgbackrest archive-push** command to archive the write-ahead log (WAL) in all repositories.

Backups

PGO supports three types of pgBackRest backups:

- Full: A full backup of all the contents of the PostgreSQL cluster
- Differential: A backup of only the files that have changed since the last full backup
- Incremental: A backup of only the files that have changed since the last full, differential, or incremental backup

Scheduling Backups

Any effective disaster recovery strategy includes having regularly scheduled backups. PGO enables this by managing a series of Kubernetes CronJobs to ensure that backups are executed at scheduled times.

Note that pgBackRest presently only supports taking one backup at a time. This may change in a future release, but for the time being we suggest that you stagger your backup times.

Please see the backup management tutorial for how to set up backup schedules and configure retention policies.

Restores

The PostgreSQL Operator can perform a full restore on a PostgreSQL cluster or a point-in-time recovery. There are also two ways to restore a cluster:

- Restore to a new cluster
- Restore in-place

For examples of this, please see the disaster recovery tutorial

Deleting a Backup

Warning

If you delete a backup that is *not* set to expire, you may be unable to meet your retention requirements. If you are deleting backups to free space, you should delete your oldest backup first.

A backup can be deleted by running the `pgbackrest expire` command directly on the pgBackRest repository Pod or a Postgres instance.

Scheduling

Deploying to your Kubernetes cluster may allow for greater reliability than other environments, but that's only the case when it's configured correctly. Fortunately, PGO, the Postgres Operator from Crunchy Data, is ready to help with helpful default settings to ensure you make the most out of your Kubernetes environment!

High Availability By Default

As shown in the high availability tutorial, PGO supports the use of [Pod Topology Spread Constraints](#) to customize your Pod deployment strategy, but useful defaults are already in place for you without any additional configuration required!

PGO's default scheduling constraints for HA is implemented for the various Pods comprising a PostgreSQL cluster, specifically to ensure the Operator always deploys a High-Availability cluster architecture by default.

Using Pod Topology Spread Constraints, the general scheduling guidelines are as follows:

- Pods are only considered from the same cluster.
- PgBouncer pods are only considered amongst other PgBouncer pods.
- Postgres pods are considered amongst all Postgres pods and pgBackRest repo host Pods.
- pgBackRest repo host Pods are considered amongst all Postgres pods and pgBackRest repo hosts Pods.

- Pods are scheduled across the different `kubernetes.io/hostname` and `topology.kubernetes.io/zone` failure domains.
- Pods are scheduled when there are fewer nodes than pods, e.g. single node.

With the above configuration, your data is distributed as widely as possible throughout your Kubernetes cluster to maximize safety.

Customization

While the default scheduling settings are designed to meet the widest variety of environments, they can be customized or removed as needed. Assuming a PostgresCluster named 'hippo', the default Pod Topology Spread Constraints applied on Postgres Instance and pgBackRest Repo Host Pods are as follows:

```
topologySpreadConstraints:
- maxSkew: 1
  topologyKey: kubernetes.io/hostname
  whenUnsatisfiable: ScheduleAnyway
  labelSelector:
    matchLabels:
      postgres-operator.crunchydata.com/cluster: hippo
    matchExpressions:
      - key: postgres-operator.crunchydata.com/data
        operator: In
        values:
          - postgres
          - pgbackrest
- maxSkew: 1
  topologyKey: topology.kubernetes.io/zone
  whenUnsatisfiable: ScheduleAnyway
  labelSelector:
    matchLabels:
      postgres-operator.crunchydata.com/cluster: hippo
    matchExpressions:
      - key: postgres-operator.crunchydata.com/data
        operator: In
        values:
          - postgres
          - pgbackrest
```

Similarly, for PgBouncer Pods they will be:

```
topologySpreadConstraints:
- maxSkew: 1
  topologyKey: kubernetes.io/hostname
  whenUnsatisfiable: ScheduleAnyway
  labelSelector:
    matchLabels:
      postgres-operator.crunchydata.com/cluster: hippo
      postgres-operator.crunchydata.com/role: pgbouncer
- maxSkew: 1
  topologyKey: topology.kubernetes.io/zone
  whenUnsatisfiable: ScheduleAnyway
  labelSelector:
    matchLabels:
      postgres-operator.crunchydata.com/cluster: hippo
      postgres-operator.crunchydata.com/role: pgbouncer
```

Which, as described in the [API documentation](#), means that there should be a maximum of one Pod difference within the `kubernetes.io/hostname` and `topology.kubernetes.io/zone` failure domains when considering either `data` Pods, i.e. Postgres Instance or pgBackRest repo host Pods from a single PostgresCluster or when considering PgBouncer Pods from a single PostgresCluster.

Any other scheduling configuration settings, such as [Affinity](#), [Anti-affinity](#), [Taints](#), [Tolerations](#), or other [Pod Topology Spread Constraints](#) will be added in addition to these defaults. Care should be taken to ensure the combined effect of these settings are appropriate for your Kubernetes cluster.

In cases where these defaults are not desired, PGO does provide a method to disable the default Pod scheduling by setting the `spec.disableDefaultPodScheduling` to 'true'.

User Management

PGO manages PostgreSQL users that you define in `PostgresCluster.spec.users`. There, you can list their [role attributes](#) and which databases they can access.

Below is some information on how the user and database management systems work. To try out some examples, please see the user and database management section of the tutorial.

Understanding Default User Management

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

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

You can see this default behavior in the connect to a cluster portion of the tutorial.

As an example, using our `hippo` Postgres cluster, we would see the following created:

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

While the above defaults may work for your application, there are certain cases where you may need to customize your user and databases:

- You may require access to the **postgres** superuser.
- You may need to define privileges for your users.
- You may need multiple databases in your cluster, e.g. in a multi-tenant application.
- Certain users may only be able to access certain databases.

Custom Users and Databases

Users and databases can be customized in the **spec.users** section of the custom resource. These can be added during cluster creation and adjusted over time, but it's important to note the following:

- If **spec.users** is set during cluster creation, PGO will **not** create any default users or databases except for **postgres**. If you want additional databases, you will need to specify them.
- If **spec.users** is set to an empty list, then PGO will skip creating any users or databases.
- For any users added in **spec.users**, PGO will create a Secret of the format **<clusterName>-pguser-<userName>**. This will contain the user credentials.
- If no databases are specified, **dbname** and **uri** will not be present in the Secret.
- If at least one **spec.users.databases** is specified, the first database in the list will be populated into the connection credentials.
- To prevent accidental data loss, PGO does not automatically drop users. We will see how to drop a user below.
- Similarly, to prevent accidental data loss PGO does not automatically drop databases. We will see how to drop a database below.
- Role attributes are not automatically dropped if you remove them. You will have to set the inverse attribute to drop them (e.g. **NOSUPERUSER**).
- The special **postgres** user can be added as one of the custom users; however, its privileges cannot be adjusted.

For specific examples of how to manage users, please see the user and database management tutorial.

Generated Passwords

PGO generates a random password for each Postgres user it creates. Postgres allows almost any character in its passwords, but your application may have stricter requirements. To have PGO generate a password without special characters, set the **spec.users.password.type** field for that user to **AlphaNumeric**. For complete control over a user's password, see the [custom passwords](#) section.

To have PGO generate a new password, remove the existing **password** field from the user *Secret*. For example, on a Postgres cluster named **hippo** in the **postgres-operator** namespace with a Postgres user named **hippo**, use the following **kubectl patch** command:

Bash:

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

Powershell:

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

Custom Passwords

There are cases where you may want to explicitly provide your own password for a Postgres user. PGO determines the password from an attribute in the user Secret called **verifier**. This contains a hashed copy of your password. When **verifier** changes, PGO will load the contents of the verifier into your Postgres cluster. This method allows for the secure transmission of the password into the Postgres database.

Postgres provides two methods for hashing passwords: SCRAM-SHA-256 and MD5. PGO uses the preferred (and as of PostgreSQL 14, default) method, SCRAM-SHA-256.

There are two ways you can set a custom password for a user. You can provide a plaintext password in the **password** field and remove the **verifier**. When PGO detects a password without a verifier it will generate the SCRAM **verifier** for you. Optionally, you can generate your own password and verifier. When both values are found in the user secret PGO will not generate anything. Once the password and verifier are found PGO will ensure the provided credential is properly set in Postgres.

Example

For example, let's say we have a Postgres cluster named **hippo** and a Postgres user named **hippo**. The Secret then would be called **hippo-pguser-hippo**. We want to set the password for **hippo** to be **datalake** and we can achieve this with a simple **kubectl patch** command. The below assumes that the Secret is stored in the **postgres-operator** namespace:

Bash:

```
kubectl patch secret -n postgres-operator hippo-pguser-hippo -p '{"stringData":{"password":"datalake","verifier":""}}'
```

Powershell:

```
kubectl patch secret -n postgres-operator hippo-pguser-hippo -p '{"stringData":{"password":"datalake","verifier":""}}'
```

Hint

We can take advantage of the [Kubernetes Secret](#) **stringData** field to specify non-binary secret data in string form.

PGO generates the SCRAM verifier and applies the updated password to Postgres, and you will be able to log in with the password **datalake**.

Database Observability

Crunchy Postgres for Kubernetes (CPK) ensures your Postgres cluster deployments are fully observable, allowing you to easily view and analyze log and metric data for your Postgres databases, as well as any other components deployed alongside your Postgres database (pgBackRest, PgBouncer, pgAdmin and more). By leveraging the OpenTelemetry

framework

and standard, CPK seamlessly collects and exposes logs and metrics in a clean and consistent way that is interoperable with a variety of different observability backends. This means you can leverage a large ecosystem of different OpenTelemetry-compatible services, backends, and tooling to store, search, manage, and monitor any log or metric data generated by your Postgres databases.

Additionally, you can seamlessly collect logging and metric data across all of your Postgres cluster deployments (which may span across multiple Kubernetes clusters, data centers and regions) in a consistent and centralized manner, greatly enhancing your ability to gain deeper insights into Postgres cluster deployments. This streamlines your ability to monitor the overall health of your various Postgres cluster deployments, while also greatly enhancing your ability to troubleshoot any issues that might occur, and answer questions about certain behavior and activities that occur within your database cluster.

Observability Overview

Observability is the ability to analyze, measure, and better understand the internal state of a system using the external outputs (i.e., the telemetry data) provided by that system. These outputs come in a variety of different forms, including:

- **Logs** - A timestamped record or file that captures information about specific activities, changes or errors within a system.
- **Metrics** - A measurement of a service captured at runtime used to identify system performance, availability, and/or reliability.
- **Traces** - A recorded sequence of events that allows you to understand the full path of a request to a system as it traverses various services and components.

When a system is observable, system administrators, analysts, and engineers can easily answer questions around why the system behaved or responded in a certain way, without requiring detailed or direct knowledge about the internal workings of that system. This is the primary goal of the observability capabilities built into CPK: to ensure you can easily answer questions about the functionality and health of your database clusters, without requiring deep knowledge of the internal workings of each component comprising your Postgres cluster.

Observability For Databases

When running a database such as Postgres, you want to be able to closely monitor and analyze the key attributes of the system such as its overall health and performance, while also answering questions about who is accessing the database and how it is being accessed. Additionally, you want to be able to easily troubleshoot any issues that might occur, while also easily identifying the root cause for those issues.

Fortunately, Postgres creates a variety of different external outputs that can be leveraged to ensure the database is observable. This includes rich sets of logs (which can be further enhanced with various Postgres extensions), as well as key metric information that can be obtained by querying system tables within the database, and by looking at pertinent data within the environment and operating system the database is running within. For instance, CPU and memory usage can be obtained by analyzing cgroup v2 information for a container-based deployment of Postgres within Kubernetes. The same is true for the various components deployed alongside of your Postgres database, such as those that provide High Availability, Disaster Recovery, Connection Pooling, and more, all of which also provide a rich set of observable outputs.

Crunchy Postgres for Kubernetes therefore leverages these outputs to ensure all of the Postgres databases within Kubernetes are fully observable, equipping you with the tools to seamlessly monitor and analyze key attributes of any database cluster within your environment in real time. This puts you in the position of being able to answer questions about the database performance and functionality, equipping you with the critical information needed to ensure your database deployments are properly tuned and configured to ensure your applications and users get the most of out of all your Postgres cluster deployments.

Observability In Kubernetes

As a cloud-native technology, Kubernetes requires a solution for observability that is able to handle the diverse application and system deployments that exist across complex and distributed cloud architectures. This includes a solution that is vendor agnostic, and provides a consistent framework and standards for collecting, processing, and exposing telemetry data. Fortunately, the OpenTelemetry framework was designed from the ground-up to provide a cloud-native approach to observability, making OpenTelemetry a perfect fit for enabling observability across all applications and systems within a Kubernetes environment.

OpenTelemetry Overview

OpenTelemetry is an open-source observability framework that is used to collect, analyze, and export telemetry data (logs, metrics, and traces) from a system in a consistent and standardized manner, that is both tool and vendor agnostic. OpenTelemetry therefore plays a key role in allowing you to better understand the behavior of your systems by making it easier to capture and transfer telemetry data in a standard and consistent manner. And because the OpenTelemetry standard is vendor and tool agnostic, you can easily send your telemetry data to a variety of different OpenTelemetry compliant services or backends without requiring any changes to how that data is created, collected, or exported. This means you can easily plug into the various observability backends to meet your observability needs, while also avoiding vendor lock-in or any costly changes to your telemetry implementation when you want to change backends and/or leverage new services.

The primary tool used to collect and process OpenTelemetry data is known as the OpenTelemetry collector. The OpenTelemetry collector is responsible for receiving telemetry data (e.g., logs and metrics) from various applications and services; the OpenTelemetry collector then transforms, filters, and modifies that data (e.g., according to OpenTelemetry conventions and the OpenTelemetry logging model). The collector then exports that data to a variety of different OpenTelemetry-compatible backends and services:

For detailed information about OpenTelemetry and the OpenTelemetry collector, please see the [OpenTelemetry Documentation](#).

Observability & OpenTelemetry in CPK

By leveraging OpenTelemetry standards and tooling, CPK seamlessly collects metrics and logging data by attaching OpenTelemetry collector sidecars to all of the components comprising your Postgres cluster. For instance, not only is telemetry data collected and exported for your Postgres databases, it is also collected and exported for the High-Availability, Disaster Recovery, Connection Pooling, and User Interface components comprising your cluster. And since CPK does all of the heavy lifting to configure those components for metrics collecting, while also properly

formatting those logs and metrics according to the OpenTelemetry conventions and standards, you can simply focus on deciding what OpenTelemetry-compatible service and tools you want to use to view and analyze telemetry data for your Postgres clusters, all via a simple YAML configuration within your PostgresCluster spec.

OpenTelemetry Logging in CPK

When OpenTelemetry logging is enabled, CPK automatically handles the setup and configuration needed to ensure all of the components comprising your Postgres cluster export pertinent logging information to a variety of different OpenTelemetry-compatible logging services and backends. CPK monitors and captures those logs using the OpenTelemetry

collector, and transforms them according to the OpenTelemetry log data model. This results in a consistent set of logs across each of the components comprising your full Postgres cluster deployment. From there, your logs can be exported to variety of different OpenTelemetry-compatible logging backends, based on the configuration you provide in your PostgresCluster spec.

For instance, to export your logs to Google Cloud, your spec would include an **instrumentation** section in your **PostgresCluster** spec similar to the following:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: otel-hippo
spec:
  instrumentation:
    config:
      detectors:
        - name: gcp
      exporters:
        googlecloud:
          log:
            default_log_name: "collector-exported-log"
            resource_filters:
              - prefix: "k8s"
              - prefix: "db"
  logs:
    exporters: ['googlecloud']
```

This means you can simply focus on where you want to send your logs, while CPK seamlessly and automatically handles everything else (e.g., capturing, processing, and transforming of your logs).

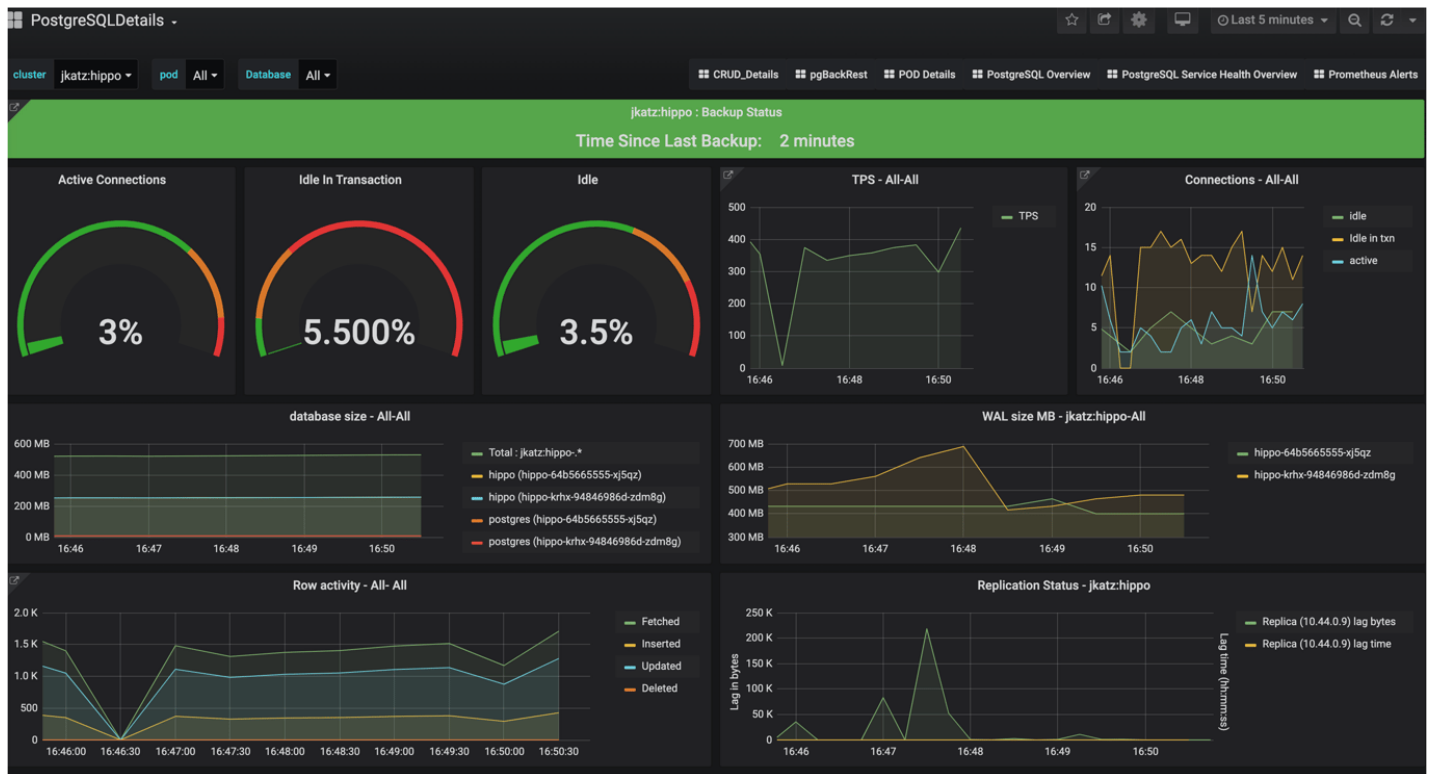
The various types of logs that are exported from your Postgres cluster using OpenTelemetry include:

- **Database Logs** - Logs from the Postgres database and the pgAudit extension
- **High Availability Logs** - Logs from Patroni, which is responsible for keeping your Postgres clusters highly available
- **Disaster Recovery Logs** - Logs produced by pgBackRest when backing-up and restoring your databases
- **Connection Pooling Logs** - Logs produced by PgBouncer when connection pooling is enabled within a Postgres cluster
- **User Interface Logs** - Logs produced by pgAdmin when a Postgres user interface is deployed to manage one or more Postgres clusters

OpenTelemetry Metrics in CPK

When OpenTelemetry metrics are enabled, CPK automatically starts collecting metrics across the various components comprising your Postgres cluster. For a detailed overview of the metrics collected via OpenTelemetry, as well as the Grafana dashboards included in CPK for viewing those metrics, please see the Monitoring section of the documentation. For details on configuring OpenTelemetry metrics, such as how to add your own custom metrics, see the OpenTelemetry Metrics guide.

Monitoring



High availability, backups, and disaster recovery systems help when something goes wrong with your PostgreSQL cluster. Monitoring helps you anticipate problems before they happen. Additionally, monitoring can help you diagnose and resolve issues that degrade performance.

There are many different ways to monitor systems within Kubernetes, including tools that come with Kubernetes itself. Here we review what Crunchy Postgres for Kubernetes provides for an out-of-the-box monitoring solution.

Getting Started

If you want to install the metrics stack, please visit the installation instructions for the PostgreSQL Operator Monitoring stack.

Components

The PostgreSQL Operator Monitoring stack is made up of several open source components:

- [Prometheus](#), a time-series database that scrapes and stores the collected metrics so they can be consumed by other services.
- [Grafana](#), a visualization tool that provides charting and other capabilities for viewing the collected monitoring data. The standard CPK monitoring stack includes several pre-built dashboards from [pgMonitor](#).
- [Alertmanager](#), a tool that can send alerts when metrics hit a certain threshold that require someone to intervene.
- [pgnodemx](#), a PostgreSQL extension that is able to pull container-specific metrics (e.g. CPU utilization, memory consumption) from the container itself via SQL queries.

- [controller-runtime](#), a set of Go libraries used to build the controllers in the **postgres-operator** that includes a [metrics server package](#).

In versions before CPK v5.8.0, this stack included [postgres_exporter](#). **postgres_exporter** both provided queries used to collect metrics information about a PostgreSQL instance, as well as serving as the mechanism to run queries defined by [pgMonitor](#) or through custom queries.

Starting from CPK v5.8.0, CPK now offers a choice of mechanisms for querying and exporting metrics from Postgres instances. While **postgres_exporter** is still an option, users can enable the **OpenTelemetryMetrics** feature gate for individual clusters. If you are using OpenTelemetry Metrics, then instead of **postgres_exporter** providing metric queries, CPK is managing those queries directly and using an OpenTelemetry SQL query library to expose those metrics. For more on information on the OpenTelemetry Architecture, see our Database Observability page and our guide to OpenTelemetry metrics.

PGO Metrics

Starting in CPK v5.8.0, the metrics endpoint provided by **controller-runtime** is exposed on the **postgres-operator** Pod and is secure by default, using https to encrypt traffic and Kubernetes authentication and authorization to ensure only service accounts with proper RBAC permissions can scrape the endpoint. The [provided metrics](#) can give you insight into the behavior and performance of the different controllers in the **postgres-operator**.

Warning

The certificates used for https are self-signed certificates generated by **controller-runtime**. If you wish to provide your own certificates, see the section below.

Installing Custom Certificates for the PGO Metrics endpoint

To provide your custom certificates, they will need to be placed in a Secret, and the Secret will need to be created in the same Namespace as the **postgres-operator**. It should contain the TLS key (**tls.key**) and TLS certificate (**tls.crt**) needed to enable encryption, and they should be named accordingly in the Secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: metrics-certs
data:
  tls.crt: $VALUE
  tls.key: $VALUE
```

With the Secret in place, you need to adjust your **postgres-operator** Deployment so that you can mount the certificates from the Secret into a Volume for the operator's metrics server to use. This entails adding a Volume and a VolumeMount as seen in the example below:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pgo
spec:
  replicas: 1
```

```

strategy: { type: Recreate }
template:
  spec:
    volumes:
      - name: metrics-config
        secret:
          secretName: metrics-certs
    serviceAccountName: pgo
    containers:
      - name: operator
        image: postgres-operator
        volumeMounts:
          - name: metrics-config
            mountPath: /tmp/k8s-metrics-server/serving-certs

```

After you configure the certificates for the **controller-runtime** metrics endpoint, you will need to update your Prometheus deployment to use these certificates, and your connection to the exporter will be encrypted. Check out the [Prometheus](#) documentation for more information on configuring TLS for [Prometheus](#).

If the certificates are properly signed and the Prometheus configuration correct, you should be able to turn off the **insecure_skip_verify** setting in the Prometheus configuration, which can be found in the **prometheus/config/prometheus.yml** file in the CPK Monitoring installer.

The CPK Monitoring installer and CPK Operator metrics

The most recent CPK Monitoring installer includes changes to automatically scrape CPK operator metrics.

If you are not seeing these metrics in the Prometheus or Grafana setup by the CPK Monitoring installer, you could double check that the Prometheus configuration includes a **pgo-metrics** scrape job. That job is configured to discover CPK Operator pods and scrape the metrics endpoint.

If you are missing that job, you may need to download a newer version of the CPK Monitoring installer and install that. This advice goes as well for users who were using CPK less than v5.8.0 and altered their monitoring stack to handle that case.

pgnodemx and the DownwardAPI

pgnodemx is able to pull and format container-specific metrics by accessing several Kubernetes fields that are mounted from the pod to the **database** container's filesystem. By default, these fields include the pod's labels and annotations, as well as the **database** pod's CPU and memory. These fields are mounted at the **/etc/database-containerinfo** path.

Visualizations

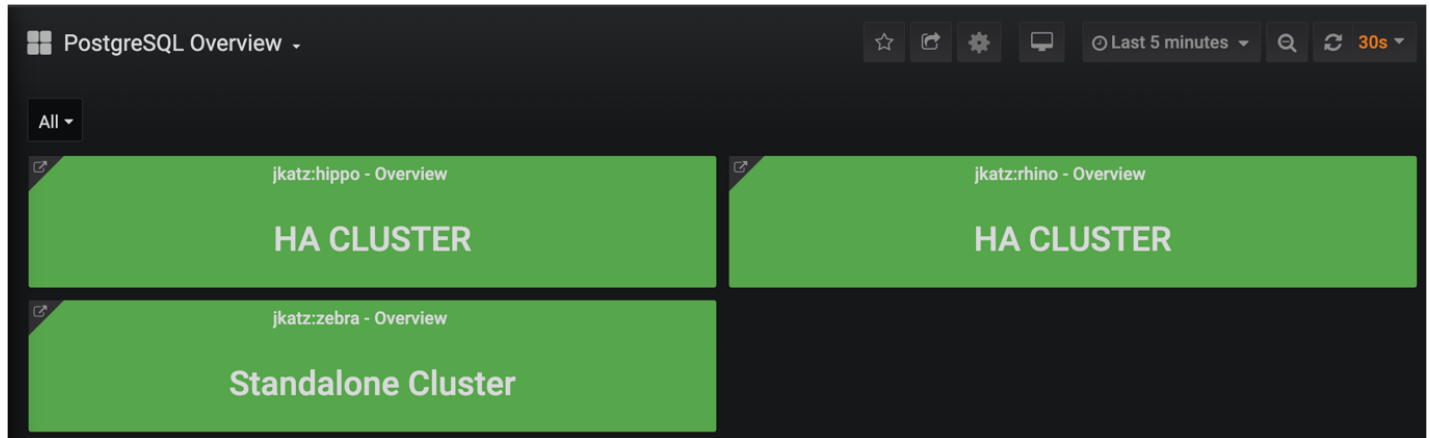
Below is a brief description of all the visualizations provided by the PostgreSQL Operator Monitoring stack. Some of the descriptions may include some directional guidance on how to interpret the charts, though this is only to provide a starting point: actual causes and effects of issues can vary between systems.

Many of the visualizations can be broken down based on the following groupings:

- Cluster: which PostgreSQL cluster should be viewed

- Pod: the specific Pod or PostgreSQL instance

Overview

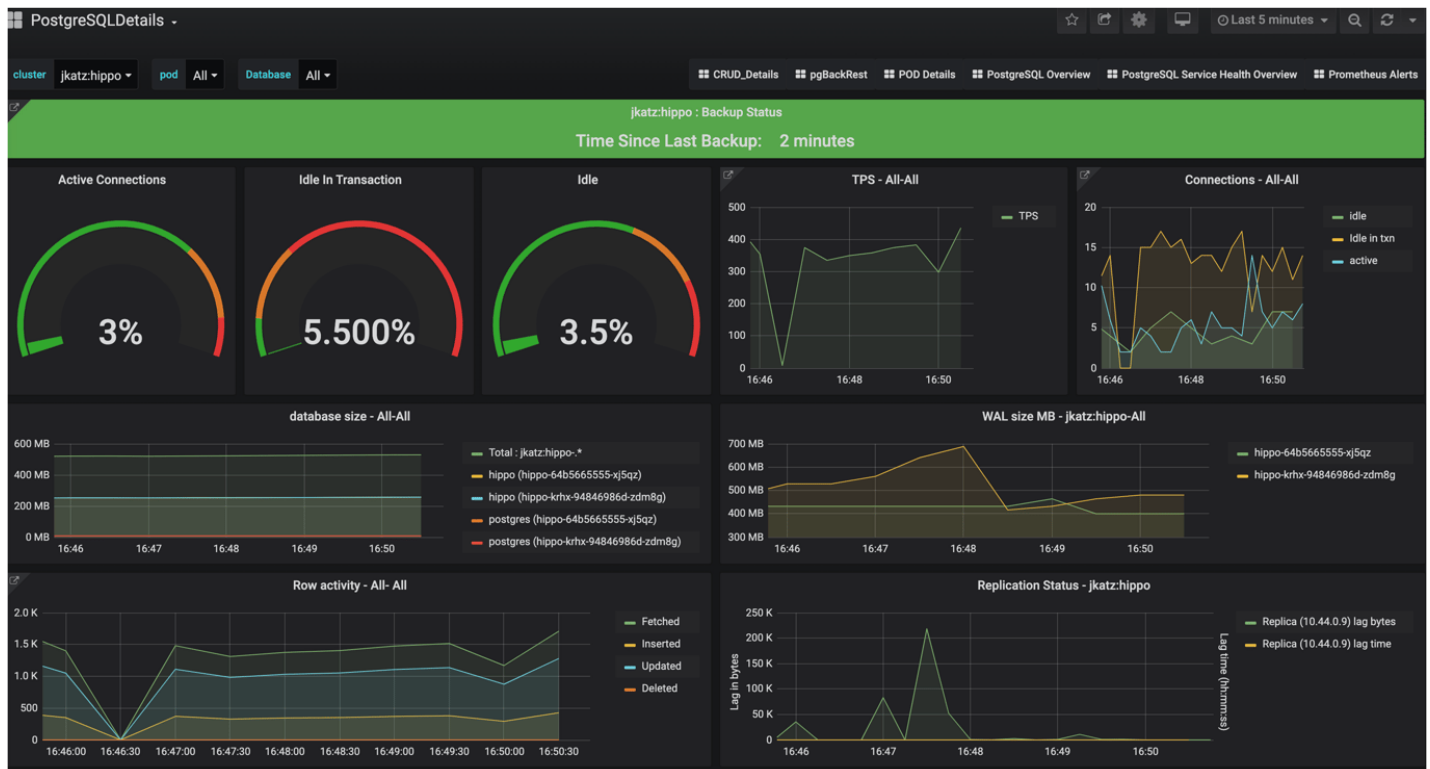


The overview provides an overview of all of the PostgreSQL clusters that are being monitoring by the PostgreSQL Operator Monitoring stack. This includes the following information:

- The name of the PostgreSQL cluster and the namespace that it is in
- The type of PostgreSQL cluster (HA [high availability] or standalone)
- The status of the cluster, as indicate by color. Green indicates the cluster is available, red indicates that it is not.

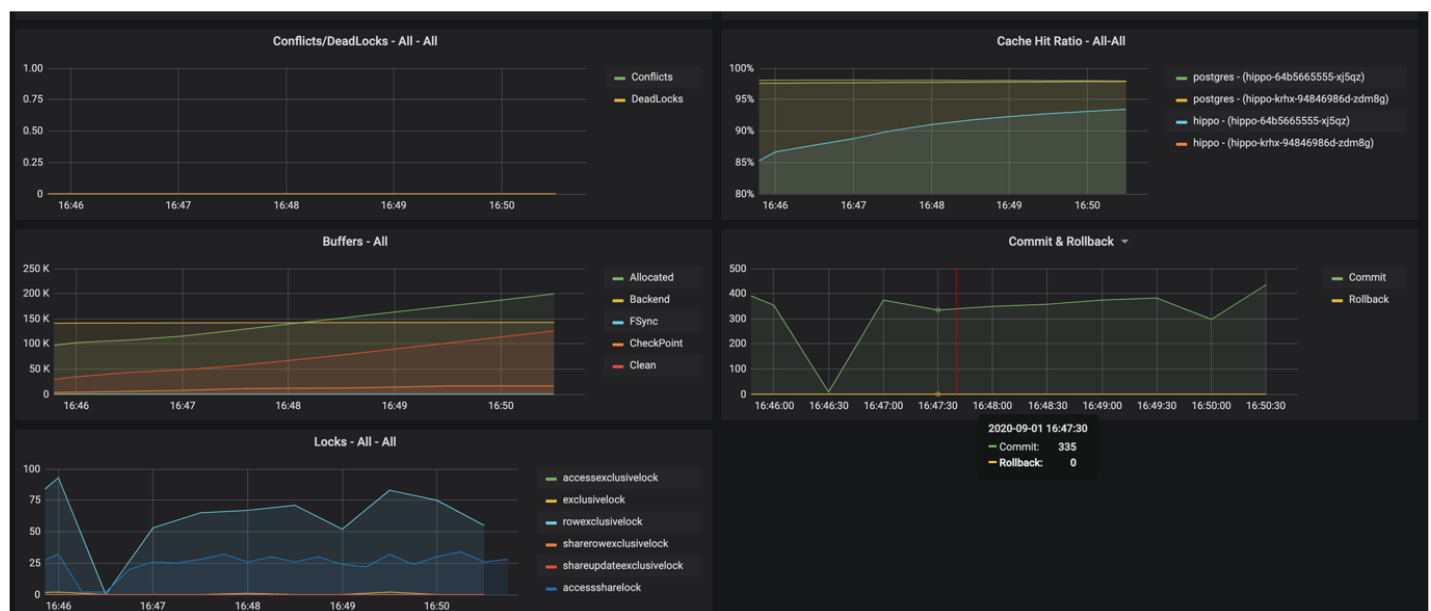
Each entry is clickable to provide additional cluster details.

PostgreSQL Details



The PostgreSQL Details view provides more information about a specific PostgreSQL cluster that is being managed and monitored by the PostgreSQL Operator. These include many key PostgreSQL-specific metrics that help make decisions around managing a PostgreSQL cluster. These include:

- **Backup Status:** The last time a backup was taken of the cluster. Green is good. Orange means that a backup has not been taken in more than a day and may warrant investigation.
- **Active Connections:** How many clients are connected to the database. Too many clients connected could impact performance and, for values approaching 100%, can lead to clients being unable to connect.
- **Idle in Transaction:** How many clients have a connection state of "idle in transaction". Too many clients in this state can cause performance issues and, in certain cases, maintenance issues.
- **Idle:** How many clients are connected but are in an "idle" state.
- **TPS:** The number of "transactions per second" that are occurring. Usually needs to be combined with another metric to help with analysis. "Higher is better" when performing benchmarking.
- **Connections:** An aggregated view of active, idle, and idle in transaction connections.
- **Database Size:** How large databases are within a PostgreSQL cluster. Typically combined with another metric for analysis. Helps keep track of overall disk usage and if any triage steps need to occur around PVC size.
- **WAL Size:** How much space write-ahead logs (WAL) are taking up on disk. This can contribute to extra space being used on your data disk, or can give you an indication of how much space is being utilized on a separate WAL PVC. If you are using replication slots, this can help indicate if a slot is not being acknowledged if the numbers are much larger than the **max_wal_size** setting (the PostgreSQL Operator does not use slots by default).
- **Row Activity:** The number of rows that are selected, inserted, updated, and deleted. This can help you determine what percentage of your workload is read vs. write, and help make database tuning decisions based on that, in conjunction with other metrics.
- **Replication Status:** Provides guidance information on how much replication lag there is between primary and replica PostgreSQL instances, both in bytes and time. This can provide an indication of how much data could be lost in the event of a failover.



- Conflicts / Deadlocks: These occur when PostgreSQL is unable to complete operations, which can result in transaction loss. The goal is for these numbers to be 0. If these are occurring, check your data access and writing patterns.
- Cache Hit Ratio: A measure of how much of the "working data", e.g. data that is being accessed and manipulated, resides in memory. This is used to understand how much PostgreSQL is having to utilize the disk. The target number of this should be as high as possible. How to achieve this is the subject of books, but certain takes efforts on your applications use PostgreSQL.
- Buffers: The buffer usage of various parts of the PostgreSQL system. This can be used to help understand the overall throughput between various parts of the system.
- Commit & Rollback: How many transactions are committed and rolled back.
- Locks: The number of locks that are present on a given system.

pgBouncer

 Info

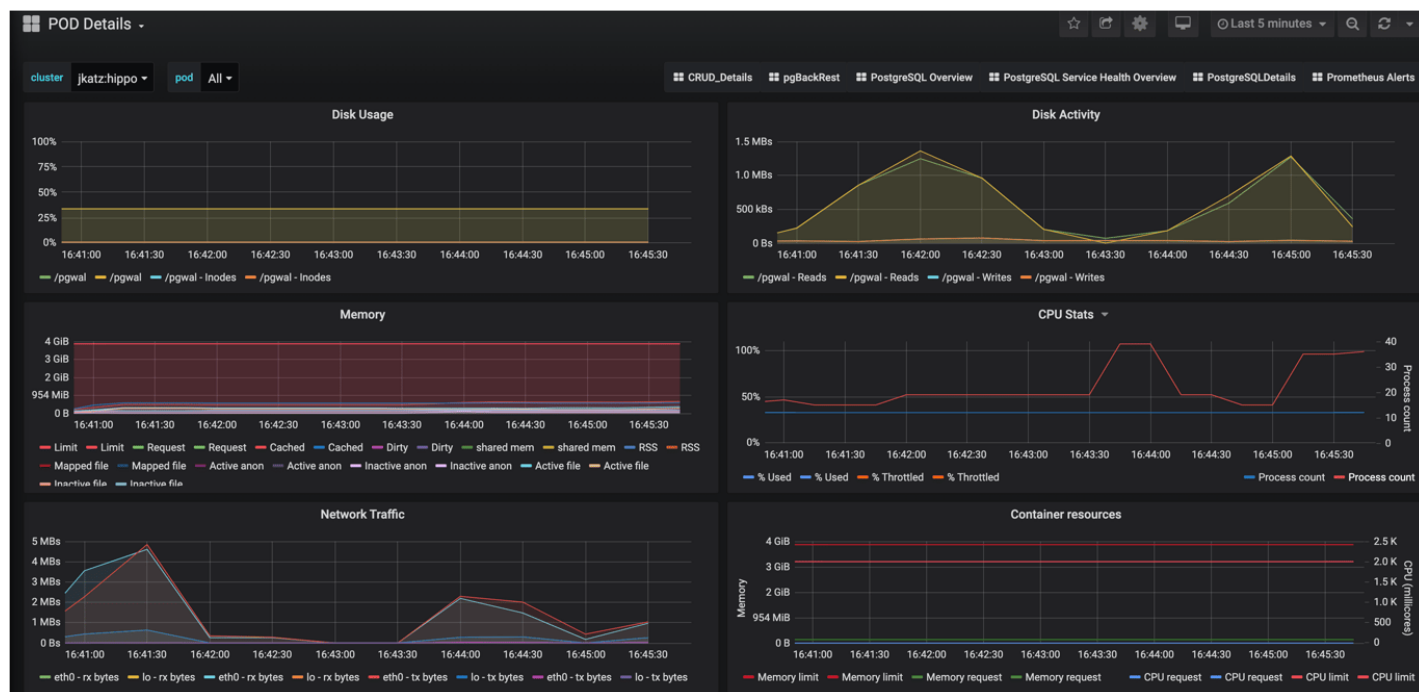
The pgBouncer dashboard will only have relevant metrics when using the **OpenTelemetryMetrics** feature gate, available in CPK v5.8.0 and above. Check the OpenTelemetry observability page for more information.

The pgBouncer dashboard provides details from the PgBouncer metrics exposed by the OpenTelemetry collector sidecar. The OpenTelemetry collector sidecar is configured to query pgBouncer with the built-in SHOW command views found in the pgBouncer documentation. For instance, metrics prefixed with **ccp_pg_bouncer_pools_** are derived from pgBouncer's **SHOW POOLS** command. See [pgBouncer documentation](#) for more on those commands.

Metrics here can be filtered by PostgresCluster, by pgBouncer pod, and finally by database pool. These metrics/visualizations include:

- PGBouncer Total State Counts: Counts for clients and servers by state (active, idle, used, waiting)
- PGBouncer Total Item Counts: Count of items registered with pgBouncer
- Connection % Used Per Pool: Current number of connections for this database as a percentage of the maximum pool size
- Client Connection State Counts Per Pool: Current waiting time in seconds
- Server Connection State Counts Per Pool: Connections with `close_needed` condition

Pod Details

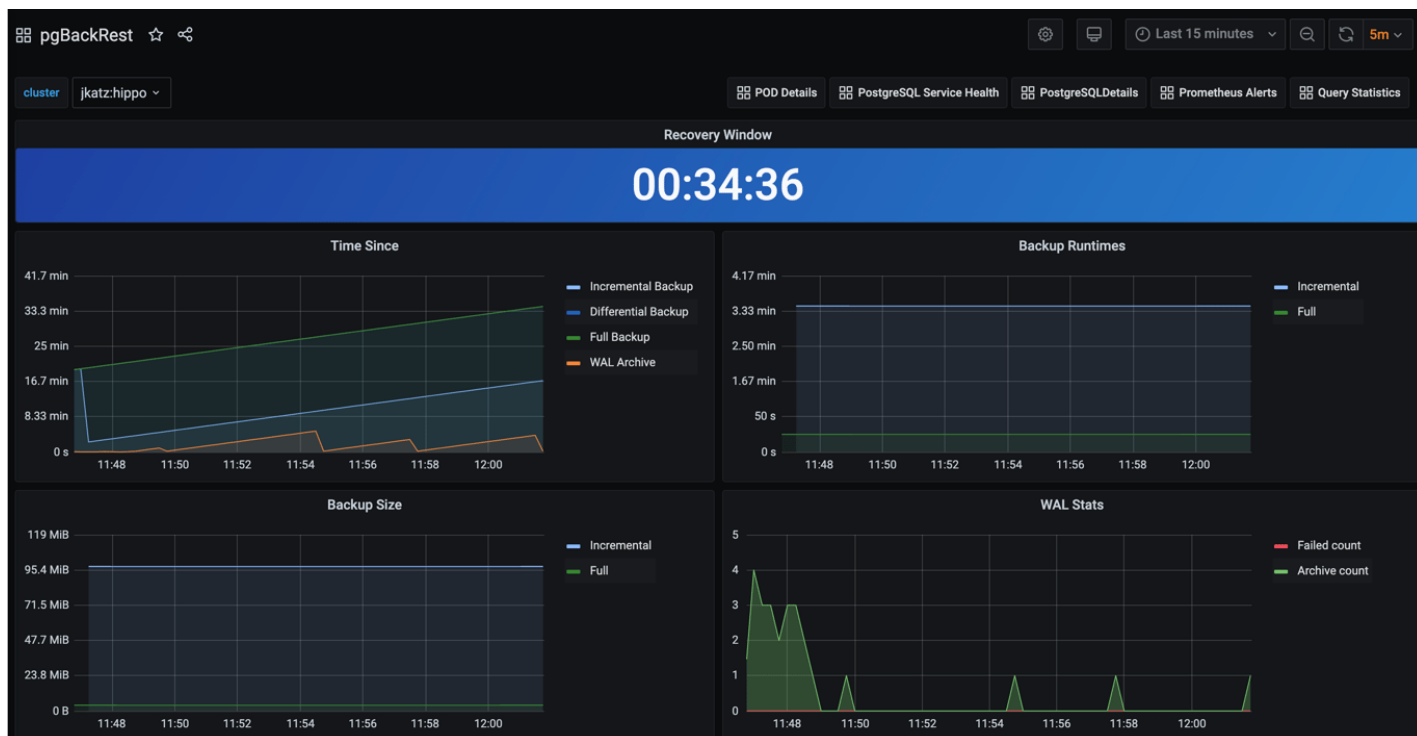


Pod details provide information about a given Pod or Pods that are being used by a PostgreSQL cluster. These are similar to "operating system" or "node" metrics, with the differences that these are looking at resource utilization by a container, not the entire node.

It may be helpful to view these metrics on a "pod" basis, by using the Pod filter at the top of the dashboard.

- Disk Usage: How much space is being consumed by a volume.
- Disk Activity: How many reads and writes are occurring on a volume.
- Memory: Various information about memory utilization, including the request and limit as well as actually utilization.
- CPU: The amount of CPU being utilized by a Pod
- Network Traffic: The amount of networking traffic passing through each network device.
- Container Resources: The CPU and memory limits and requests.

Backups



There are a variety of reasons why you need to monitor your backups, starting from answering the fundamental question of "do I have backups available?" Backups can be used for a variety of situations, from cloning new clusters to restoring clusters after a disaster. Additionally, Postgres can run into issues if your backup repository is not healthy, e.g. if it cannot push WAL archives. If your backups are set up properly and healthy, you will be set up to mitigate the risk of data loss!

The backup, or pgBackRest panel, will provide information about the overall state of your backups. This includes:

- **Recovery Window:** This is an indicator of how far back you are able to restore your data from. This represents all of the backups and archives available in your backup repository. Typically, your recovery window should be close to your overall data retention specifications.
- **Time Since Last Backup:** this indicates how long it has been since your last backup. This is broken down into pgBackRest backup type (full, incremental, differential) as well as time since the last WAL archive was pushed.
- **Backup Runtimes:** How long the last backup of a given type (full, incremental, differential) took to execute. If your backups are slow, consider providing more resources to the backup jobs and tweaking pgBackRest's performance tuning settings.
- **Backup Size:** How large the backups of a given type (full, incremental, differential).
- **WAL Stats:** Shows the metrics around WAL archive pushes. If you have failing pushes, you should be able to see if there is a transient or permanent error that is preventing WAL archives from being pushed. If left untreated, this could end up causing issues for your Postgres cluster.

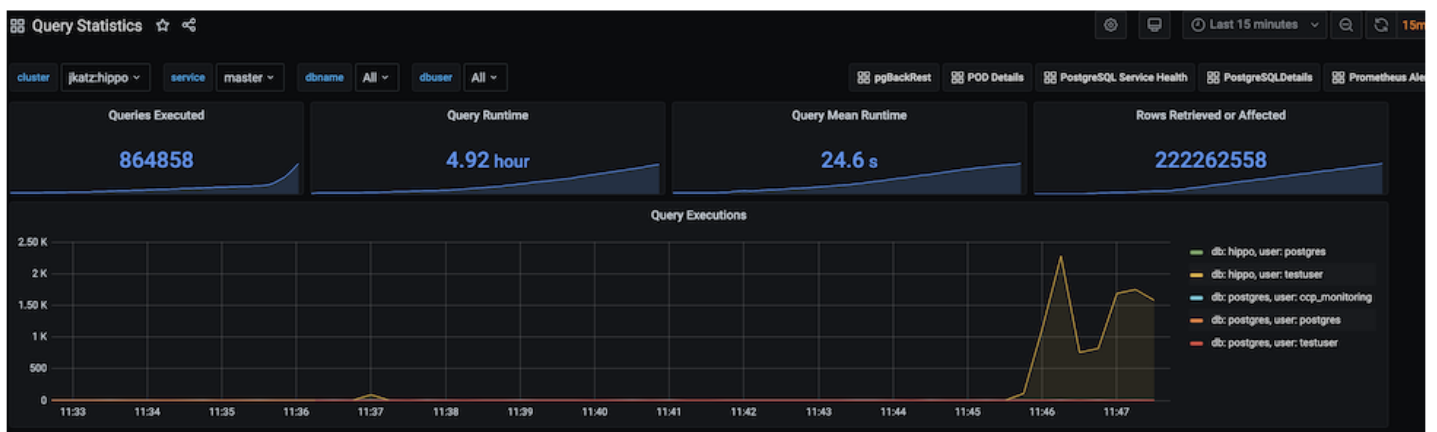
PostgreSQL Service Health Overview



The Service Health Overview provides information about the Kubernetes Services that sit in front of the PostgreSQL Pods. This provides information about the status of the network.

- **Saturation:** How much of the available network to the Service is being consumed. High saturation may cause degraded performance to clients or create an inability to connect to the PostgreSQL cluster.
- **Traffic:** Displays the number of transactions per minute that the Service is handling.
- **Errors:** Displays the total number of errors occurring at a particular Service.
- **Latency:** What the overall network latency is when interfacing with the Service.

Query Runtime



Looking at the overall performance of queries can help optimize a Postgres deployment, both from providing resources to query tuning in the application itself.

You can get a sense of the overall activity of a PostgreSQL cluster from the chart that is visualized above:

- **Queries Executed:** The total number of queries executed on a system during the period.
- **Query runtime:** The aggregate runtime of all the queries combined across the system that were executed in the period.
- **Query mean runtime:** The average query time across all queries executed on the system in the given period.

- Rows retrieved or affected: The total number of rows in a database that were either retrieved or had modifications made to them.

PostgreSQL Operator Monitoring also further breaks down the queries so you can identify queries that are being executed too frequently or are taking up too much time.

Query Statistics

🔍

🗨

🕒 Last 15 minutes

🔍

🔄 15m

Query Mean Runtime (Top N)					
dbname	role	query	Runtime	exported_role	pg_cluster
hippo	master	copy pgbench_accounts from stdin	58.8 s	testuser	jkatzhippo
hippo	master	vacuum analyze pgbench_accounts	50.9 s	testuser	jkatzhippo
postgres	master	select lsn:text as lsn, pg_catal	31.6 s	postgres	jkatzhippo
hippo	master	alter table pgbench_accounts add primary	11.6 s	testuser	jkatzhippo
postgres	master	CREATE DATABASE "hippo"	1.51 s	postgres	jkatzhippo
postgres	master	SELECT current_database() as dbname, n.n	282 ms	ccp_monitoring	jkatzhippo

Query Max Runtime (Top N)					
dbname	role	query	Runtime	exported_role	pg_cluster
postgres	master	select lsn:text as lsn, pg_catal	2.50 min	postgres	jkatzhippo
hippo	master	vacuum analyze pgbench_accounts	2.10 min	testuser	jkatzhippo
hippo	master	copy pgbench_accounts from stdin	1.63 min	testuser	jkatzhippo
hippo	master	alter table pgbench_accounts add primary	28.3 s	testuser	jkatzhippo
postgres	master	SELECT current_database() as dbname, n.n	6.66 s	ccp_monitoring	jkatzhippo
postgres	master	SELECT dbname as dbname, pg_database_sl	3.99 s	ccp_monitoring	jkatzhippo

Query Total Runtime (Top N)					
dbname	role	query	Runtime	exported_role	pg_cluster
hippo	master	UPDATE pgbench_accounts SET abalance = a	861 ms	testuser	jkatzhippo
hippo	master	UPDATE pgbench_branches SET bbalance = b	127 ms	testuser	jkatzhippo
hippo	master	UPDATE pgbench_tellers SET tbalance = tb	19.2 ms	testuser	jkatzhippo
postgres	master	WITH all_backups AS (SELECT config_file	4.53 ms	ccp_monitoring	jkatzhippo
postgres	master	SELECT * FROM pg_stat_database	4.18 ms	ccp_monitoring	jkatzhippo
postgres	master	SELECT * FROM pg_stat_database_conflicts	1.04 ms	ccp_monitoring	jkatzhippo

- Query Mean Runtime (Top N): This highlights the N number of slowest queries by average runtime on the system. This might indicate you are missing an index somewhere, or perhaps the query could be rewritten to be more efficient.
- Query Max Runtime (Top N): This highlights the N number of slowest queries by absolute runtime. This could indicate that a specific query or the system as a whole may need more resources.
- Query Total Runtime (Top N): This highlights the N of slowest queries by aggregate runtime. This could indicate that a ORM is looping over a single query and executing it many times that could possibly be rewritten as a single, faster query.

Alerts

Prometheus Alerts											
Active Alerts											
Time	alertname	deployment	exp_type	instance	ip	kubernetes_namespace	pg_cluster	pod	service	severity	severity_num
2020-09-01 17:24:57	PGIsUp	zebra	pg	10.44.0.7:9187	10.44.0.7	jkatz	jkatz:zebra	zebra-bfff764bd-mrm4w	postgresql	critical	300
2020-09-01 17:24:57	PGExporterScrapeError	zebra	pg	10.44.0.7:9187	10.44.0.7	jkatz	jkatz:zebra	zebra-bfff764bd-mrm4w	postgresql	critical	300

Alerting lets one view and receive alerts about actions that require intervention, for example, a HA cluster that cannot self-heal. The alerting system is powered by [Alertmanager](#).

The alerts that come installed by default include:

- **PGExporterScrapeError**: The Crunchy PostgreSQL Exporter is having issues scraping statistics used as part of the monitoring stack.
- **PGIsUp**: A PostgreSQL instance is down.
- **PGIdleTxn**: There are too many connections that are in the "idle in transaction" state.
- **PGQueryTime**: A single PostgreSQL query is taking too long to run. Issues a warning at 12 hours and goes critical after 24.
- **PGConnPerc**: Indicates that there are too many connection slots being used. Issues a warning at 75% and goes critical above 90%.
- **PGDiskSize**: Indicates that a PostgreSQL database is too large and could be in danger of running out of disk space. Issues a warning at 75% and goes critical at 90%.
- **PGReplicationByteLag**: Indicates that a replica is too far behind a primary instance, which could risk data loss in a failover scenario. Issues a warning at 50MB and goes critical at 100MB.
- **PGReplicationSlotsInactive**: Indicates that a replication slot is inactive. Not attending to this can lead to out-of-disk errors.
- **PGXIDWraparound**: Indicates that a PostgreSQL instance is nearing transaction ID wraparound. Issues a warning at 50% and goes critical at 75%. It's important that you [vacuum your database](#) to prevent this.
- **PGEmergencyVacuum**: Indicates that autovacuum is not running or cannot keep up with ongoing changes, i.e. it's past its "freeze" age. Issues a warning at 110% and goes critical at 125%.
- **PGArchiveCommandStatus**: Indicates that the archive command, which is used to ship WAL archives to pgBackRest, is failing.
- **PGSequenceExhaustion**: Indicates that a sequence is over 75% used.
- **PGSettingsPendingRestart**: Indicates that there are settings changed on a PostgreSQL instance that requires a restart.

Optional alerts that can be enabled:

- **PGMinimumVersion**: Indicates if PostgreSQL is below a desired version.
- **PGRecoveryStatusSwitch_Replica**: Indicates that a replica has been promoted to a primary.
- **PGConnectionAbsent_Prod**: Indicates that metrics collection is absent from a PostgreSQL instance.
- **PGSettingsChecksum**: Indicates that PostgreSQL settings have changed from a previous state.
- **PGDataChecksum**: Indicates that there are data checksum failures on a PostgreSQL instance. This could be a sign of data corruption.

You can modify these alerts as you see fit, and add your own alerts as well! Please see the installation instructions for general setup of the PostgreSQL Operator Monitoring stack.

Disaster Recovery

Advanced high-availability and disaster recovery strategies involve spreading your database clusters across multiple data centers to help maximize uptime. In Kubernetes, this technique is known as "[federation](#)". Federated Kubernetes clusters can communicate with each other, coordinate changes, and provide resiliency for applications that have high uptime requirements.

As of this writing, federation in Kubernetes is still in ongoing development and is something we monitor with intense interest. As Kubernetes federation continues to mature, we wanted to provide a way to deploy PostgreSQL clusters managed by the PostgreSQL Operator that can span multiple Kubernetes clusters.

At a high-level, the PostgreSQL Operator follows the "active-standby" data center deployment model for managing the PostgreSQL clusters across Kubernetes clusters. In one Kubernetes cluster, the PostgreSQL Operator deploys PostgreSQL as an "active" PostgreSQL cluster, which means it has one primary and one-or-more replicas. In another Kubernetes cluster, the PostgreSQL cluster is deployed as a "standby" cluster: every PostgreSQL instance is a replica.

A side-effect of this is that in each of the Kubernetes clusters, the PostgreSQL Operator can be used to deploy both active and standby PostgreSQL clusters, allowing you to mix and match! While the mixing and matching may not be ideal for how you deploy your PostgreSQL clusters, it does allow you to perform online moves of your PostgreSQL data to different Kubernetes clusters as well as manual online upgrades.

Lastly, while this feature does extend high-availability, promoting a standby cluster to an active cluster is **not** automatic. While the PostgreSQL clusters within a Kubernetes cluster support self-managed high-availability, a cross-cluster deployment requires someone to promote the cluster from standby to active.

Standby Cluster Overview

Standby PostgreSQL clusters are managed like any other PostgreSQL cluster that the PostgreSQL Operator manages. For example, adding replicas to a standby cluster is identical to adding them to a primary cluster.

The main difference between a primary and standby cluster is that there is no primary instance on the standby: one PostgreSQL instance is reading in the database changes from either the backup repository or via streaming replication, while other instances are replicas of it.

Any replicas created in the standby cluster are known as cascading replicas, i.e., replicas replicating from a database server that itself is replicating from another database server. More information about [cascading replication](#) can be found in the PostgreSQL documentation.

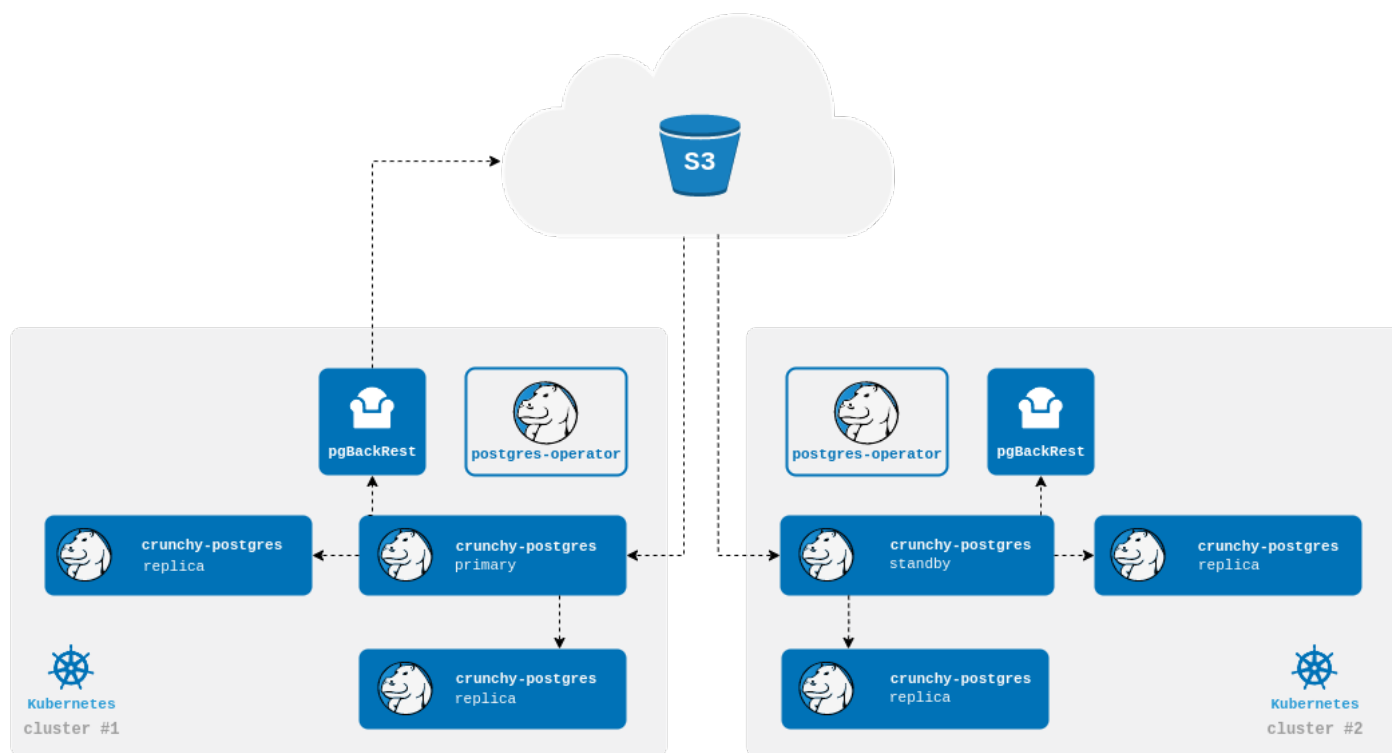
Because standby clusters are effectively read-only, certain functionality that involves making changes to a database, e.g., PostgreSQL user changes, is blocked while a cluster is in standby mode. Additionally, backups and restores are blocked as well. While [pgBackRest](#) supports backups from standbys, this requires direct access to the primary database, which cannot be done until the PostgreSQL Operator supports Kubernetes federation.

Types of Standby Clusters

There are three ways to deploy a standby cluster with the Postgres Operator.

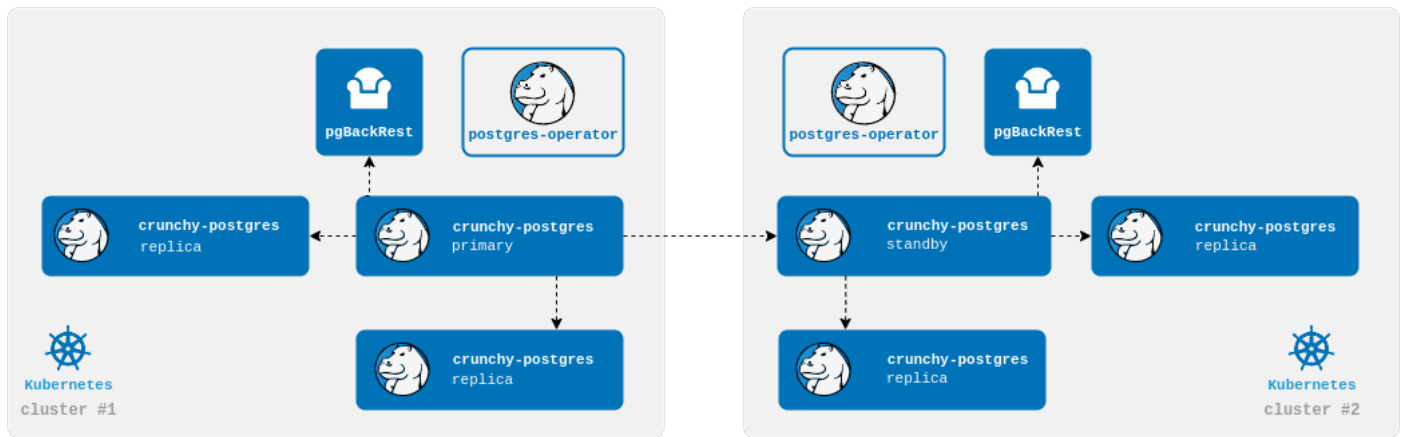
Repo-based Standby

A repo-based standby will connect to a pgBackRest repo stored in an external storage system (S3, GCS, Azure Blob Storage, or any other Kubernetes storage system that can span multiple clusters). The standby cluster will receive WAL files from the repo and will apply those to the database.



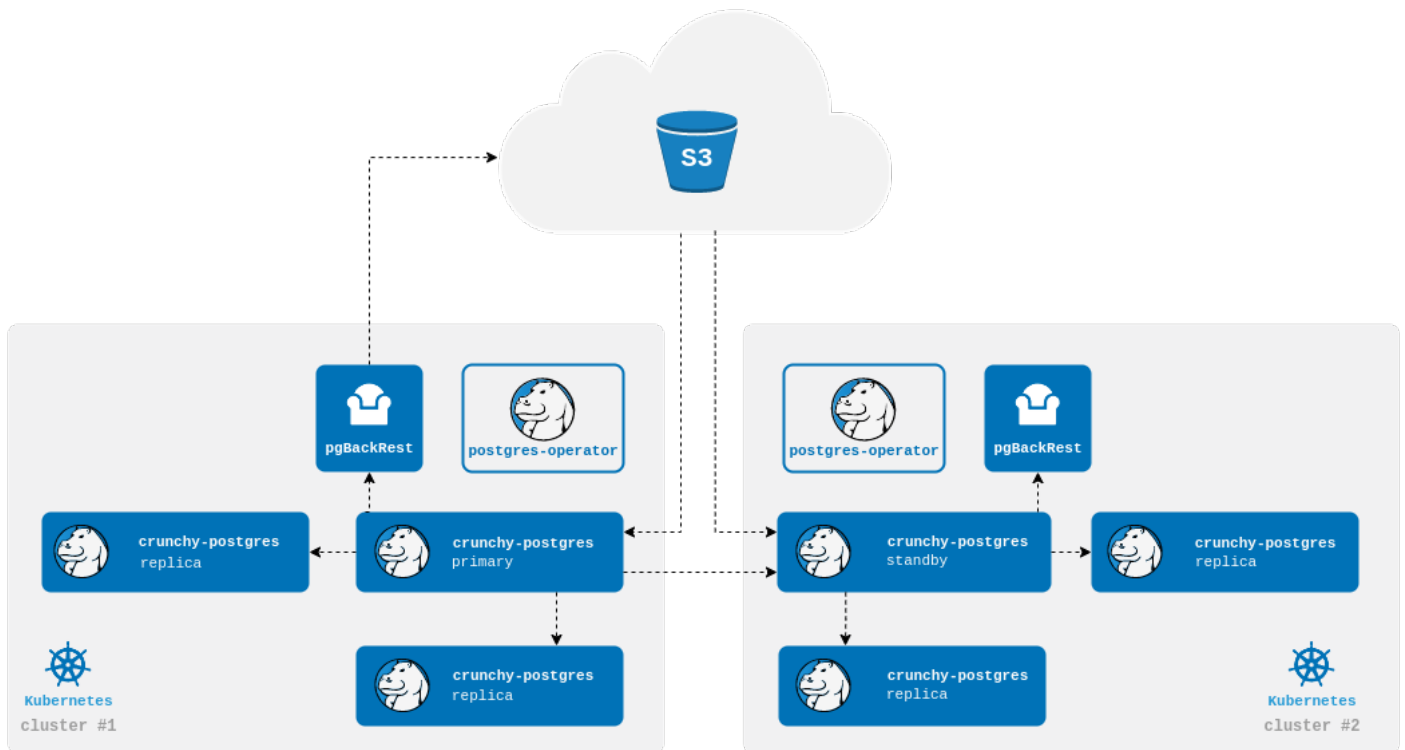
Streaming Standby

A streaming standby relies on an authenticated connection to the primary over the network. The standby will receive WAL records directly from the primary as they are generated.



Streaming Standby with an External Repo

You can also configure the operator to create a cluster that takes advantage of both methods. The standby cluster will bootstrap from the pgBackRest repo and continue to receive WAL files as they are pushed to the repo. The cluster will also directly connect to primary and receive WAL records as they are generated. Using a repo while also streaming ensures that your cluster will still be up to date with the pgBackRest repo if streaming falls behind.



For creating a standby Postgres cluster with PGO, please see the disaster recovery tutorial.

Promoting a Standby Cluster

There comes a time when a standby cluster needs to be promoted to an active cluster. Promoting a standby cluster means that the standby leader PostgreSQL instance will become a primary and start accepting both reads and writes. This has

the net effect of pushing WAL (transaction archives) to the pgBackRest repository. Before doing this, we need to ensure we don't accidentally create a split-brain scenario.

If you are promoting the standby while the primary is still running, i.e., if this is not a disaster scenario, you will want to shutdown the active PostgreSQL cluster.

The standby can be promoted once the primary is inactive, e.g., is either **shutdown** or failing. This process essentially removes the standby configuration from the Kubernetes cluster's DCS, which triggers the promotion of the current standby leader to a primary PostgreSQL instance. You can view this promotion in the PostgreSQL standby leader's (soon to be active leader's) logs.

Once the former standby cluster has been successfully promoted to an active PostgreSQL cluster, the original active PostgreSQL cluster can be safely deleted and recreated as a standby cluster.

Upgrade

Upgrading to a new version of Crunchy Postgres for Kubernetes (CPK) depends on the tool used during the initial install, as well as the version being upgraded to. This section provides detailed instructions for upgrading CPK 5.x using Kustomize, Helm or OperatorHub, along with information for upgrading from CPK v4 to CPK v5.

Info

Depending on version updates, upgrading CPK may automatically rollout changes to managed Postgres clusters. This could result in downtime--we cannot guarantee no interruption of service, though CPK attempts graceful incremental rollouts of affected pods, with the goal of zero downtime.

Registering CPK Prior to Upgrading

A registration token is required when upgrading Certified and Marketplace OperatorHub installations. Therefore, if you have installed CPK using these either of these installation methods, be sure to [register your Crunchy Postgres for Kubernetes installation](#) prior to upgrading.

Upgrading CPK 5.x

- Kustomize Upgrade
- Helm Upgrade
- OperatorHub Upgrade

Upgrading from CPK v4 to CPK v5

- V4 to V5 Upgrade Methods

Kustomize

If you installed Crunchy Postgres for Kubernetes (CPK) using Kustomize and a `kubectl apply` command, you can upgrade in most cases as simply as re-running the command after you've pulled in the new changes. For instance, assuming you are using the CPK installation from the [Postgres Operator examples repository](#), you would simply issue the command:

```
kubectl apply --server-side -k kustomize/install/default
```

Upgrading from CPK v5.3.x and Below

CPK versions from 5.1.x through 5.3.x include a pgo-upgrade deployment, which is no longer needed. After upgrading to v5.4.x, delete the deployment:

```
kubectl delete deployment pgo-upgrade
```

Upgrading from CPK v5.0.x and below

Upgrading from these versions of CPK requires additional steps. Please reference the v5.1.8 Upgrade documentation for more information. Once you have completed the steps to upgrade to CPK v5.1.8, you can continue your upgrade normally.

Helm

Once Crunchy Postgres for Kubernetes (CPK) v5 has been installed with Helm, it can then be upgraded using the `helm upgrade` command. However, before running the `upgrade` command, any CustomResourceDefinitions (CRDs) must first be manually updated. (This is specifically due to a [design decision in Helm v3](#), in which any CRDs in the Helm chart are only applied when using the `helm install` command.)

If you would like, before upgrading the CRDs, you can review the changes with `kubectl diff`. They can be verbose, so a pager like `less` may be useful:

```
kubectl diff -f helm/install/crds | less
```

Use the following command to update the CRDs using [server-side apply](#) before running `helm upgrade`. The `--force-conflicts` flag tells Kubernetes that you recognize Helm created the CRDs during `helm install`.

```
kubectl apply --server-side --force-conflicts -f helm/install/crds
```

Then, perform the upgrade using Helm:

```
helm upgrade $NAME -n $NAMESPACE
```

Upgrading from CPK v5.3.x and Below

CPK versions earlier than v5.4.0 include a pgo-upgrade deployment. When upgrading to v5.4.x, users should expect the pgo-upgrade deployment to be deleted automatically.

Upgrading from CPK v5.0.x and below

Upgrading from these versions of CPK requires additional steps. Please reference the v5.1.8 Upgrade documentation for more information. Once you have completed the steps to upgrade to CPK v5.1.8, you can continue your upgrade normally.

OperatorHub

Upgrading Crunchy Postgres for Kubernetes Using OperatorHub on OpenShift

OperatorHub provides multiple upgrade approval strategies, which are configured during installation of Crunchy Postgres for Kubernetes. Therefore, whether Crunchy Postgres for Kubernetes installation upgrades automatically or manually will depend on the specific approval strategy selected. Please see the [OperatorHub upgrade documentation](#) for additional details about available upgrade strategies.

Registering Crunchy Postgres for Kubernetes Prior to Upgrading

As described in the OperatorHub installation guide, the Marketplace and Certified installers have a registration requirement. This requirement will be enforced when Crunchy Postgres for Kubernetes is upgraded. Therefore, to ensure your Crunchy Postgres for Kubernetes services remain uninterrupted, please be sure to [register your Crunchy Postgres for Kubernetes installation](#).

CPK v4 to CPK v5

You can upgrade from CPK v4 to CPK v5 through a variety of methods by following this guide. We present these methods based upon a variety of factors, including but not limited to:

- Redundancy / ability to roll back
- Available resources
- Downtime preferences

These methods include:

- *Migrating Using Data Volumes* This allows you to migrate from v4 to v5 using the existing data volumes that you created in v4. This is the simplest method for upgrade and is the most resource efficient, but you will have a greater potential for downtime using this method.

- *Migrate From Backups* This allows you to create a Postgres cluster with v5 from the backups taken with v4. This provides a way for you to create a preview of your Postgres cluster through v5, but you would need to take your applications offline to ensure all the data is migrated.
- *Migrate Using a Standby Cluster* This allows you to run a v4 and a v5 Postgres cluster in parallel, with data replicating from the v4 cluster to the v5 cluster. This method minimizes downtime and lets you preview your v5 environment, but is the most resource intensive.

You should choose the method that makes the most sense for your environment.

Prerequisites

There are several prerequisites for using any of these upgrade methods.

- CPK v4 is currently installed within the Kubernetes cluster, and is actively managing any existing v4 PostgreSQL clusters.
- Any CPK v4 clusters being upgraded have been properly initialized using CPK v4, which means the v4 **pgcluster** custom resource should be in a **pgcluster Initialized** status:

```
kubectl get pgcluster hippo -o jsonpath='{ .status }'
{"message":"Cluster has been initialized","state":"pgcluster Initialized"}
```

- The CPK v4 **pgo** client is properly configured and available for use.
- CPK v5 is currently installed within the Kubernetes cluster.

For these examples, we will use a Postgres cluster named **hippo**.

Additional Considerations

Upgrading to CPK v5 may result in a base image upgrade from EL-7 (UBI / CentOS) to EL-8 (UBI). Based on the contents of your Postgres database, you may need to perform additional steps.

Due to changes in the GNU C library **glibc** in EL-8, you may need to reindex certain indexes in your Postgres cluster. For more information, please read the [PostgreSQL Wiki on Locale Data Changes](#), how you can determine if your indexes are affected, and how to fix them.

Upgrade Method #1: Data Volumes

Info

Before attempting to upgrade from v4.x to v5, please familiarize yourself with the prerequisites applicable for all v4.x to v5 upgrade methods.

This upgrade method allows you to migrate from CPK v4 to CPK v5 using the existing data volumes that were created in CPK v4. Note that this is an "in place" migration method: this will immediately move your Postgres clusters from being

managed by CPK v4 to being managed by CPK v5. If you wish to have some failsafes in place, please use one of the other migration methods.

Please also note that you will need to perform the cluster upgrade in the same namespace as the original cluster in order for your v5 cluster to access the existing PVCs.

Step 1: Prepare the CPK v4 Cluster for Migration

You will need to set up your CPK v4 Postgres cluster so that it can be migrated to a CPK v5 cluster. The following describes how to set up a CPK v4 cluster for using this migration method.

- Scale down any existing replicas within the cluster. This will ensure that the primary PVC does not change again prior to the upgrade.

You can get a list of replicas using the `pgo scaledown --query` command, e.g.:

```
pgo scaledown hippo --query
```

If there are any replicas, you will see something similar to:

Cluster: hippo		
REPLICA	STATUS	NODE ...
hippo	running	node01 ...

Scaledown any replicas that are running in this cluster, e.g.:

```
pgo scaledown hippo --target=hippo
```

- Once all replicas are removed and only the primary remains, proceed with deleting the cluster while retaining the data and backups. You can do this with the `--keep-data` and `--keep-backups` flags:

You MUST run this command with the `--keep-data` and `--keep-backups` flag otherwise you risk deleting ALL of your data.

```
pgo delete cluster hippo --keep-data --keep-backups
```

- The PVC for the primary Postgres instance and the pgBackRest repository should still remain. You can verify this with the command below:

```
kubectl get pvc --selector=pg-cluster=hippo
```

This should yield something similar to:

NAME	STATUS	VOLUME ...	
hippo-jgut	Bound	pvc-a0b89bdb-	...
hippo-pgbr-repo	Bound	pvc-25501671-	...

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

Step 2: Migrate to CPK v5

With the CPK v4 cluster's volumes prepared for the move to CPK v5, you can now create a **PostgresCluster** custom resource using these volumes. This migration method does not carry over any specific configurations or customizations from CPK v4: you will need to create the specific **PostgresCluster** configuration that you need.

⚠ Warning

Additional steps are required to set proper file permissions when using certain storage options, such as NFS and HostPath storage, due to a known issue with how fsGroups are applied. When migrating from CPK v4, this will require the user to manually set the group value of the pgBackRest repo directory, and all subdirectories, to **26** to match the **postgres** group used in CPK v5. Please see [here](#) for more information.

To complete the upgrade process, your **PostgresCluster** custom resource **MUST** include the following:

- A **spec.dataSource.volumes** section that points to the PostgreSQL data, PostgreSQL WAL (if applicable) and pgBackRest repository PVCs from the v4 cluster.

For example, using the **hippo** cluster:

```
spec:
  dataSource:
    volumes:
      pgDataVolume:
        pvcName: hippo-jgut
        directory: "hippo-jgut"
      pgBackRestVolume:
        pvcName: hippo-pgbr-repo
        directory: "hippo-backrest-shared-repo"
      # Only specify external WAL PVC if enabled in CPK v4 cluster. If enabled
      # in v4, a WAL volume must be defined for the v5 cluster as well.
      # pgWALVolume:
      #   pvcName: hippo-jgut-wal
```

Please see the Data Migration section for more details on how to properly populate this section of the spec when migrating from a CPK v4 cluster.

ℹ Info

Note that when migrating data volumes from v4 to v5, CPK relabels all volumes for CPK v5, but **will not remove existing CPK v4 labels**. This results in PVCs that are labeled for both CPK v4 and v5, which can lead to unintended behavior.

To avoid that behavior, follow the instructions in the section on removing CPK v4 labels.

- If you customized Postgres parameters, you will need to ensure they match in the CPK v5 cluster. For more information, please review the tutorial on customizing a Postgres cluster.
- Once the **PostgresCluster** spec is populated according to these guidelines, you can create the **PostgresCluster** custom resource. For example, if the **PostgresCluster** you're creating is a modified version of the [postgres example](#) in the [CPK examples repo](#), you can run the following command:

```
kubectl apply -k kustomize/postgres
```

Your upgrade is now complete! You should now remove the `spec.dataSource.volumes` section from your `PostgresCluster`. For more information on how to use CPK v5, we recommend reading through the CPK v5 tutorial.

Upgrade Method #2: Backups

Info

Before attempting to upgrade from v4.x to v5, please familiarize yourself with the prerequisites applicable for all v4.x to v5 upgrade methods.

This upgrade method allows you to migrate from CPK v4 to CPK v5 by creating a new CPK v5 Postgres cluster using a backup from a CPK v4 cluster. This method allows you to preserve the data in your CPK v4 cluster while you transition to CPK v5. To fully move the data over, you will need to incur downtime and shut down your CPK v4 cluster.

Step 1: Prepare the CPK v4 Cluster for Migration

- Ensure you have a recent backup of your cluster. You can do so with the `pgo backup` command, e.g.:

```
pgo backup hippo
```

Please ensure that the backup completes. You will see the latest backup appear using the `pgo show backup` command.

- Next, delete the cluster while keeping backups (using the `--keep-backups` flag):

```
pgo delete cluster hippo --keep-backups
```

Warning

Additional steps are required to set proper file permissions when using certain storage options, such as NFS and HostPath storage, due to a known issue with how fsGroups are applied. When migrating from CPK v4, this will require the user to manually set the group value of the pgBackRest repo directory, and all subdirectories, to `26` to match the `postgres` group used in CPK v5. Please see [here](#) for more information.

Step 2: Migrate to CPK v5

With the CPK v4 Postgres cluster's backup repository prepared, you can now create a `PostgresCluster` custom resource. This migration method does not carry over any specific configurations or customizations from CPK v4: you will need to create the specific `PostgresCluster` configuration that you need.

To complete the upgrade process, your `PostgresCluster` custom resource **MUST** include the following:

- You will need to configure your pgBackRest repository based upon whether you are using a PVC to store your backups, or an object storage system such as S3/GCS. Please follow the directions based on the repository type you are using.

PVC-based Backup Repository

When migrating from a PVC-based backup repository, you will need to configure a pgBackRest repo at `spec.backups.pgbackrest.repos.volume` with the name `repo1`. The `volumeClaimSpec` should match the attributes of the pgBackRest repo PVC being used as part of the migration, i.e. it must have the same `storageClassName`, `accessModes`, `resources`, etc. For example, if your v4 Postgres cluster volume was 1Gi of `standard` storage with a `ReadWriteOnce` access mode, your v5 cluster would look something like this (note the `repo1` name):

```
spec:
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              storageClassName: standard
              accessModes:
                - "ReadWriteOnce"
              resources:
                requests:
                  storage: 1Gi
```

Please note that you will need to perform the cluster upgrade in the same namespace as the original cluster in order for your v5 cluster to access the existing PVCs.

S3 / GCS Backup Repository

When migrating from a S3 or GCS based backup repository, you will need to configure your `spec.backups.pgbackrest.repos.volume` to point to the backup storage system. For instance, if AWS S3 storage is being utilized, the repo would be defined similar to the following:

```
spec:
  backups:
    pgbackrest:
      repos:
        - name: repo1
          s3:
            bucket: hippo
            endpoint: s3.amazonaws.com
            region: us-east-1
```

Any required secrets or desired custom pgBackRest configuration should be created and configured as described in the backup tutorial.

You will also need to ensure that the “pgbackrest-repo-path” configured for the repository matches the path used by the CPK v4 cluster. The default repository path follows the pattern `/backrestrepo/<clusterName>-backrest-shared-repo`. Note that the path name here is different than migrating from a PVC-based repository.

Using the `hippo` Postgres cluster as an example, you would set the following in the `spec.backups.pgbackrest.global` section:

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

- Once you have configured the pgBackRest repository configuration in step 1, set the `spec.dataSource` section to restore from the backups used for this migration. For example:

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

You can also provide other pgBackRest restore options, e.g. if you wish to restore to a specific point-in-time (PITR).

- If you are using a PVC-based pgBackRest repository, then you will also need to specify a pgBackRestVolume data source that references the CPK v4 pgBackRest repository PVC:

```
spec:
  dataSource:
    volumes:
      pgBackRestVolume:
        pvcName: hippo-pgbr-repo
        directory: "hippo-backrest-shared-repo"
      postgresCluster:
        repoName: repo1
```

- If you customized other Postgres parameters, you will need to ensure they match in the CPK v5 cluster. For more information, please review the tutorial on customizing a Postgres cluster.
- Once the `PostgresCluster` spec is populated according to these guidelines, you can create the `PostgresCluster` custom resource. For example, if the `PostgresCluster` you're creating is a modified version of the [postgres example](#) in the [CPK examples repo](#), you can run the following command:

```
kubectl apply -k kustomize/postgres
```

WARNING: Once the `PostgresCluster` custom resource is created, it will become the owner of the PVC. *This means that if the `PostgresCluster` is then deleted (e.g. if attempting to revert back to a CPK v4 cluster), then the PVC will be deleted as well.*

If you wish to protect against this, first remove the reference to the pgBackRest PVC in the `PostgresCluster` spec:

```
kubectl patch postgrescluster hippo-pgbr-repo --type='json' -p='[{"op": "remove", "path": "/spec/dataSource/volumes"}]'
```

Then relabel the PVC prior to deleting the `PostgresCluster` custom resource:

```
kubectl label pvc hippo-pgbr-repo postgres-operator.crunchydata.com/cluster- postgres-operator.crunchydata.com/pgbackrest-repo- postgres-operator.crunchydata.com/pgbackrest-volume- postgres-operator.crunchydata.com/pgbackrest-
```

You will also need to remove all ownership references from the PVC:

```
kubectl patch pvc hippo-pgbr-repo --type='json' -p='[{"op": "remove", "path": "/metadata/ownerReferences"}]'
```

It is recommended to set the reclaim policy for any PV's bound to existing PVC's to `Retain` to ensure data is retained in the event a PVC is accidentally deleted during the upgrade.

Your upgrade is now complete! For more information on how to use CPK v5, we recommend reading through the CPK v5 tutorials.

Upgrade Method #3: Standby Cluster

Info

Before attempting to upgrade from v4.x to v5, please familiarize yourself with the prerequisites applicable for all v4.x to v5 upgrade methods.

This upgrade method allows you to migrate from CPK v4 to CPK v5 by creating a new CPK v5 Postgres cluster in a "standby" mode, allowing it to mirror the CPK v4 cluster and continue to receive data updates in real time. This has the advantage of being able to fully inspect your CPK v5 Postgres cluster while leaving your CPK v4 cluster up and running, thus minimizing downtime when you cut over. The tradeoff is that you will temporarily use more resources while this migration is occurring.

This method only works if your CPK v4 cluster uses S3 or an S3-compatible storage system, or GCS. For more information on standby clusters, please refer to the standby cluster tutorial.

Step 1: Migrate to CPK v5

Create a **PostgresCluster** custom resource. This migration method does not carry over any specific configurations or customizations from CPK v4: you will need to create the specific **PostgresCluster** configuration that you need.

To complete the upgrade process, your **PostgresCluster** custom resource **MUST** include the following:

- Configure your pgBackRest to use an object storage system such as S3/GCS. You will need to configure your **spec.backups.pgbackrest.repos.volume** to point to the backup storage system. For instance, if AWS S3 storage is being utilized, the repo would be defined similar to the following:

```
spec:
  backups:
    pgbackrest:
      repos:
        - name: repo1
          s3:
            bucket: hippo
            endpoint: s3.amazonaws.com
            region: us-east-1
```

Any required secrets or desired custom pgBackRest configuration should be created and configured as described in the backup tutorial.

You will also need to ensure that the "pgbackrest-repo-path" configured for the repository matches the path used by the CPK v4 cluster. The default repository path follows the pattern **/backrestrepo/<clusterName>-backrest-shared-repo**. Note that the path name here is different than migrating from a PVC-based repository.

Using the **hippo** Postgres cluster as an example, you would set the following in the **spec.backups.pgbackrest.global** section:

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

- A `spec.standby` cluster configuration within the spec that is populated according to the name of pgBackRest repo configured in the spec. For example:

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

- If you customized other Postgres parameters, you will need to ensure they match in the CPK v5 cluster. For more information, please review the tutorial on customizing a Postgres cluster.
- Once the `PostgresCluster` spec is populated according to these guidelines, you can create the `PostgresCluster` custom resource. For example, if the `PostgresCluster` you're creating is a modified version of the [postgres example](#) in the [CPK examples repo](#), you can run the following command:

```
kubectl apply -k kustomize/postgres
```

- Once the standby cluster is up and running and you are satisfied with your set up, you can promote it.

First, you will need to shut down your CPK v4 cluster. You can do so with the following command, e.g.:

```
pgo update cluster hippo --shutdown
```

You can then update your CPK v5 cluster spec to promote your standby cluster:

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

Note: When the v5 cluster is running in non-standby mode, you will not be able to restart the v4 cluster, as that data is now being managed by the v5 cluster.

Once the v5 cluster is up and running, you will need to run the following SQL commands as a PostgreSQL superuser. For example, you can login as the `postgres` user, or exec into the Pod and use `psql`:

```
--add the managed replication user
CREATE ROLE _crunchyrepl WITH LOGIN REPLICATION;

-- allow for the replication user to execute the functions required as part of "rewinding"
GRANT EXECUTE ON function pg_catalog.pg_ls_dir(text, boolean, boolean) TO _crunchyrepl;
GRANT EXECUTE ON function pg_catalog.pg_stat_file(text, boolean) TO _crunchyrepl;
GRANT EXECUTE ON function pg_catalog.pg_read_binary_file(text) TO _crunchyrepl;
GRANT EXECUTE ON function pg_catalog.pg_read_binary_file(text, bigint, big-
int, boolean) TO _crunchyrepl;
```

The above step will be automated in an upcoming release.

Your upgrade is now complete! Once you verify that the CPK v5 cluster is running and you have recorded the user credentials from the v4 cluster, you can remove the old cluster:

```
pgo delete cluster hippo
```

For more information on how to use CPK v5, we recommend reading through the CPK v5 tutorials.

FAQ

Project FAQ

What is The PGO Project?

The PGO Project is the open source project associated with the development of [PGO](#), the [Postgres Operator](#) for Kubernetes from [Crunchy Data](#).

PGO is a [Kubernetes Operator](#), providing a declarative solution for managing your PostgreSQL clusters. Within a few moments, you can have a Postgres cluster complete with high availability, disaster recovery, and monitoring, all over secure TLS communications.

PGO is the upstream project from which [Crunchy Postgres for Kubernetes](#) is derived. You can find more information on Crunchy Postgres for Kubernetes [here](#).

What's the difference between PGO and Crunchy Postgres for Kubernetes?

PGO is the Postgres Operator from Crunchy Data. It developed pursuant to the PGO Project and is designed to be a frequently released, fast-moving project where all new development happens.

[Crunchy Postgres for Kubernetes](#) is produced by taking selected releases of PGO, combining them with Crunchy Certified PostgreSQL and PostgreSQL containers certified by Crunchy Data, maintained for commercial support, and made available to customers as the Crunchy Postgres for Kubernetes offering.

Where can I find support for PGO?

For general questions or community support, we welcome you to join our [community Discord](#) and ask your questions there.

Information regarding support for PGO is available in the Support section of the PGO documentation, which you can find [here](#).

For additional information regarding commercial support and Crunchy Postgres for Kubernetes, you can [contact Crunchy Data](#).

Under which open source license is PGO source code available?

The PGO source code is available under the [Apache License 2.0](#).

Where are the release tags for PGO v5?

With PGO v5, we've made some changes to our overall process. Instead of providing quarterly release tags as we did with PGO v4, we're focused on ongoing active development in the v5 primary development branch (**master**, which will become **main**). Consistent with our practices in v4, previews of stable releases with the release tags are made available in the [Crunchy Data Developer Portal](#).

These changes allow for more rapid feature development and releases in the upstream PGO project, while providing [Crunchy Postgres for Kubernetes](#) users with stable releases for production use.

To the extent you have constraints specific to your use, please feel free to reach out on info@crunchydata.com to discuss how we can address those specifically.

How can I get involved with the PGO Project?

PGO is developed by the PGO Project. The PGO Project that welcomes community engagement and contribution.

The PGO source code and community issue trackers are hosted at [GitHub](#).

For questions or community support, please join our [community Discord](#).

For information regarding contribution, please review the contributor guide [here](#).

Please register for the [Crunchy Data Developer Portal mailing list](#) to receive updates regarding Crunchy Postgres for Kubernetes releases and the [Crunchy Data newsletter](#) for general updates from Crunchy Data.

Where do I report a PGO bug?

The PGO Project uses GitHub for its [issue tracking](#). You can file your issue [here](#).

How often is PGO released?

The PGO team currently plans to release new builds approximately every few weeks. The PGO team will flag certain builds as “stable” at their discretion. Note that the term “stable” does not imply fitness for production usage or any kind of warranty whatsoever.

Release Notes

Here you'll find the release notes divided by major release from 5.x onward. For earlier releases that are current in **extended support** and not receiving new fixes please refer to those versions of the documentation.

Crunchy Postgres for Kubernetes 5.8.x Release notes

Release notes for each of the 5.8.x releases.

Component versions

Crunchy Postgres for Kubernetes	Postgres	pgBackRest	pgbouncer	Patroni	pgadmin
---------------------------------	----------	------------	-----------	---------	---------

5.8.2	17.5	2.54.2	1.24	4.0.4	9.2
5.8.1	17.4	2.54.2	1.24	4.0.4	9.2
5.8.0	17.4	2.54.2	1.24	4.0.4	9.1

Postgres extension versions

Crunchy Postgres for Kubernetes	PostGIS	pgRouting	pgaudit	pg_cron	pg_partman	pgnodemx	set_user	val3json	TimescaleDB	orafce	pgvector			
5.8.2	3.1.12 (earliest)	3.5.2 (latest)	3.7.1 (latest)	1.5	3 (earliest)	17.1 (latest)	1.5	5.2.4	1.7	4.1.0	2.6	2.19.3	4.14.2	0.8.0
5.8.1	3.1.12 (earliest)	3.5.2 (latest)	3.7.1 (latest)	1.5	3 (earliest)	17.1 (latest)	1.5	5.2.4	1.7	4.1.0	2.6	2.18.2	4.14.0	0.8.0
5.8.0	3.1.12 (earliest)	3.5.2 (latest)	3.7.1 (latest)	1.5	3 (earliest)	17.1 (latest)	1.5	5.2.4	1.7	4.1.0	2.6	2.17.2	4.14.0	0.8.0

A bold version number indicates that the component version was updated in latest release.

5.8.2

Breaking Changes

- OTel Collector is now at 0.125.0. This update required changes that make the new version incompatible with previous CPK versions. Updating to CPK 5.8.2 requires updating the OTel Collector sidecar.

Changes

- PostgreSQL versions 17.5, 16.9, 15.13, 14.18 and 13.21 are now available.
- The pgBouncer is now at version 1.24.1.
- The Citus 13.0.3 is now available for Postgres 15, 16 and 17.
- The orafce is now at version 4.14.2.
- The pg_parquet 0.4.0 extension is now available.
- The Timescaledb extension is at version 2.19.3 for PG 17, 16, 15, and 14.

Fixes

- PostgresCluster now allows the **--target-timeline** restore option.
- OpenTelemetry Logs and Metrics can now be turned on with backups turned off.
- Metrics for replication slots now included with OpenTelemetryMetrics feature.
- Removed fieldPath CEL validation to maintain support for OpenShift v4.14.
- Metric and functionality for **pg_hba_checksum** now included with OpenTelemetryMetrics feature.
- OpenTelemetryMetrics: the ccp_stat_database metric is now collected every 5 seconds, and the ccp_stat_user_tables metric is now collected every 5 minutes.
- pgBackRest repository host pods no longer enter a CrashLoopBackOff when creating pgBackRest directories on CIFS mounts.

5.8.1

Changes

- Postgres Exporter is now at 0.17.1.
- pgAdmin is now at version 9.2.
- The Timescaledb extension is at version 2.18.2 for PG 17, 16, 15, and 14.

5.8.0

Features

- A new API is available for deploying and managing Crunchy Data Warehouse. Please note that deploying Crunchy Data Warehouse requires access to the Crunchy Data Warehouse container image, which is available by commercial subscription only at this time. For more information, contact Crunchy Data at info@crunchydata.com.
- Enable OpenTelemetry logging for your Postgres cluster, allowing you to export logging information for your databases and supporting infrastructure (specifically high availability, disaster recovery, connection pooling and pgAdmin logs) to a variety of OpenTelemetry compatible services and backends. Use the OpenTelemetryLogging [feature gate](#) to enable OpenTelemetry logging.
- Enable OpenTelemetry metrics collection as a seamless alternative to using the Postgres Exporter for metrics collection. Use the OpenTelemetryMetrics [feature gate](#) to enable OpenTelemetry metrics.
- Collect and visualize PgBouncer metrics using OpenTelemetry metrics collection and the CPK monitoring stack. The OpenTelemetryMetrics [feature gate](#) must be enabled to collect PgBouncer metrics.
- Operator metrics created by **controller-runtime** are now scraped by Prometheus when the Crunchy Postgres for Kubernetes monitoring stack is deployed.
- You can store Patroni logs in the Postgres data volume by configuring **spec.patroni.logging** in the PostgresCluster spec.
- Starting with Crunchy Postgres for Kubernetes v5.8, all containers are built using the Red Hat Universal Base Image 9 (UBI 9).
- Configure Postgres [client authentication](#) settings and [configuration parameters](#) using the **spec.authentication-** and **spec.config.parameters** fields in PostgresCluster. These provide immediate feedback about ignored and acceptable values.
- Use Secrets to configure OAuth2 authentication in PGAdmin.
- Provide **ipFamilies** and **ipFamilyPolicy** settings when configuring Services for the Postgres database and PgBouncer.
- Customize the credentials for the PgBouncer system account by updating the **password** field in the PgBouncer Secret.
- The operator emits a warning event when a PostgresCluster is using a major version of Postgres that is no longer receiving updates.

Breaking Changes

- Component images are now tagged with a consistent suffix. Images with the same suffix are compatible with each other. See the [Container Tags](#) section of the documentation for more details.
- PostGIS 3.0 is not supported in Crunchy Postgres for Kubernetes v5.8, and UBI 9 based builds of PostGIS 3.0 will not be provided.
- UBI 9 builds of pgAdmin4 v4.30 will not be included with Crunchy Postgres for Kubernetes v5.8. While UBI 8 builds of pgAdmin v4.30 will remain compatible with Crunchy Postgres for Kubernetes v5.8, please migrate to the [PGAdmin API](#) to ensure you are using the latest version of pgAdmin available.
- UBI 9 provides faster sorting for builtin collations, but also requires you to recreate indexes in your existing databases. Follow the instructions in Changing Base Images when upgrading to UBI 9. Please note that you do not need to immediately update your Postgres and component container images to UBI 9 following the upgrade to CPK v5.8. This means UBI 8 containers will remain compatible for the time being, so that you can update your PostgresCluster's at your earliest convenience. You can use related images to run Crunchy Postgres for Kubernetes v5.8 while still using UBI 8 images for the various **PostgresCluster** containers. It is recommended that you update your images as soon as possible to ensure you are using Postgres and component containers that are fully validated with the CPK v5.8 release line.

Changes

- The **InstanceSidecars** feature gate is now enabled by default.
- Validation has been added to ensure only cloud-based repos can be used with **spec.datasource.pgbackrest**.
- MD5 authentication is now disabled by default. Managed users have always used SCRAM-SHA-256 and are unaffected. If you have set custom MD5 passwords, use authentication rules to re-enable their access.
- PgAdmin4 is now at version 9.1.
- PgBouncer is now at version 1.24.0.
- PgBackRest is now at version 2.54.2.

Fixes

- Addressed an edge case where a snapshot doesn't yet have a status field, which resulted in a **nil** pointer error.
- Changed the pgAdmin readinessProbe path to ensure proper cleanup of pgAdmin session files.

Note: After upgrading an existing pgAdmin pod, it is safe to manually delete any old session files that are not automatically deleted by the cleanup script.

- Standby clusters now shut down properly when **spec.shutdown** is set to **true**.

Crunchy Postgres for Kubernetes 5.7.x Release notes

Release notes for each of the 5.7.x releases.

Component versions

Crunchy Postgres for Kubernetes	Postgres	pgBackRest	pgbouncer	Patroni	pgadmin
5.7.6	17.5	2.54.2	1.24	4.0.4	4.30, 9.2

5.7.5	17.4	2.54.2	1.24	4.0.4	4.30, 9.2
5.7.4	17.4	2.54.1	1.23	4.0.4	4.30, 8.14
5.7.3	17.2	2.54.1	1.23	4.0.4	4.30, 8.14
5.7.2	17.2	2.54.0	1.23	3.3.5	4.30, 8.14
5.7.1	17.2	2.53.1	1.23	3.3.4	4.30, 8.12
5.7.0	17.0	2.53.1	1.23	3.3.3	4.30, 8.12

Postgres extension versions

	Crunchy Postgres for Kubernetes	PostGIS	pgRouting	pgaudit	pg_cron	pg_partman	pgnodemx	set_user	wal2json	TimescaleDB	orafce	pgvector		
5.7.6	3.0.11 (earliest)	3.5.2 (latest)	3.6.0 (earliest)	3.4.2 (latest)	1.5.2 (earliest)	17.0 (latest)	1.9.5	5.2.4	1.7	4.1.0	2.6	2.19.3	4.14.2	0.8.0
5.7.5	3.0.11 (earliest)	3.5.2 (latest)	3.6.0 (earliest)	3.4.2 (latest)	1.5.2 (earliest)	17.0 (latest)	1.9.5	5.2.4	1.7	4.1.0	2.6	2.18.2	4.14.0	0.8.0
5.7.4	3.0.11 (earliest)	3.5.2 (latest)	3.6.0 (earliest)	3.4.2 (latest)	1.5.2 (earliest)	17.0 (latest)	1.9.5	5.2.4	1.7	4.1.0	2.6	2.17.2	4.14.0	0.8.0
5.7.3	3.0.11 (earliest)	3.4.4 (latest)	3.6.0 (earliest)	3.4.2 (latest)	1.5.2 (earliest)	17.0 (latest)	1.9.5	5.2.2	1.7	4.1.0	2.6	2.17.2	4.14.0	0.8.0
5.7.2	3.0.11 (earliest)	3.4.3 (latest)	3.6.0 (earliest)	3.4.2 (latest)	1.4.3 (earliest)	17.0 (latest)	1.9.4	5.1.0	1.7	4.1.0	2.6	2.17.2	4.14.0	0.8.0
5.7.1	2.5.11 (earliest)	3.4.3 (latest)	3.6.0 (earliest)	3.4.2 (latest)	1.4.3 (earliest)	17.0 (latest)	1.9.4	5.1.0	1.7	4.1.0	2.6	2.17.0	4.10.3	0.7.4
5.7.0	2.5.11 (earliest)	3.4.3 (latest)	3.6.0 (earliest)	3.4.2 (latest)	1.4.3 (earliest)	17.0 (latest)	1.9.4	5.1.0	1.7	4.1.0	2.6	2.17.0	4.10.3	0.7.4

A bold version number indicates that the component version was updated in latest release.

5.7.6

Changes

- PostgreSQL versions 17.5, 16.9, 15.13, 14.18 and 13.21 are now available.
- The pgBouncer is now at version 1.24.1.
- The Citus 13.0.3 is now available for Postgres 15, 16 and 17.
- The orafce is now at version 4.14.2.
- The pg_parquet 0.4.0 extension is now available.
- The Timescaledb extension is at version 2.19.3 for PG 17, 16, 15, and 14.

Fixes

- PostgresCluster now allows the `--target-timeline` restore option

5.7.5

Breaking Changes

- Component images are now tagged with a consistent suffix. Images with the same suffix are compatible with each other. See the [Container Tags](#) section of the documentation for more details.

Changes

- Postgres Exporter is now at 0.17.1.
- pgAdmin is now at version 9.2.
- The Timescaledb extension is at version 2.18.2 for PG 17, 16, 15, and 14.

Fixes

- Standby clusters now shut down properly when `spec.shutdown` is set to `true`.
- Validation has been added to ensure only cloud-based repos can be used with `spec.datasource.pgbackrest`.
- Changed the pgAdmin readinessProbe path to ensure proper cleanup of pgAdmin session files.

Note: After upgrading an existing pgAdmin pod, it is safe to manually delete any old session files that are not automatically deleted by the cleanup script.

5.7.4

Changes

- PostgreSQL versions 17.4, 16.8, 15.12, 14.17 and 13.20 are now available.
- Citus 13.0.0 is now available for Postgres 14, 15, 16 and 17.
- pg_partman is now at version 5.2.4 for PG 17, 16, 15 and 14.
- PostGIS version 3.5.2 is now available for PG 17.
- The hypopg 1.4.1 extension is now available.
- The pg_jobmon 1.4.1 extension is now available.
- The pg_parquet 0.2.0 extension is now available.

Fixes

- When a backup schedule is removed the associated CronJob is now deleted.

5.7.3

Changes

- Tolerate broken replication while configuring PostgreSQL.
- Patroni is now at version 4.0.4.
- pgBackRest is now at version 2.54.1.
- Postgres Exporter is now at 0.16.0.
- The pg_cron extension is now at version 1.6.5.
- pg_partman is now at version 5.2.2 for PG 17, 16, 15 and 14.

Fixes

- A service account is now reconciled for the pgBackRest repo host to facilitate EKS IAM role integration. After upgrading, you will need to delete any manually and/or CPK-initiated backup Jobs or wait for your next scheduled backup to run.

5.7.2

Changes

- Continue when Postgres restore intentionally exits early multiple times
- Always pass a `--jobs` argument to `pg_upgrade`
- Patroni is now at version 3.3.5.
- pgBackrest is now at version 2.54.0
- pgAdmin is now at version 8.14.
- orafce is now at version 4.14.0.
- pgvector is now at version 0.8.0.
- The Timescaledb extension is at version 2.17.2 for PG 17, 16, 15, and 14.

5.7.1

Features

- The operator emits a warning event when a `postgrescluster` is using a major version of Postgres that is no longer receiving updates

Changes

- PostgreSQL versions 17.2, 16.6, 15.10, 14.15, 13.18, and 12.22 are now available.
- Patroni is now at version 3.3.4.

Fixes

- The `CrunchyBridgeCluster.spec.secret` field is now required.

5.7.0

Features

- Asynchronous archiving by default. CPK will take control of the spool-path. If you have set the spool-path in the `backups.pgbackrest.global` section of your spec, remove that setting after upgrading. You can also delete that directory. If you would like to opt out of asynchronous archiving, set `spec.backups.pgbackrest.global.archive-async: "n"`. After upgrading, a new log will be introduced to track WAL archiving at `pgdata/pgbackrest/log/db-archive-push-async.log`.
- You can now enable backups from replicas within your pgBackRest configuration. Ensure you have at least one Postgres replica available, and then set `spec.backups.pgbackrest.global.backup-standby: "y"`

- You can now disable backups when provisioning new Postgres cluster by omitting the **backups** section from your **PostgresCluster** spec.
- You can now use Kerberos authentication with pgAdmin4 deployments created via the **PGAdmin** API.
- Liveness and readiness probes are now enabled by default when the operator is run. Additionally, all CPK installers have been updated to use these probes when creating the operator Deployment.
- You can now make the operator highly available by adding one or more additional replicas to the **pgo** Deployment.
- You can now configure the operator to watch a certain subset of namespaces using the new **PGO_TARGET_NAMESPACES** environment variable. This means you can now configure the operator to watch one namespace, all namespaces, or a specific subset of namespaces.
- Postgres authentication against TLS-enabled LDAP servers is now easier. A custom certification authority (CA) certificate using the existing **spec.config.files** method now mounts a Secret containing the **ca.crt** file. See breaking changes and the user management section.
- You can now easily enable or disable CPK feature gates via **values.yaml** settings when installing CPK via Helm. Contributed by Daniel Holmes (@jaitaiwan)
- You can now leverage [Kubernetes Volume Snapshots](#) when cloning a **PostgresCluster**. Enable **VolumeSnapshots** feature gate in your operator installation, and then configure a **VolumeSnapshotClass** within the spec of your source **PostgresCluster** using **spec.backups.snapshots.volumeSnapshotClassName**. Now when you clone the **PostgresCluster**, a snapshot will be leveraged to reduce the overall time to create and initialize the clone.

Breaking Changes

- SSL certificates present in the base image are no longer exposed directly. TLS-based LDAP connections now require you to specify the certificates to be used for TLS connections. TLS authentication to LDAP servers will break for users that were relying on certificates present in the base image. See the user management section for details.

Changes

- PostgreSQL version 17.0 is now available.
- PostGIS versions 3.4.3 is now available.
- Patroni is now at version 3.3.3.
- pgBackrest is now at version 2.53.1.
- pgBouncer is now at version 1.23.1.
- pgMonitor is now at version 5.1.1.
- pgAdmin is now at version 8.12.
- The pgAudit 17.0 extension is now available.
- The pg_cron extension is now at version 1.6.4.
- The pgvector extension is now at version 0.7.4.
- The pgnodemx extension is now at version 1.7.
- The TimescaleDB extension is at version 2.17.0 for PG 17, 16, 15, and 14.
- pgAdmin and pgBackRest images have **tar** as required by the **kubect1 cp** command.

- The `AutoCreateUserSchema` feature gate now defaults to `true`.

Fixes

- The `externalTrafficPolicy` is now properly configured for the primary, replica, PgBouncer and pgAdmin Services.

Crunchy Postgres for Kubernetes 5.6.x Release notes

Release notes for each of the 5.6.x releases.

Component versions

Crunchy Postgres for Kubernetes		Postgres	pgBackRest	pgbouncer	Patroni	pgadmin
5.6.8	16.9	2.54.2	1.24	4.0.4	4.30, 9.2	
5.6.7	16.8	2.54.2	1.24	4.0.4	4.30, 9.2	
5.6.6	16.8	2.54.1	1.23	4.0.4	4.30, 8.14	
5.6.5	16.6	2.54.1	1.23	4.0.4	4.30, 8.14	
5.6.4	16.6	2.54.0	1.23	3.3.5	4.30, 8.14	
5.6.3	16.6	2.53.1	1.23	3.3.4	4.30, 8.12	
5.6.2	16.4	2.53.1	1.23	3.3.3	4.30, 8.12	
5.6.1	16.4	2.52.1	1.22	3.1.2	4.30, 8.10	
5.6.0	16.3	2.51	1.22	3.1.2	4.30, 8.6	

Postgres extension versions

Crunchy Postgres for Kubernetes	PostGIS	pgRouting	pgaudit	pg_cron	pg_partman	pgnodemx	set_user	wal2json	TimescaleDB	Bacpac	pgvector			
5.6.8	3.0.11 (earliest)	3.4.4 (latest)	3.6 (earliest)	3.4.2 (latest)	3.2 (earliest)	17.0 (latest)	1.9	5.2.4	1.7	4.1.0	2.6	2.19.3	4.14.2	0.8.0
5.6.7	3.0.11 (earliest)	3.4.4 (latest)	3.6 (earliest)	3.4.2 (latest)	3.2 (earliest)	17.0 (latest)	1.9	5.2.4	1.7	4.1.0	2.6	2.18.2	4.14.0	0.8.0
5.6.6	3.0.11 (earliest)	3.4.4 (latest)	3.6 (earliest)	3.4.2 (latest)	3.2 (earliest)	17.0 (latest)	1.9	5.2.4	1.7	4.1.0	2.6	2.17.2	4.14.0	0.8.0
5.6.5	3.0.11 (earliest)	3.4.4 (latest)	3.6 (earliest)	3.4.2 (latest)	3.2 (earliest)	17.0 (latest)	1.9	5.2.2	1.7	4.1.0	2.6	2.17.2	4.14.0	0.8.0
5.6.4	3.0.11 (earliest)	3.4.3 (latest)	3.6 (earliest)	3.4.2 (latest)	3.3 (earliest)	16.0 (latest)	1.9	4.5.1.0	1.7	4.1.0	2.6	2.17.2	4.14.0	0.8.0
5.6.3	2.5.11 (earliest)	3.4.3 (latest)	3.6 (earliest)	3.4.2 (latest)	3.3 (earliest)	16.0 (latest)	1.9	4.5.1.0	1.7	4.1.0	2.6	2.17.0	4.10.3	0.7.4
5.6.2	2.5.11 (earliest)	3.4.3 (latest)	3.6 (earliest)	3.4.2 (latest)	3.3 (earliest)	16.0 (latest)	1.9	4.5.1.0	1.7	4.1.0	2.6	2.17.0	4.10.3	0.7.4
5.6.1	2.5.11 (earliest)	3.4.2 (latest)	3.6 (earliest)	3.4.2 (latest)	3.3 (earliest)	16.0 (latest)	1.9	2.5.1.0	1.6	4.0.1	2.5	2.15.3	4.10.3	0.7.3
5.6.0	2.5.11 (earliest)	3.4.2 (latest)	3.6 (earliest)	3.4.2 (latest)	3.3 (earliest)	16.0 (latest)	1.9	2.5.1.0	1.6	4.0.1	2.5	2.14.2	4.9.4	0.7.0

A bold version number indicates that the component version was updated in latest release.

5.6.8

Changes

- PostgreSQL versions 17.5, 16.9, 15.13, 14.18 and 13.21 are now available.
- The pgBouncer is now at version 1.24.1.
- The Citus 13.0.3 is now available for Postgres 15, 16 and 17.
- The orafce is now at version 4.14.2.
- The pg_parquet 0.4.0 extension is now available.
- The Timescaledb extension is at version 2.19.3 for PG 17, 16, 15, and 14.

Fixes

- PostgresCluster now allows the `--target-timeline` restore option

5.6.7

Breaking Changes

- Component images are now tagged with a consistent suffix. Images with the same suffix are compatible with each other. See the [Container Tags](#) section of the documentation for more details.

Changes

- Postgres Exporter is now at 0.17.1.
- pgAdmin is now at version 9.2.
- The Timescaledb extension is at version 2.18.2 for PG 17, 16, 15, and 14.

Fixes

- Validation has been added to ensure only cloud-based repos can be used with `spec.datasource.pgbackrest`.

5.6.6

Changes

- PostgreSQL versions 16.8, 15.12, 14.17 and 13.20 are now available.
- Citus 13.0.0 is now available for Postgres 14, 15, 16 and 17.
- pg_partman is now at version 5.2.4 for PG 17, 16, 15 and 14.
- The hypopg 1.4.1 extension is now available.
- The pg_jobmon 1.4.1 extension is now available.
- The pg_parquet 0.2.0 extension is now available.

Fixes

- When a backup schedule is removed the associated CronJob is now deleted.

5.6.5

Changes

- Tolerate broken replication while configuring PostgreSQL.
- Patroni is now at version 4.0.4.
- pgBackRest is now at version 2.54.1.
- Postgres Exporter is now at 0.16.0.
- The pg_cron extension is now at version 1.6.5.
- pg_partman is now at version 5.2.2 for PG 17, 16, 15 and 14.

5.6.4

Changes

- Continue when Postgres restore intentionally exits early multiple times
- Always pass a `--jobs` argument to `pg_upgrade`
- Patroni is now at version 3.3.5.
- pgBackrest is now at version 2.54.0
- pgAdmin is now at version 8.14.
- orafce is now at version 4.14.0.
- pgvector is now at version 0.8.0.
- The Timescaledb extension is at version 2.17.2 for PG 17, 16, 15, and 14.

5.6.3

Features

- The operator emits a warning event when a `postgrescluster` is using a major version of Postgres that is no longer receiving updates

Changes

- PostgreSQL versions 16.6, 15.10, 14.15, 13.18, and 12.22 are now available.
- Patroni is now at version 3.3.4.

Fixes

- The `CrunchyBridgeCluster.spec.secret` field is now required.

5.6.2

Features

- You can now easily enable or disable CPK feature gates via `values.yaml` settings when installing CPK via Helm. Contributed by Daniel Holmes (@jaitaiwan)

Changes

- PostGIS version 3.4.3 is now available.
- Patroni is now at version 3.3.3.
- pgBackrest is now at version 2.53.1.
- pgBouncer is now at version 1.23.1.
- pgMonitor is now at version 5.1.1.
- pgAdmin is now at version 8.12.
- The `pg_cron` extension is now at version 1.6.4.
- The `pgvector` extension is now at version 0.7.4.
- The `pgnodemx` extension is now at version 1.7.
- The TimescaleDB extension is at version 2.17.0 for PG 17, 16, 15, and 14.
- pgAdmin and pgBackRest images have `tar` as required by the `kubect1 cp` command.
- The `AutoCreateUserSchema` feature gate now defaults to `true`.

Fixes

- The `externalTrafficPolicy` is now properly configured for the primary, replica, PgBouncer and pgAdmin Services.
- Standalone pgAdmin failed in certain ARM environments

5.6.1

Features

- Use the `postgres-operator.crunchydata.com/autoCreateUserSchema=true` annotation to automatically create a schema for any Postgres users defined via `spec.users`. With this setting enabled, CPK creates a writable schema for each user (avoiding the `PUBLIC` schema, which Postgres 15 secured against unintended writes).

Changes

- PostgreSQL versions 16.4, 15.8, 14.13, 13.16, and 12.20 are now available.
- The `pgvector` extension is now at version 0.7.3.
- The `orafce` extension is now at version 4.10.3.
- The TimescaleDB extension is at version 2.15.3 for PG 16, 15, and 14.
- When migrating from Timescale DB 2.14.x, you must run [this SQL script](#) after you run `ALTER EXTENSION`. For more details, see the following pull request [#6797](#).

5.6.0

Features

- Configure your **PostgresCluster** to automatically expand Postgres data volume when additional database storage is needed.
- pgAdmin updates
- Enable TLS for pgAdmin deployments using custom TLS certificates.
- Use Postgres as the backend for pgAdmin deployments.
- Have PGO reconcile a pgAdmin Service by defining a service name in your **PGAdmin** spec.
- Select a **PostgresCluster** by name in your **PGAdmin** spec.
- Manage pgAdmin users via the **PGAdmin** spec
- Set passwords declaratively for users defined under **spec.user**.
- Configure the service type for the Postgres replica service.
- Provision Crunchy Bridge clusters using the new **CrunchyBridgeCluster** API.

Changes

- **SeccompProfile** is now set to RuntimeDefault in all Pods.
- The PGAdmin API now utilizes Gunicorn as the web server for any pgAdmin deployments.
- Attempts to use the **PASSWORD** option in **spec.users.options** will be rejected.

Fixes

- **StatefulSets** (pgAdmin and pgBackRest repo hosts) will now recover from a bad rollout.
- Various spelling fixes. Contributed by Josh Soref (@jsoref)

Crunchy Postgres for Kubernetes 5.5.x Release notes

Release notes for each of the 5.5.x releases.

Component versions

Crunchy Postgres for Kubernetes		Postgres	pgBackRest	pgbouncer	Patroni	pgadmin
5.5.10	16.9	2.54.2	1.24	4.0.4	4.30, 9.2	
5.5.9	16.8	2.54.2	1.24	4.0.4	4.30, 9.2	
5.5.8	16.8	2.54.1	1.23	4.0.4	4.30, 8.14	
5.5.7	16.6	2.54.1	1.23	4.0.4	4.30, 8.14	
5.5.6	16.6	2.54.0	1.23	3.3.5	4.30, 8.14	
5.5.5	16.6	2.53.1	1.23	3.3.4	4.30, 8.12	
5.5.4	16.4	2.53.1	1.23	3.3.3	4.30, 8.12	
5.5.3	16.4	2.52.1	1.22	3.1.2	4.30, 8.10	
5.5.2	16.3	2.51	1.22	3.1.2	4.30, 8.6	
5.5.1	16.2	2.49	1.21	3.1.2	4.30, 7.8	
5.5.0	16.1	2.47	1.21	3.1.1	4.30, 7.8	

Postgres extension versions

Crunchy Postgres for Kubernetes	PostGIS	pgRouting	pgaudit	pg_cron	pg_partman	pgnodemx	set_user	wal2json	TimescaleDB	orafce	pgvector
5.5.10 3.0.11 (earliest)	3.4.4 (latest)	3.0.6 (earliest)	3.4.2 (latest)	1.5.2 (earliest)	17.0 (latest)	1.9.5	5.2.4	1.7	4.1.0 2.6	2.19.3	4.14.2 0.8.0
5.5.9 3.0.11 (earliest)	3.4.4 (latest)	3.0.6 (earliest)	3.4.2 (latest)	1.5.2 (earliest)	17.0 (latest)	1.9.5	5.2.4	1.7	4.1.0 2.6	2.18.2	4.14.0 0.8.0
5.5.8 3.0.11 (earliest)	3.4.4 (latest)	3.0.6 (earliest)	3.4.2 (latest)	1.5.2 (earliest)	17.0 (latest)	1.9.5	5.2.4	1.7	4.1.0 2.6	2.17.2	4.14.0 0.8.0
5.5.7 3.0.11 (earliest)	3.4.4 (latest)	3.0.6 (earliest)	3.4.2 (latest)	1.5.2 (earliest)	17.0 (latest)	1.9.5	5.2.2	1.7	4.1.0 2.6	2.17.2	4.14.0 0.8.0
5.5.6 3.0.11 (earliest)	3.4.3 (latest)	3.0.3 (earliest)	3.4.2 (latest)	1.4.3 (earliest)	16.0 (latest)	1.9.4	5.1.0	1.7	4.1.0 2.6	2.17.2	4.14.0 0.8.0
5.5.5 2.5.11 (earliest)	3.4.3 (latest)	3.0.3 (earliest)	3.4.2 (latest)	1.4.3 (earliest)	16.0 (latest)	1.9.4	5.1.0	1.7	4.1.0 2.6	2.17.0	4.10.3 0.7.4
5.5.4 2.5.11 (earliest)	3.4.3 (latest)	3.0.3 (earliest)	3.4.2 (latest)	1.4.3 (earliest)	16.0 (latest)	1.9.4	5.1.0	1.7	4.1.0 2.6	2.17.0	4.10.3 0.7.4
5.5.3 2.5.11 (earliest)	3.4.2 (latest)	3.0.3 (earliest)	3.4.2 (latest)	1.4.3 (earliest)	16.0 (latest)	1.9.2	5.1.0	1.6	4.0.1 2.5	2.15.3	4.10.3 0.7.3
5.5.2 2.5.11 (earliest)	3.4.2 (latest)	3.0.3 (earliest)	3.4.2 (latest)	1.4.3 (earliest)	16.0 (latest)	1.9.2	5.1.0	1.6	4.0.1 2.5	2.14.2	4.9.4 0.7.0
5.5.1 2.5.9 (earliest)	3.4.0 (latest)	3.0.3 (earliest)	3.4.2 (latest)	1.4.3 (earliest)	16.0 (latest)	1.9.0	5.0.1	1.6	4.0.1 2.5	2.13.0	4.9.1 0.6.0
5.5.0 2.4.10 (earliest)	3.4.0 (latest)	3.0.3 (earliest)	3.4.2 (latest)	1.4.3 (earliest)	16.0 (latest)	1.9.0	5.0.0	1.6	4.0.1 2.5	2.12.2	4.7.0 0.5.1

A bold version number indicates that the component version was updated in latest release.

5.5.10

Changes

- PostgreSQL versions 17.5, 16.9, 15.13, 14.18 and 13.21 are now available.
- The pgBouncer is now at version 1.24.1.
- The Citus 13.0.3 is now available for Postgres 15, 16 and 17.
- The orafce is now at version 4.14.2.
- The pg_parquet 0.4.0 extension is now available.
- The Timescaledb extension is at version 2.19.3 for PG 17, 16, 15, and 14.

Fixes

- PostgresCluster now allows the `--target-timeline` restore option

5.5.9

Breaking Changes

- Component images are now tagged with a consistent suffix. Images with the same suffix are compatible with each other. See the [Container Tags](#) section of the documentation for more details.

Changes

- Postgres Exporter is now at 0.17.1.

- pgAdmin is now at version 9.2.
- The Timescaledb extension is at version 2.18.2 for PG 17, 16, 15, and 14.

5.5.8

Changes

- PostgreSQL versions 16.8, 15.12, 14.17 and 13.20 are now available.
- Citus 13.0.0 is now available for Postgres 14, 15, 16 and 17.
- pg_partman is now at version 5.2.4 for PG 17, 16, 15 and 14.
- The hypopg 1.4.1 extension is now available.
- The pg_jobmon 1.4.1 extension is now available.
- The pg_parquet 0.2.0 extension is now available.

Fixes

- When a backup schedule is removed the associated CronJob is now deleted.

5.5.7

Changes

- Tolerate broken replication while configuring PostgreSQL.
- Patroni is now at version 4.0.4.
- pgBackRest is now at version 2.54.1.
- Postgres Exporter is now at 0.16.0.
- The pg_cron extension is now at version 1.6.5.
- pg_partman is now at version 5.2.2 for PG 17, 16, 15 and 14.

5.5.6

Changes

- Continue when Postgres restore intentionally exits early multiple times
- Patroni is now at version 3.3.5.
- pgBackrest is now at version 2.54.0.
- pgAdmin is now at version 8.14.
- orafce is now at version 4.14.0.
- pgvector is now at version 0.8.0.
- The Timescaledb extension is at version 2.17.2 for PG 17, 16, 15, and 14.

5.5.5

Features

- The operator emits a warning event when a `postgrescluster` is using a major version of Postgres that is no longer receiving updates

Changes

- PostgreSQL versions 16.6, 15.10, 14.15, 13.18, and 12.22 are now available.
- Patroni is now at version 3.3.4.

5.5.4

Features

- You can now easily enable or disable CPK feature gates via values.yaml settings when installing CPK via Helm. Contributed by Daniel Holmes (@jaitaiwan)

Changes

- PostGIS version 3.4.3 is now available.
- Patroni is now at version 3.3.3.
- pgBackrest is now at version 2.53.1.
- pgBouncer is now at version 1.23.1.
- pgMonitor is now at version 5.1.1.
- pgAdmin is now at version 8.12.
- The pg_cron extension is now at version 1.6.4.
- The pgvector extension is now at version 0.7.4.
- The pgnodemx extension is now at version 1.7.
- The TimescaleDB extension is at version 2.17.0 for PG 17, 16, 15, and 14.
- pgAdmin and pgBackRest images have `tar` as required by the `kubect1 cp` command.

Fixes

- Standalone pgAdmin failed in certain ARM environments

5.5.3

Changes

- PostgreSQL versions 16.4, 15.8, 14.13, 13.16, and 12.20 are now available.

- The pgvector extension is now at version 0.7.3.
- The orafce extension is now at version 4.10.3.
- The TimescaleDB extension is at version 2.15.3 for PG 16, 15, and 14. • When migrating from Timescale DB 2.14.x you must run [this SQL script](#) after you run **ALTER EXTENSION** For more details, see the following pull request [#6797](#).

5.5.2

Features

- Warn when a **PASSWORD** option is included in **spec.users.options**.
- pgAdmin v8 is now supported by the Namespace-Scoped PGAdmin API.

Changes

- PostgreSQL versions 16.3, 15.7, 14.12, 13.15, and 12.19 are now available.
- PostGIS versions 3.4.2, 3.3.6, 3.2.7, 3.1.11, 3.0.11, and 2.5.11 are now available.
- pgAdmin v8.6 is now available.
- pgBackRest is now at version 2.51.
- pgBouncer is now at version 1.22.1.
- The orafce extension is now at version 4.9.4.
- The pg_partman extension is now at version 5.1.0 for PG 16, 15 and 14.
- The pgvector extension is now at version 0.7.0.
- The TimescaleDB extension is now at version 2.14.2 for PG 16, 15, 14, and 13.
- The **postgres-operator** image now uses UBI Minimal.

Notable Security Fixes

Crunchy PostgreSQL 16.3-0, 15.7-0, and 14.12-0 include:

- [CVE-2024-4317](#) Restrict visibility of **pg_stats_ext** and **pg_stats_ext_exprs** entries to the table owner. These views failed to hide statistics for expressions that involve columns the accessing user does not have permission to read. View columns such as **most_common_vals** might expose security-relevant data. The potential interactions here are not fully clear, so in the interest of erring on the side of safety, make rows in these views visible only to the owner of the associated table. By itself, this fix will only fix the behavior in newly initdb'd database clusters. If you wish to apply this change in an existing cluster, you will need to do the following:
 - Find the SQL script **fix-CVE-2024-4317.sql** in the share directory of the PostgreSQL installation. In Crunchy Data's PostgreSQL 16 RPM packages, the script can be found in folder **/usr/pgsql-16/share/** after installing the **postgres16-server** RPM. Be sure to use the script appropriate to your PostgreSQL major version. If you do not see this file, either your version is not vulnerable (only v14-v16 are affected) or your minor version is too old to have the fix.
 - In each database of the cluster, run the **fix-CVE-2024-4317.sql** script as superuser. In psql this would look like **\i /usr/pgsql-16/share/fix-CVE-2024-4317.sql** (adjust the file path as appropriate). Any error probably indicates that you've used the wrong script version. It will not hurt to run the script more than once.
 - Do not forget to include the **template0** and **template1** databases, or the vulnerability will still exist in databases you create later. To fix **template0**, you'll need to temporarily make it accept connections.

Do that with: `ALTER DATABASE template0 WITH ALLOW_CONNECTIONS true;` and then after fixing `template0`, undo it with `ALTER DATABASE template0 WITH ALLOW_CONNECTIONS false;`

5.5.1

Fixes

- Only load `datasource.pgbackrest.configuration` when performing a cloud based restore.
- Queue an event based on instance Patroni 'master' role change
- The pgAdmin controller now owns any objects it creates
- pgAdmin can now be accessed from Kubernetes networks by default
- Allow numeric characters in pgAdmin config settings. Contributed by Roman Gherta (@rgherta).

Changes

- PostgreSQL versions 16.2, 15.6, 14.11, 13.14, and 12.18 are now available.
- pgBackRest is now at version 2.49.
- patroni is now at version 3.1.2.
- pgMonitor is now at version 4.11.
- The orafce extension is now at version 4.9.1.
- The pg_cron extension is now at version 1.6.2.
- The pg_partman extension is now at version 5.0.1 for PG 16, 15 and 14.
- The pgvector extension is now at version 0.6.0.
- The TimescaleDB extension is now available for PG 16. The extension is at version 2.13.0 for PG 16, 15, 14, and 13.

5.5.0

Features

- The monitoring stack has undergone a number of significant improvements in 5.5, including:
 - Transitioning the `crunchy-postgres-exporter` image into a component container, thereby decoupling it from the `postgres-operator`.
 - The ability to append custom exporter queries to the default queries provided by Crunchy Postgres for Kubernetes.
 - You can now monitor your standby clusters by editing the `ccp_monitoring` password.
 - Postgres 16 support!
- We added a new API for pgAdmin 4, which allows you to create a single pgAdmin 4 to manage multiple clusters in a namespace! This new API also comes with a new image containing the latest version of pgAdmin 4.

Changes

- When specified, the `citus` extension is loaded before other `shared_preload_libraries`.
- You can reduce metrics to those provided by pgMonitor by setting the `postgres-operator.crunchydata.com/postgres-exporter-collectors` annotation to `None`.

- PostgreSQL versions 16.1, 15.5, 14.10, 13.13, 12.17, and 11.22 are now available.
- As of February, 2023, public builds will offer the latest PG 16 and 15.
- pgBouncer is now at version 1.21.0.
- The orafce extension is now at version 4.7.0.
- The pg_partman extension is now at version 5.0.0 for PG 16, 15 and 14.
- The pgAudit16 extension is now at version 16.0.
- The pgvector extension is now at version 0.5.1.
- The TimescaleDB extension now at version 2.12.2 for PG 15, 14 and 13, version 2.11.2 for PG 12 and version 2.3.1 for PG 11.
- DNS names for the replica service have been added to the certificates generated for the PostgresCluster to facilitate TLS connections between pgBouncer and read replicas. Contributed by Scott Zelenka (@szelenka)

Crunchy Postgres for Kubernetes 5.4.x Release notes

Release notes for each of the 5.4.x releases.

Component versions

Crunchy Postgres for Kubernetes		Postgres	pgBackRest	pgbouncer	Patroni	pgadmin
5.4.9	16.6	2.53.1	1.23	3.3.4	4.30	
5.4.8	16.4	2.53.1	1.23	3.3.3	4.30	
5.4.7	16.4	2.52.1	1.22	3.1.2	4.30	
5.4.6	16.3	2.51	1.22	3.1.2	4.30	
5.4.5	16.2	2.49	1.21	3.1.2	4.30	
5.4.4	16.1	2.47	1.21	3.1.1	4.30	
5.4.3	16.0	2.47	1.19	3.1.1	4.30	
5.4.2	15.4	2.47	1.19	3.1.0	4.30	
5.4.1	15.3	2.45	1.19	2.1.7	4.30	
5.4.0	15.3	2.45	1.19	2.1.7	4.30	

Postgres extension versions

Crunchy Postgres for Kubernetes	PostGIS	PostGIS-3D	Routing	pgaudit	pg_cron	pg_partman	pgnodemon	set_user	val2json	TimescaleDB	orafce	pgvector		
5.4.9	2.5.11 (earliest)	3.4.3 (latest)	2.6.3 (earliest)	3.4.2 (latest)	2.4.3 (earliest)	16.0 (latest)	1.5.4	5.1.0	1.7	4.1.0	2.6	2.17.0	4.10.3	0.7.4
5.4.8	2.5.11 (earliest)	3.4.3 (latest)	2.6.3 (earliest)	3.4.2 (latest)	2.4.3 (earliest)	16.0 (latest)	1.5.4	5.1.0	1.7	4.1.0	2.6	2.17.0	4.10.3	0.7.4
5.4.7	2.5.11 (earliest)	3.4.2 (latest)	2.6.3 (earliest)	3.4.2 (latest)	2.4.3 (earliest)	16.0 (latest)	1.5.2	5.1.0	1.6	4.0.1	2.5	2.15.3	4.10.3	0.7.3
5.4.6	2.5.11 (earliest)	3.4.2 (latest)	2.6.3 (earliest)	3.4.2 (latest)	2.4.3 (earliest)	16.0 (latest)	1.5.2	5.1.0	1.6	4.0.1	2.5	2.14.2	4.9.4	0.7.0
5.4.5	2.5.9 (earliest)	3.4.0 (latest)	2.6.3 (earliest)	3.4.2 (latest)	2.4.3 (earliest)	16.0 (latest)	1.5.2	5.0.1	1.6	4.0.1	2.5	2.13.0	4.9.1	0.6.0
5.4.4	2.4.10 (earliest)	3.4.0 (latest)	2.6.3 (earliest)	3.4.2 (latest)	2.4.3 (earliest)	16.0 (latest)	1.5.0	5.0.0	1.6	4.0.1	2.5	2.12.2	4.7.0	0.5.1

5.4.3	2.4.10 (earliest)	3.4.0 (latest)	2.6.3 (earliest)	3.4.2 (latest)	1.3.4 (earliest)	1.7.0 (latest)	1.6.0	4.7.4	1.6	4.0.1	2.5	2.11.2	4.6.1	0.4.4
5.4.2	2.4.10 (earliest)	3.3.2 (latest)	2.6.3 (earliest)	3.3.1 (latest)	1.3.4 (earliest)	1.7.0 (latest)	1.5.2	4.7.3	1.4	4.0.1	2.5	2.10.3	4.2.6	0.4.4
5.4.1	2.4.10 (earliest)	3.3.2 (latest)	2.6.3 (earliest)	3.3.1 (latest)	1.3.4 (earliest)	1.7.0 (latest)	1.5.2	4.7.3	1.4	4.0.1	2.5	2.10.3	4.2.6	0.4.4
5.4.0	2.4.10 (earliest)	3.3.2 (latest)	2.6.3 (earliest)	3.3.1 (latest)	1.3.4 (earliest)	1.7.0 (latest)	1.5.2	4.7.3	1.4	4.0.1	2.5	2.10.3	4.2.6	0.4.4

A bold version number indicates that the component version was updated in latest release.

5.4.9

Features

- The operator emits a warning event when a `postgrescluster` is using a major version of Postgres that is no longer receiving updates

Changes

- PostgreSQL versions 16.6, 15.10, 14.15, 13.18, and 12.22 are now available.
- Patroni is now at version 3.3.4.

5.4.8

Features

- You can now easily enable or disable CPK feature gates via `values.yaml` settings when installing CPK via Helm. Contributed by Daniel Holmes (@jaitaiwan)

Changes

- PostGIS version 3.4.3 is now available.
- Patroni is now at version 3.3.3.
- pgBackrest is now at version 2.53.1.
- pgBouncer is now at version 1.23.1.
- pgMonitor is now at version 5.1.1.
- The `pg_cron` extension is now at version 1.6.4.
- The `pgvector` extension is now at version 0.7.4.
- The `pgnodemx` extension is now at version 1.7.
- The TimescaleDB extension is at version 2.17.0 for PG 17, 16, 15, and 14.
- pgAdmin and pgBackRest images have `tar` as required by the `kubect1 cp` command.

5.4.7

Changes

- PostgreSQL versions 16.4, 15.8, 14.13, 13.16, and 12.20 are now available.
- pgBackRest is now at version 2.53.
- The pgvector extension is now at version 0.7.3.
- The orafce extension is now at version 4.10.3.
- The TimescaleDB extension is at version 2.15.3 for PG 16, 15, and 14. • When migrating from Timescale DB 2.14.x you must run [this SQL script](#) after you run `ALTER EXTENSION` For more details, see the following pull request [#6797](#).

5.4.6

Features

- Warn when a `PASSWORD` option is included in `spec.users.options`.

Changes

- PostgreSQL versions 16.3, 15.7, 14.12, 13.15, and 12.19 are now available.
- PostGIS versions 3.4.2, 3.3.6, 3.2.7, 3.1.11, 3.0.11, and 2.5.11 are now available.
- pgBackRest is now at version 2.51.
- pgBouncer is now at version 1.22.1.
- The orafce extension is now at version 4.9.4.
- The pg_partman extension is now at version 5.1.0 for PG 16, 15 and 14.
- The pgvector extension is now at version 0.7.0.
- The TimescaleDB extension is now at version 2.14.2 for PG 16, 15, 14, and 13.
- The `postgres-operator` image now uses UBI Minimal.

Notable Security Fixes

Crunchy PostgreSQL 16.3-0, 15.7-0, and 14.12-0 include:

- [CVE-2024-4317](#) Restrict visibility of `pg_stats_ext` and `pg_stats_ext_exprs` entries to the table owner. These views failed to hide statistics for expressions that involve columns the accessing user does not have permission to read. View columns such as `most_common_vals` might expose security-relevant data. The potential interactions here are not fully clear, so in the interest of erring on the side of safety, make rows in these views visible only to the owner of the associated table. By itself, this fix will only fix the behavior in newly initdb'd database clusters. If you wish to apply this change in an existing cluster, you will need to do the following:
 - Find the SQL script `fix-CVE-2024-4317.sql` in the share directory of the PostgreSQL installation. In Crunchy Data's PostgreSQL 16 RPM packages, the script can be found in folder `/usr/pgsql-16/share/` after installing the `postgres16-server` RPM. Be sure to use the script appropriate to your PostgreSQL major version. If you do not see this file, either your version is not vulnerable (only v14-v16 are affected) or your minor version is too old to have the fix.
 - In each database of the cluster, run the `fix-CVE-2024-4317.sql` script as superuser. In psql this would look like `\i /usr/pgsql-16/share/fix-CVE-2024-4317.sql` (adjust the file path as appropriate). Any error probably indicates that you've used the wrong script version. It will not hurt to run the script more than once.
 - Do not forget to include the `template0` and `template1` databases, or the vulnerability will still exist in databases you create later. To fix `template0`, you'll need to temporarily make it accept connections.

Do that with: `ALTER DATABASE template0 WITH ALLOW_CONNECTIONS false;` and then after fixing `template0`, undo it with `ALTER DATABASE template0 WITH ALLOW_CONNECTIONS false;`

5.4.5

Fixes

- Only load `datasource.pgbackrest.configuration` when performing a cloud based restore.
- Queue an event based on instance Patroni 'master' role change
- Make Standalone PgAdmin controller the owner of the objects it creates
- Allow numeric characters in pgAdmin config settings. Contributed by Roman Gherta (@rgherta).

Changes

- PostgreSQL versions 16.2, 15.6, 14.11, 13.14, and 12.18 are now available.
- pgBackRest is now at version 2.49.
- patroni is now at version 3.1.2.
- pgMonitor is now at version 4.11.
- The orafce extension is now at version 4.9.1.
- The pg_cron extension is now at version 1.6.2.
- The pg_partman extension is now at version 5.0.1 for PG 16, 15 and 14.
- The pgvector extension is now at version 0.6.0.
- The TimescaleDB extension is now available for PG 16. The extension is at version 2.13.0 for PG 16, 15, 14, and 13.

5.4.4

Changes

- PostgreSQL versions 16.1, 15.5, 14.10, 13.13, 12.17, and 11.22 are now available.
- pgBouncer is now at version 1.21.0.
- The orafce extension is now at version 4.7.0.
- The pg_partman extension is now at version 5.0.0 for PG 16, 15 and 14.
- The pgAudit16 extension is now at version 16.0.
- The pgvector extension is now at version 0.5.1.
- The TimescaleDB extension now at version 2.12.2 for PG 15, 14 and 13, version 2.11.2 for PG 12 and version 2.3.1 for PG 11.

5.4.3

Changes

- PostgreSQL version 16.0 is now available. This release of PostgreSQL 16 does not include the TimescaleDB extension.
- PostGIS versions 3.4.0, 3.3.4 are now available.
- Patroni is now at version 3.1.1.
- pgMonitor is now at version 4.10.
- The orafce extension is now at version 4.6.1.
- The pg_cron extension is now at version 1.6.0.
- The pg_partman extension is now at version 4.7.4.
- The pgAudit Analyze extension is now at version 1.0.9.
- The pgnodemx extension is now at version 1.6.
- The pgRouting extension is now at version 3.4.2 for PG 16, and version 3.3.4 for PG 15 & 14.
- pscycopg is now at version 2.9.7.
- The TimescaleDB extension is now at version 2.11.2.

5.4.2

Changes

- PostgreSQL versions 15.4, 14.9, 13.12, 12.16, and 11.21 are now available.
- Patroni is now at version 3.1.0.
- pgBackrest is now at version 2.47.
- pgBouncer is now at version 1.19.1.

5.4.1

Fixes

- Backup jobs for S3-compatible object storage repositories would fail with a message about config hash mismatch. This is now fixed.
- PGO now prevents empty image values from impacting a PostgresCluster. With this change, a warning event explains that the cluster will be updated once the necessary images are defined. PostgresClusters with images defined continue to reconcile normally.
- Recovering from missing images during a Postgres major version upgrade is easier now. Conditions on PGUpgrade are more clearly defined, and new validation checks the upgrade image field.

5.4.0

Features

- The **PGUpgrade** API has been added to Crunchy Postgres for Kubernetes OLM installer.
- The **pggo-upgrade** deployment is no longer needed and [can be removed](#).

- Added the ability to add volumes for **tablespace** support (guarded by feature gate)
- ARM images are now available• [PostgreSQL](#) versions 15.3, 14.8, 13.11 are now available.• [PostGIS](#) versions 3.1.8, 3.2.4 & 3.3.2 are now available.
- The [pgvector](#) extension, version 0.4.4, is now available.

Changes

- Trivy has been integrated into Continuous Integration pipelines for the detection and resolution of CVE's within Go binaries and container image builds.
- Major Upgrade doc change providing clarity around deleting old WAL files. Contributed by Stefan Midjich (@stemid).
- Documentation update to bring our Keycloak example into alignment with the latest version. Contributed by David Jeffers (@dajeffers).
- The **pgaudit_analyze** tool is deprecated and may be removed in a future release.

Fixes

- The major PG upgrades documentation now includes the proper guidance/instructions for updating the **pgAudit** extension.
- PostgresClusters that do not request huge pages can now initialize and be restored on nodes with huge pages. Kubernetes container runtimes still configure cgroups incorrectly in these cases, but **initdb** no longer crashes.
- The custom TLS documentation now includes the proper information for the Common Name for the certificates for both the **customTLSSecret** and the **customReplicationTLSSecret**.

Crunchy Postgres for Kubernetes 5.3.x Release Notes

Release notes for each of the 5.3.x releases.

Component versions

Crunchy Postgres for Kubernetes		Postgres	pgBackRest	pgbouncer	Patroni	pgadmin
5.3.9	15.8	2.52.1	1.22	3.1.2		4.30
5.3.8	15.7	2.51	1.22	3.1.2		4.30
5.3.7	15.6	2.49	1.21	3.1.2		4.30
5.3.6	15.5	2.47	1.21	3.1.1		4.30
5.3.5	15.4	2.47	1.19	3.1.1		4.30
5.3.4	15.4	2.47	1.19	3.1.0		4.30
5.3.3	15.3	2.45	1.19	2.1.7		4.30
5.3.2	15.3	2.45	1.19	2.1.7		4.30
5.3.1	15.2	2.40	1.18	2.1.7		4.30
5.3.0	15.1	2.40	1.17	2.1.3		4.30

Postgres extension versions

Crunchy Postgres for Kubernetes	PostGIS	Routing	pgaudit	pg_cron	pg_partman	pgnodemon	set_user_val2	jsonb	TimescaleDB	orafce	pgvector
5.3.9 2.5.11 (earliest)	3.3.6 (latest)	3 (earliest)	3.3.4 (latest)	3 (earliest)	1.7.0 (latest)	2 5.1.0 1.6 4.0.1 2.5	2.15.3	4.10.3	0.7.3		
5.3.8 2.5.11 (earliest)	3.3.6 (latest)	3 (earliest)	3.3.4 (latest)	3 (earliest)	1.7.0 (latest)	2 5.1.0 1.6 4.0.1 2.5	2.14.2	4.9.4	0.7.0		
5.3.7 2.5.9 (earliest)	3.3.4 (latest)	3 (earliest)	3.3.4 (latest)	3 (earliest)	1.7.0 (latest)	2 5.0.1 1.6 4.0.1 2.5	2.13.0	4.9.1	0.6.0		
5.3.6 2.4.10 (earliest)	3.3.4 (latest)	3 (earliest)	3.3.4 (latest)	3 (earliest)	1.7.0 (latest)	2 5.0.0 1.6 4.0.1 2.5	2.12.2	4.7.0	0.4.4		
5.3.5 2.4.10 (earliest)	3.3.4 (latest)	3 (earliest)	3.3.4 (latest)	3 (earliest)	1.7.0 (latest)	2 4.7.4 1.6 4.0.1 2.5	2.11.2	4.6.1	0.4.4		
5.3.4 2.4.10 (earliest)	3.2.2 (latest)	3 (earliest)	3.3.1 (latest)	3 (earliest)	1.7.0 (latest)	2 4.7.3 1.4 4.0.1 2.5	2.10.3	4.2.6	0.4.4		
5.3.3 2.4.10 (earliest)	3.2.2 (latest)	3 (earliest)	3.3.1 (latest)	3 (earliest)	1.7.0 (latest)	2 4.7.3 1.4 4.0.1 2.5	2.10.3	4.2.6	0.4.4		
5.3.2 2.4.10 (earliest)	3.2.2 (latest)	3 (earliest)	3.3.1 (latest)	3 (earliest)	1.7.0 (latest)	2 4.7.3 1.4 4.0.1 2.5	2.10.3	4.2.6			
5.3.1 2.4.10 (earliest)	3.2.2 (latest)	3 (earliest)	3.3.1 (latest)	3 (earliest)	1.7.0 (latest)	2 4.7.2 1.3.0 4.0.1 2.5	2.9.2	4.1.1			
5.3.0 2.3 (earliest)	3.2.1 (latest)	3 (earliest)	3.3.1 (latest)	3 (earliest)	1.7.0 (latest)	2 4.7.1 1.3.0 3.0.0 2.5	2.8.1	3.25.1			

A bold version number indicates that the component version was updated in latest release.

5.3.9

Changes

- PostgreSQL versions 16.4, 15.8, 14.13, 13.16, and 12.20 are now available.
- pgBackRest is now at version 2.53.
- The pgvector extension is now at version 0.7.3.
- The orafce extension is now at version 4.10.3.
- The TimescaleDB extension is at version 2.15.3 for PG 16, 15, and 14. • When migrating from Timescale DB 2.14.x you must run [this SQL script](#) after you run **ALTER EXTENSION**For more details, see the following pull request [#6797](#).

5.3.8

Features

- Warn when a **PASSWORD** option is included in **spec.users.options**.

Changes

- PostgreSQL versions 16.3, 15.7, 14.12, 13.15, and 12.19 are now available.
- PostGIS versions 3.4.2, 3.3.6, 3.2.7, 3.1.11, 3.0.11, and 2.5.11 are now available.
- pgBackRest is now at version 2.51.
- pgBouncer is now at version 1.22.1.
- The orafce extension is now at version 4.9.4.
- The pg_partman extension is now at version 5.1.0 for PG 16, 15 and 14.

- The pgvector extension is now at version 0.7.0.
- The TimescaleDB extension is now at version 2.14.2 for PG 16, 15, 14, and 13.
- The **postgres-operator** image now uses UBI Minimal.

Notable Security Fixes

Crunchy PostgreSQL 16.3-0, 15.7-0, and 14.12-0 include:

- [CVE-2024-4317](#) Restrict visibility of **pg_stats_ext** and **pg_stats_ext_exprs** entries to the table owner. These views failed to hide statistics for expressions that involve columns the accessing user does not have permission to read. View columns such as **most_common_vals** might expose security-relevant data. The potential interactions here are not fully clear, so in the interest of erring on the side of safety, make rows in these views visible only to the owner of the associated table. By itself, this fix will only fix the behavior in newly initdb'd database clusters. If you wish to apply this change in an existing cluster, you will need to do the following:
 - Find the SQL script **fix-CVE-2024-4317.sql** in the share directory of the PostgreSQL installation. In Crunchy Data's PostgreSQL 16 RPM packages, the script can be found in folder **/usr/pgsql-16/share/** after installing the **postgresql16-server** RPM. Be sure to use the script appropriate to your PostgreSQL major version. If you do not see this file, either your version is not vulnerable (only v14-v16 are affected) or your minor version is too old to have the fix.
 - In each database of the cluster, run the **fix-CVE-2024-4317.sql** script as superuser. In psql this would look like **\i /usr/pgsql-16/share/fix-CVE-2024-4317.sql** (adjust the file path as appropriate). Any error probably indicates that you've used the wrong script version. It will not hurt to run the script more than once.
 - Do not forget to include the **template0** and **template1** databases, or the vulnerability will still exist in databases you create later. To fix **template0**, you'll need to temporarily make it accept connections. Do that with: **ALTER DATABASE template0 WITH ALLOW_CONNECTIONS true;** and then after fixing **template0**, undo it with **ALTER DATABASE template0 WITH ALLOW_CONNECTIONS false;**

5.3.7

Fixes

- Only load **datasource.pgbackrest.configuration** when performing a cloud based restore.
- Queue an event based on instance Patroni 'master' role change
- Allow numeric characters in pgAdmin config settings. Contributed by Roman Gherta (@rgherta).

Changes

- PostgreSQL versions 15.6, 14.11, 13.14, and 12.18 are now available.
- pgBackRest is now at version 2.49.
- patroni is now at version 3.1.2.
- The orafce extension is now at version 4.9.1.
- The pg_cron extension is now at version 1.6.2.
- The pg_partman extension is now at version 5.0.1 for PG 16, 15 and 14.
- The pgvector extension is now at version 0.6.0.
- The TimescaleDB extension is now available for PG 16. The extension is at version 2.13.0 for PG 16, 15, 14, and 13.

5.3.6

Changes

- PostgreSQL versions 15.5, 14.10, 13.13, 12.17, and 11.22 are now available.
- pgBouncer is now at version 1.21.0.
- The orafce extension is now at version 4.7.0.
- The pg_partman extension is now at version 5.0.0 for PG 15 and 14.
- The pgvector extension is now at version 0.5.1.
- The TimescaleDB extension now at version 2.12.2 for PG 15, 14 and 13, version 2.11.2 for PG 12 and version 2.3.1 for PG 11.

5.3.5

Changes

- Patroni is now at version 3.1.1.
- PostGIS version 3.3.4 is now available.
- The orafce extension is now at version 4.6.1.
- The pg_cron extension is now at version 1.6.0.
- The pg_partman extension is now at version 4.7.4.
- The pgAudit Analyze extension is now at version 1.0.9.
- The pgnodemx extension is now at version 1.6.
- The pgRouting extension is now at version 3.3.4 for PG 15 & 14.
- pscycopg is now at version 2.9.7.
- The TimescaleDB extension is now at version 2.11.2.

5.3.4

Changes

- PostgreSQL versions 15.4, 14.9, 13.12, 12.16, and 11.21 are now available.
- Patroni is now at version 3.1.0.
- pgBackrest is now at version 2.47.
- pgBouncer is now at version 1.19.1.

Fixes

- PostgresClusters that do not request huge pages can now be restored on nodes with huge pages.

5.3.3

Changes

- The `pgaudit_analyze` tool is deprecated and may be removed in a future release.

Fixes

- Backup jobs for S3-compatible object storage repositories would fail with a message about config hash mismatch. This is now fixed.

5.3.2

Fixes

- PostgresClusters that do not request huge pages can now initialize on nodes with huge pages. Kubernetes container runtimes still configure cgroups incorrectly in these cases, but `initdb` no longer crashes.

5.3.1

This release contains new component and Postgres versions, but no additional fixes or changes.

5.3.0

Features

- PostgreSQL 15 support.
- Enable TLS for the PostgreSQL exporter using the new `spec.monitoring.pgmonitor.exporter.customTLSCredentials` field.
- Configure pgBackRest for IPv6 environments using the `postgres-operator.crunchydata.com/pgbackrest-ip-version` annotation.
- Configure the [TTL](#) for pgBackRest backup Jobs.
- Use Helm's [OCI registry capability](#) to install Crunchy Postgres for Kubernetes.

Changes

- JIT is now explicitly disabled for the monitoring user, allowing users to opt-into using JIT elsewhere in the database without impacting exporter functionality. Contributed by Kirill Petrov (@chobostar).
- PGO now logs both `stdout` and `stderr` when running a SQL file referenced via `spec.databaseInitSQL` during database initialization. Contributed by Jeff Martin (@jmartin127).
- The `pgnodemx` and `pg_stat_statements` extensions are now automatically upgraded.

- The `postgres-startup` init container now logs an error message if the version of PostgreSQL installed in the image does not match the PostgreSQL version specified using `spec.postgresVersion`.
- Limit the monitoring user to local connections using SCRAM authentication. Contributed by Scott Zelenka (@szelenka)
- Skip a scheduled backup when the prior one is still running. Contributed by Scott Zelenka (@szelenka)
- The `dataSource.volumes` migration strategy had been improved to better handle `PGDATA` directories with invalid permissions and a missing `postgresql.conf` file.

Fixes

- A `psycopg2` error is no longer displayed when connecting to a database using pgAdmin 4.
- With the exception of the `--repo` option itself, PGO no longer prevents users from specifying pgBackRest options containing the string "repo" (e.g. `--repo1-retention-full`).
- PGO now properly filters Jobs by namespace when reconciling restore or data migrations Job, ensuring PostgresClusters with the same name can be created within different namespaces.
- The Major PostgreSQL Upgrades API (`PGUpgrade`) now properly handles clusters that have various extensions enabled.

Crunchy Postgres for Kubernetes 5.2.x Release Notes

Release notes for each of the 5.2.x releases.

Component versions

Crunchy Postgres for Kubernetes		Postgres	pgBackRest	pgbouncer	Patroni	pgadmin
5.2.8	14.11	2.49	1.21	3.1.2		4.30
5.2.7	14.10	2.47	1.21	3.1.1		4.30
5.2.6	14.9	2.47	1.19	3.1.1		4.30
5.2.5	14.9	2.47	1.19	3.1.0		4.30
5.2.4	14.8	2.45	1.19	2.1.7		4.30
5.2.3	14.8	2.45	1.19	2.1.7		4.30
5.2.2	14.7	2.41	1.18	2.1.7		4.30
5.2.1	14.6	2.40	1.17	2.1.3		4.30
5.2.0	14.5	2.40	1.17	2.1.3		4.30

Postgres extension versions

Crunchy Postgres for Kubernetes	PostGIS	PostGIS-Routing	pgaudit	pg_cron	pg_partman	pgnodemonset	useval2	jsonT	TimescaleDB	dbal	pgvector			
5.2.8	2.5.9 (earliest)	3.3.4 (latest)	2.6.3 (earliest)	3.3.4 (latest)	1.4.3 (earliest)	1.7.0 (latest)	1.6.2	5.0.1	1.6	4.0.1	2.5	2.13.0	4.9.1	0.6.0
5.2.7	2.4.10 (earliest)	3.3.4 (latest)	2.6.3 (earliest)	3.3.4 (latest)	1.4.3 (earliest)	1.7.0 (latest)	1.6.1	5.0.0	1.6	4.0.1	2.5	2.12.2	4.7.0	0.5.1
5.2.6	2.4.10 (earliest)	3.3.4 (latest)	2.6.3 (earliest)	3.3.4 (latest)	1.4.3 (earliest)	1.7.0 (latest)	1.6.1	4.7.4	1.6	4.0.1	2.5	2.11.2	4.6.1	0.4.4
5.2.5	2.3 (earliest)	3.3.2 (latest)	2.5 (earliest)	3.3.1 (latest)	1.4.3 (earliest)	1.7.0 (latest)	1.5.2	4.7.3	1.4	4.0.1	2.5	2.10.3	4.2.6	0.4.4

5.2.4	2.3 (earliest)	3.3.2 (latest)	3.3.3 (earliest)	3.3.1 (latest)	1.7.4 (earliest)	1.7.0 (latest)	4.7.2	4.7.3	1.4	4.0.1	2.5	2.10.3	4.2.6	0.4.4
5.2.3	2.3 (earliest)	3.3.2 (latest)	3.3.3 (earliest)	3.3.1 (latest)	1.7.4 (earliest)	1.7.0 (latest)	4.7.2	4.7.3	1.4	4.0.1	2.5	2.10.3	4.2.6	
5.2.2	2.3 (earliest)	3.1.8 (latest)	3.3.3 (earliest)	3.1.4 (latest)	1.7.4 (earliest)	1.6.2 (latest)	4.7.2	1.3.0	4.0.1	2.5	2.9.2	4.1.1		
5.2.1	2.3 (earliest)	3.1.7 (latest)	3.3.3 (earliest)	3.1.4 (latest)	1.7.4 (earliest)	1.6.2 (latest)	4.7.2	4.7.1	1.3.0	3.0.0	2.5	2.8.1	3.25.1	
5.2.0	2.3 (earliest)	3.1.6 (latest)	3.3.3 (earliest)	3.1.4 (latest)	1.7.4 (earliest)	1.6.2 (latest)	4.7.1	4.7.0	1.3.0	3.0.0	2.4	2.7.2	3.24.0	

A bold version number indicates that the component version was updated in latest release.

5.2.8

Fixes

- Only load `datasource.pgbackrest.configuration` when performing a cloud based restore.
- Queue an event based on instance Patroni 'master' role change

Changes

- PostgreSQL versions 14.11, 13.14, and 12.18 are now available.
- pgBackRest is now at version 2.49.
- patroni is now at version 3.1.2.
- The orafce extension is now at version 4.9.1.
- The pg_cron extension is now at version 1.6.2.
- The pg_partman extension is now at version 5.0.1 for PG 16, 15 and 14.
- The pgvector extension is now at version 0.6.0.
- The TimescaleDB extension is now available for PG 16. The extension is at version 2.13.0 for PG 16, 15, 14, and 13.

5.2.7

Changes

- PostgreSQL versions 14.10, 13.13, 12.17, and 11.22 are now available.
- pgBouncer is now at version 1.21.0.
- The orafce extension is now at version 4.7.0.
- The pg_partman extension is now at version 5.0.0 for PG 14.
- The pgvector extension is now at version 0.5.1.
- The TimescaleDB extension now at version 2.12.2 for PG 14 and 13, version 2.11.2 for PG 12 and version 2.3.1 for PG 11.

5.2.6

Changes

- Patroni is now at version 3.1.1.
- PostGis version 3.3.4 is now available.
- The orafce extension is now at version 4.6.1.
- The pg_cron extension is now at version 1.6.0.
- The pg_partman extension is now at version 4.7.4.
- The pgAudit Analyze extension is now at version 1.0.9.
- The pgnodemx extension is now at version 1.6.
- The pgRouting extension is now at version 3.3.4 for PG 15 & 14.
- pscycopg is now at version 2.9.7.
- The TimescaleDB extension is now at version 2.11.2.

5.2.5

Changes

- PostgreSQL versions 14.9, 13.12, 12.16, and 11.21 are now available.
- Patroni is now at version 3.1.0.
- pgBackrest is now at version 2.47.
- pgBouncer is now at version 1.19.1.

Fixes

- PostgresClusters that do not request huge pages can now be restored on nodes with huge pages.

5.2.4

Changes

- The `pgaudit_analyze` tool is deprecated and may be removed in a future release.

Fixes

- Backup jobs for S3-compatible object storage repositories would fail with a message about config hash mismatch. This is now fixed.

5.2.3

Fixes

- PostgresClusters that do not request huge pages can now initialize on nodes with huge pages. Kubernetes container runtimes still configure cgroups incorrectly in these cases, but `initdb` no longer crashes.

5.2.2

This release contains new component and Postgres versions, but no additional fixes or changes.

5.2.1

Fixes

- With the exception of the `--repo` option itself, PGO no longer prevents users from specifying pgBackRest options containing the string “repo” (e.g. `--repo1-retention-full`).
- PGO now properly filters Jobs by namespace when reconciling restore or data migrations Job, ensuring PostgresClusters with the same name can be created within different namespaces.

5.2.0

Major Features

This and all PGO v5 releases are compatible with a brand new **pggo** command line interface. Please see the [pggo CLI documentation](#) for its release notes and more details.

Features

- Added the ability to customize and influence the scheduling of pgBackRest backup Jobs using **affinity** and **tolerations**.
- You can now pause the reconciliation and rollout of changes to a PostgreSQL cluster using the **spec.paused** field.
- Leaf certificates provisioned by PGO as part of a PostgreSQL cluster's TLS infrastructure are now automatically rotated prior to expiration.
- PGO now has support for feature gates.
- You can now add custom sidecars to both PostgreSQL instance Pods and PgBouncer Pods using the **spec.instances.containers** and **spec.proxy.pgBouncer.containers** fields.
- It is now possible to configure standby clusters to replicate from a remote primary using streaming replication.
- Added the ability to provide a custom **nodePort** for the primary PostgreSQL, pgBouncer and pgAdmin services.
- Added the ability to define custom labels and annotations for the primary PostgreSQL, pgBouncer and pgAdmin services.

Changes

- All containers are now run with the minimum capabilities required by the container runtime.
- The PGO documentation now includes instructions for rotating the root TLS certificate.
- A **fsGroupChangePolicy** of **OnRootMismatch** is now set on all Pods.
- The **runAsNonRoot** security setting is on every container rather than every pod.

Fixes

- A better timeout has been set for the `pg_ctl start` and `stop` commands that are run during a restore.
- A restore can now be re-attempted if PGO is unable to cleanly start or stop the database during a previous restore attempt.

Crunchy Postgres for Kubernetes 5.1.x Release Notes

Release notes for each of the 5.1.x releases.

Component versions

Crunchy Postgres for Kubernetes		Postgres	pgBackRest	pgbouncer	Patroni	pgadmin
5.1.8	14.9	2.47	1.19	3.1.0		4.30
5.1.7	14.8	2.45	1.19	2.1.7		4.30
5.1.6	14.8	2.45	1.19	2.1.7		4.30
5.1.5	14.7	2.41	1.18	2.1.7		4.30
5.1.4	14.6	2.41	1.17	2.1.4		4.30
5.1.3	14.5	2.40	1.17	2.1.4		4.30
5.1.2	14.4	2.38	1.15	2.1.3		4.30
5.1.1	14.3	2.38	1.15	2.1.3		4.30
5.1.0	14.3	2.38	1.15	2.1.3		4.30

Postgres extension versions

Crunchy Postgres for Kubernetes	PostGIS	PostGIS-GeoJSON	PostGIS-Routing	pgaudit	pg_cron	pg_partman	pg_repack	pg_stat_statements	pg_temp	pg_trgm	pg_visibility	set_config	user_defined	json2json	TimescaleDB	DBAL	pgvector
5.1.8	3.0 (earliest)	3.3.2 (latest)	3.3.3 (earliest)	3.3.1 (latest)	1.5.4 (earliest)	1.7.0 (latest)	1.5.2	4.7.3	1.4	4.0.1	2.5	2.10.3	4.2.6	0.4.4			
5.1.7	3.0 (earliest)	3.3.2 (latest)	3.3.3 (earliest)	3.3.1 (latest)	1.5.4 (earliest)	1.7.0 (latest)	1.5.2	4.7.3	1.4	4.0.1	2.5	2.10.3	4.2.6	0.4.4			
5.1.6	2.3 (earliest)	3.3.2 (latest)	3.3.3 (earliest)	3.3.1 (latest)	1.5.4 (earliest)	1.7.0 (latest)	1.5.2	4.7.3	1.4	4.0.1	2.5	2.10.3	4.2.6				
5.1.5	2.3 (earliest)	3.1.8 (latest)	3.3.3 (earliest)	3.1.4 (latest)	1.5.4 (earliest)	1.6.2 (latest)	1.5.2	4.7.2	1.3.0	4.0.1	2.5	2.9.2	4.1.1				
5.1.4	2.3 (earliest)	3.1.7 (latest)	3.3.3 (earliest)	3.1.4 (latest)	1.5.4 (earliest)	1.6.2 (latest)	1.5.2	4.7.1	1.3.0	3.0.0	2.5	2.8.1	3.25.1				
5.1.3	2.3 (earliest)	3.1.6 (latest)	3.3.3 (earliest)	3.1.4 (latest)	1.5.4 (earliest)	1.6.2 (latest)	1.5.1	4.6.2	1.3.0	3.0.0	2.4	2.7.2	3.24.0				
5.1.2	2.3 (earliest)	3.1.5 (latest)	3.3.3 (earliest)	3.1.4 (latest)	1.5.4 (earliest)	1.6.2 (latest)	1.5.1	4.6.1	1.3.0	3.0.0	2.4	2.6.1					
5.1.1	2.3 (earliest)	3.1.5 (latest)	3.3.3 (earliest)	3.1.4 (latest)	1.5.4 (earliest)	1.6.2 (latest)	1.5.1	4.6.1	1.3.0	3.0.0	2.4	2.6.1					
5.1.0	2.3 (earliest)	3.1.4 (latest)	3.3.3 (earliest)	3.1.4 (latest)	1.5.4 (earliest)	1.6.2 (latest)	1.5.1	4.6.0	1.3.0	3.0.0	2.4	2.6.0					

A bold version number indicates that the component version was updated in latest release.

5.1.8

Changes

- PostgreSQL versions 14.9, 13.12, 12.16, and 11.21 are now available.

- Patroni is now at version 3.1.0.
- pgBackrest is now at version 2.47.
- pgBouncer is now at version 1.19.1.

Fixes

- PostgresClusters that do not request huge pages can now be restored on nodes with huge pages.

5.1.7

Changes

- The `pgaudit_analyze` tool is deprecated and may be removed in a future release.

Fixes

- Backup jobs for S3-compatible object storage repositories would fail with a message about config hash mismatch. This is now fixed.

5.1.6

Fixes

- PostgresClusters that do not request huge pages can now initialize on nodes with huge pages. Kubernetes container runtimes still configure cgroups incorrectly in these cases, but `initdb` no longer crashes.

5.1.5

This release contains new component and Postgres versions, but no additional fixes or changes.

5.1.4

Fixes

- With the exception of the `--repo` option itself, PGO no longer prevents users from specifying pgBackRest options containing the string “repo” (e.g. `--repo1-retention-full`).
- PGO now properly filters Jobs by namespace when reconciling restore or data migrations Job, ensuring PostgresClusters with the same name can be created within different namespaces.

5.1.3

Fixes

- A better timeout has been set for the `pg_ctl start` and `stop` commands that are run during a restore.

- A restore can now be re-attempted if PGO is unable to cleanly start or stop the database during a previous restore attempt.

5.1.2

This release contains new component and Postgres versions, but no additional fixes or changes.

5.1.1

Fixes

- It is now possible to perform major PostgreSQL version upgrades when using an external WAL directory.
- The documentation for pgAdmin 4 now clearly states that any pgAdmin user created by PGO will have a `@pgo` suffix.

5.1.0

Major Features

pgAdmin 4 Integration

PGO v5.1 reintroduces the pgAdmin 4 integration from [PGO v4](#). v5.1 adds the `spec.userInterface.pgAdmin` section to the `PostgresCluster` custom resource to enable pgAdmin 4 integration for a Postgres cluster. Any users defined in `spec.users` are synced with pgAdmin 4, allowing for a seamless management experience.

Please see the [pgAdmin 4 section](#) of the PGO documentation for more information about this integration.

Removal of SSH Requirement for Local Backups

Previous versions of PGO relied on the use of `ssh` to take backups and store archive files on Kubernetes-managed storage. PGO v5.1 now uses mTLS to securely transfer and manage these files.

The upgrade to pgBackRest TLS is seamless and transparent if using related image environment variables with your PGO Deployment (please see the [PostgresCluster CRD reference](#) for more information). This is because PGO will automatically handle updating all image tags across all existing PostgresCluster's following the upgrade to v5.1, seamlessly rolling out any new images as required for proper pgBackRest TLS functionality.

If you are not using related image environment variables, and are instead explicitly defining images via the `image` fields in your PostgresCluster spec, then an additional step is required in order to ensure a seamless upgrade. Specifically, all `postgrescluster.spec.image` and `postgrescluster.spec.backups.pgbackrest.image` fields must first be updated to specify images containing pgBackRest 2.38. Therefore, prior to upgrading, please update all `postgrescluster.spec.image` and `postgrescluster.spec.backups.pgbackrest.image` fields to the latest versions of the `crunchy-postgres` and `crunchy-pgbackrest` containers available per the [Components and Compatibility guide](#) (please note that the `crunchy-postgres` container should be updated to the latest version available for the major version of PostgreSQL currently being utilized within a cluster).

In the event that PGO is upgraded to v5.1 *before* updating your image tags, simply update any `image` fields in your PostgresCluster spec as soon as possible following the upgrade.

Features

- Set Pod Disruption Budgets (PDBs) for both Postgres and PgBouncer instances.
- Postgres configuration changes requiring a database restart are now automatically rolled out to all instances in the cluster.
- Do not recreate instance Pods for changes that only require a Postgres restart. These types of changes are now applied more quickly.
- Support for manual switchovers or failovers.
- Rotate PgBouncer TLS certificates without downtime.
- Add support for using Active Directory for securely authenticating with PostgreSQL using the GSSAPI.
- Support for using AWS IAM roles with S3 with backups when PGO is deployed in EKS.
- The characters used for password generation can now be controlled using the `postgrescluster.spec.users.password.type` parameter. Choices are `AlphaNumeric` and `ASCII`; defaults to `ASCII`.
- Introduction for automatically checking for updates for PGO and Postgres components. If an update is discovered, it is included in the PGO logs.

Changes

- As a result of [a fix in PgBouncer v1.16](#), PGO no longer sets verbosity settings in the PgBouncer configuration to catch missing `%include` directives. Users can increase verbosity in their own configuration files to maintain the previous behavior.
- The Postgres `archive_timeout` setting now defaults to 60 seconds (`60s`), which matches the behavior from PGO v4. If you do not require for WAL files to be generated once a minute (e.g. generally idle system where a window of data-loss is acceptable or a development system), you can set this to `0`:

```
spec:
  patroni:
    dynamicConfiguration:
      postgresql:
        parameters:
          archive_timeout: 0
```

- All Pods now have `enableServiceLinks` set to `false` in order to ensure injected environment variables do not conflict with the various applications running within.

Fixes

- The names of CronJobs created for scheduled backups are shortened to `<cluster name>-<repo#>-<backup type>` to allow for longer PostgresCluster names.

Crunchy Postgres for Kubernetes 5.0.x Release Notes

Release notes for each of the 5.0.x releases.

Component versions

Crunchy Postgres for Kubernetes		Postgres	pgBackRest	pgbouncer	Patroni	pgadmin
5.0.9	14.6	2.41	1.17	2.1.4		n/a
5.0.8	14.5	2.40	1.17	2.1.4		n/a
5.0.7	14.4	2.38	1.16	2.1.3		n/a
5.0.6	14.3	2.38	1.16	2.1.2		n/a
5.0.5	14.2	2.36	1.16	2.1.2		n/a
5.0.4	14.1	2.36	1.16	2.1.2		n/a
5.0.3	14.0	2.35	1.15	2.1.1		n/a
5.0.2	13.4	2.35	1.15	2.1.0		n/a
5.0.1	13.3	2.35	1.15	2.1.0		n/a
5.0.0	13.3	2.35	1.15	2.0.2		n/a

Postgres extension versions

Crunchy Postgres for Kubernetes	PostGIS	SpGist	Routing	pgaudit	pg_cron	pg_partman	pgnodemxset	userwal2	jsonb	TimescaleDB	Draftee		
5.0.9	2.3 (earliest)	3.2 (latest)	2.6.3 (earliest)	3.1.4 (latest)	1.2.4 (earliest)	1.6.2 (latest)	4.1	4.7.1	1.3.0	3.0.0	2.4	2.8.1	3.25.1
5.0.8	2.3 (earliest)	3.2 (latest)	2.6.3 (earliest)	3.1.4 (latest)	1.2.4 (earliest)	1.6.2 (latest)	4.1	4.6.2	1.3.0	3.0.0	2.4	2.7.2	3.22.0
5.0.7	2.3 (earliest)	3.2 (latest)	2.6.3 (earliest)	3.1.4 (latest)	1.2.4 (earliest)	1.6.2 (latest)	4.1	4.6.1	1.3.0	3.0.0	2.4	2.6.1	n/a
5.0.6	2.3 (earliest)	3.2.1 (latest)	2.6.3 (earliest)	3.1.4 (latest)	1.2.4 (earliest)	1.6.2 (latest)	4.1	4.6.1	1.3.0	3.0.0	2.4	2.6.1	n/a
5.0.5	2.3 (earliest)	3.1 (latest)	2.6.3 (earliest)	3.1.4 (latest)	1.2.2 (earliest)	1.6.2 (latest)	4.1	4.6.0	1.2.0	3.0.0	2.4	2.5.0	n/a
5.0.4	2.3 (earliest)	3.1 (latest)	2.6.3 (earliest)	3.1.4 (latest)	1.2.2 (earliest)	1.6.1 (latest)	3.1	4.6.0	1.2.0	3.0.0	2.4	2.5.0	n/a
5.0.3	2.3 (earliest)	3.1 (latest)	2.6.3 (earliest)	3.1.3 (latest)	1.2.2 (earliest)	1.6.0 (latest)	3.1	4.5.1	1.0.5	3.0.0	2.4	2.4.2	n/a
5.0.2	2.3 (earliest)	3.1 (latest)	2.6.3 (earliest)	3.1.3 (latest)	1.2.2 (earliest)	1.5.0 (latest)	3.1	4.5.1	1.0.4	2.0.1	2.3	2.4.0	n/a
5.0.1	2.3 (earliest)	3.1 (latest)	2.6.3 (earliest)	3.1.3 (latest)	1.2.2 (earliest)	1.5.0 (latest)	3.1	4.5.1	1.0.4	2.0.0	2.3	2.3.1	n/a
5.0.0	2.3 (earliest)	3.1 (latest)	2.6.3 (earliest)	3.1.3 (latest)	1.2.2 (earliest)	1.5.0 (latest)	3.1	4.5.1	1.0.4	2.0.0	2.3	2.2.0	n/a

A bold version number indicates that the component version was updated in latest release.

5.0.9

Fixes

- With the exception of the `--repo` option itself, PGO no longer prevents users from specifying pgBackRest options containing the string “repo” (e.g. `--repo1-retention-full`).

- PGO now properly filters Jobs by namespace when reconciling restore or data migrations Job, ensuring PostgresClusters with the same name can be created within different namespaces.

5.0.8

Fixes

- A better timeout has been set for the `pg_ctl` start and stop commands that are run during a restore.
- A restore can now be re-attempted if PGO is unable to cleanly start or stop the database during a previous restore attempt.

5.0.7

This release contains new component and Postgres versions, but no additional fixes or changes.

5.0.6

This release contains new component and Postgres versions, but no additional fixes or changes.

5.0.5

Features

- A S3, GCS or Azure data source can now be configured when bootstrapping a new PostgresCluster. This allows existing cloud-based pgBackRest repositories to be utilized to bootstrap new clusters, while also ensuring those new clusters create and utilize their own pgBackRest repository for archives and backups (rather than writing to the repo utilized to bootstrap the cluster).
- It is now possible to configure the number of workers for the PostgresCluster controller.

Fixes

- Reduce scope of automatic OpenShift environment detection. This looks specifically for the existence of the `SecurityContextConstraint` API.
- An external IP is no longer copied to the primary service (e.g. `hippo-primary`) when the `LoadBalancer` service type has been configured for PostgreSQL.
- pgBackRest no longer logs to `log /tmp` emptyDir by default. Instead, pgBackRest logs to either the `PGDATA` volume (if running inside of a PG instance Pod) or a pgBackRest repository volume (if running inside a dedicated repo host Pod).
- All pgBackRest configuration resources are now copied from the source cluster when cloning a PG cluster.
- Image pull secrets are now set on directory move jobs.
- Resources are now properly set on the `nss-wrapper-init` container.

5.0.4

Features

- The JDBC connection string for the Postgres database and a PgBouncer instance is now available in the User Secret using `jdbc-uri` and `pgbouncer-jdbc-uri` respectively.
- Editing the `password` field of a User Secret now [changes a password](#), instead of having to create a verifier.

Changes

- [PostGIS](#) is now automatically enabled when using the `crunchy-postgres-gis` container.
- The [Downward API](#) is mounted to the `database` containers.
- [pgnodemx](#) can now be enabled and used without having to enable monitoring.
- The description of the `name` field for an instance set now states that a name is only optional when a single instance set is defined.

Fixes

- Fix issue when performing a restore with PostgreSQL 14. Specifically, if there are mismatched PostgreSQL configuration parameters, PGO will resume replay and let PostgreSQL crash so PGO can ultimately fix it, vs. the restore pausing indefinitely.
- The pgBackRest Pod no longer automatically mounts the default Service Account. Reported by (@Shrivastava-Varsha).
- The Jobs that move data between volumes now have the correct Security Context set.
- The UBI 8 `crunchy-upgrade` container contains all recent PostgreSQL versions that can be upgraded.
- Ensure controller references are used for all objects that need them, instead of owner references.
- It is no longer necessary to have external WAL volumes enabled in order to upgrade a PGO v4 cluster to PGO v5 using the "Migrate From Backups" or "Migrate Using a Standby Cluster" upgrade methods.

5.0.3

Features

- The Postgres containers are renamed. `crunchy-postgres-ha` is now `crunchy-postgres`, and `crunchy-postgres-gis-ha` is now `crunchy-postgres-gis`.
- Some network filesystems are sensitive to Linux user and group permissions. Process GIDs can now be configured through `PostgresCluster.spec.supplementalGroups` for when your PVs don't advertise their [GID requirements](#).
- A replica service is now automatically reconciled for access to Postgres replicas within a cluster.
- The Postgres primary service and PgBouncer service can now each be configured to have either a `ClusterIP`, `NodePort` or `LoadBalancer` service type. Suggested by Bryan A. S. (@bryanasdev000).
- [Pod Topology Spread Constraints](#) can now be specified for Postgres instances, the pgBackRest dedicated repository host as well as PgBouncer. Suggested by Annette Clewett.
- Default topology spread constraints are included to ensure PGO always attempts to deploy a high availability cluster architecture.

- PGO can now execute a custom SQL script when initializing a Postgres cluster.
- Custom resource requests and limits are now configurable for all **init** containers, therefore ensuring the desired [Quality of Service \(QoS\)](#) class can be assigned to the various Pods comprising a cluster.
- Custom resource requests and limits are now configurable for all Jobs created for a **PostgresCluster**.
- A [Pod Priority Class](#) is configurable for the Pods created for a **PostgresCluster**.
- An **imagePullPolicy** can now be configured for Pods created for a **PostgresCluster**.
- Existing **PGDATA**, Write-Ahead Log (WAL) and pgBackRest repository volumes can now be migrated from PGO v4 to PGO v5 by specifying a **volumes** data source when creating a **PostgresCluster**.
- There is now a migration guide available for moving Postgres clusters between PGO v4 to PGO v5.
- The pgAudit extension is now enabled by default in all clusters.
- There is now additional validation for PVC definitions within the **PostgresCluster** spec to ensure successful PVC reconciliation.
- Postgres server certificates are now automatically reloaded when they change.

Changes

- The supplemental group **65534** is no longer applied by default. Upgrading the operator will perform a rolling update on all **PostgresCluster** custom resources to remove it.

If you need this GID for your network filesystem, you should perform the following steps when upgrading:

- Before deploying the new operator, deploy the new CRD. You can get the new CRD from the [Postgres Operator Examples](#) repository and executing the following command:

```
kubectl apply -k kustomize/install
```

- Add the group to your existing **PostgresCluster** custom resource:

```
kubectl edit postgrescluster/hippo

kind: PostgresCluster ... spec: supplementalGroups: - 65534 ...
```

or

```
kubectl patch postgrescluster/hippo --type=merge --patch='{"spec":{"supplemental-Groups":[65534]}}'
```

or

by modifying **spec.supplementalGroups** in your manifest.

- Deploy the new operator. If you are using an up-to-date version of the manifest, you can run:

```
kubectl apply -k kustomize/install
```

- A dedicated pgBackRest repository host is now only deployed if a **volume** repository is configured. This means that deployments that use only cloud-based (**s3**, **gcs**, **azure**) repos will no longer see a dedicated repository host, nor will **SSH** run in within that Postgres cluster. As a result of this change, the **spec.backups.pgbackrest.re-**

`poHost.dedicated` section is removed from the `PostgresCluster` spec, and all settings within it are consolidated under the `spec.backups.pgbackrest.repoHost` section. When upgrading please update the `PostgresCluster` spec to ensure any settings from section `spec.backups.pgbackrest.repoHost.dedicated` are moved into section `spec.backups.pgbackrest.repoHost`.

- PgBouncer now uses SCRAM when authenticating into Postgres.
- Generated Postgres certificates include the FQDN and other local names of the primary Postgres service. To regenerate the certificate of an existing cluster, delete the `tls.key` field from its certificate secret. Suggested by @ackerr01.

Fixes

- Validation for the `PostgresCluster` spec is updated to ensure at least one repo is always defined for section `spec.backups.pgbackrest.repos`.
- A restore will now complete successfully if `max_connections` and/or `max_worker_processes` is configured to a value higher than the default when backing up the Postgres database. Reported by Tiberiu Patrascu (@tpatrascu).
- The installation documentation now properly defines how to set the `PGO_TARGET_NAMESPACE` environment variable for a single namespace installation.
- Ensure the full allocation of shared memory is available to Postgres containers. Reported by Yuyang Zhang (@helloqiu).
- OpenShift auto-detection logic now looks for the presence of the `SecurityContextConstraints` API to avoid false positives when APIs with an `openshift.io` Group suffix are installed in non-OpenShift clusters. Reported by Jean-Daniel.

5.0.2

This release contains new component and Postgres versions, but no additional fixes or changes.

5.0.1

Features

- Custom affinity rules and tolerations can now be added to pgBackRest restore Jobs.
- OLM bundles can now be generated for PGO 5.

Changes

- The `replicas` value for an instance set must now be greater than `0`, and at least one instance set must now be defined for a `PostgresCluster`. This is to prevent the cluster from being scaled down to `0` instances, since doing so results in the inability to scale the cluster back up.
- Refreshed the `PostgresCluster` CRD documentation using the latest version of `crdoc` (`v0.3.0`).
- The PGO test suite now includes a test to validate image pull secrets.
- Related Image functionality has been implemented for the OLM installer as required to support offline deployments.
- The name of the PGO Deployment and ServiceAccount has been changed to `pgo` for all installers, allowing both PGO v4.x and PGO v5.x to be run in the same namespace. If you are using Kustomize to install PGO and are upgrading from

PGO 5.0.0, please see the Upgrade Guide for additional steps that must be completed as a result of this change in order to ensure a successful upgrade.

- PGO now automatically detects whether or not it is running in an OpenShift environment.
- Postgres users and databases can be specified in `PostgresCluster.spec.users`. The credentials stored in the `{cluster}-pguser` Secret are still valid, but they are no longer reconciled. References to that Secret should be replaced with `{cluster}-pguser-{cluster}`. Once all references are updated, the old `{cluster}-pguser` Secret can be deleted.
- The built-in `postgres` superuser can now be managed the same way as other users. Specifying it in `PostgresCluster.spec.users` will give it a password, allowing it to connect over the network.
- PostgreSQL data and pgBackRest repo volumes are now reconciled using labels.

Fixes

- It is now possible to customize `shared_preload_libraries` when monitoring is enabled.
- Fixed a typo in the description of the `openshift` field in the PostgresCluster CRD.
- When a new cluster is created using an existing PostgresCluster as its dataSource, the original primary for that cluster will now properly initialize as a replica following a switchover. This is fixed with the upgrade to Patroni 2.1.0).
- A consistent `startupInstance` name is now set in the PostgresCluster status when bootstrapping a new cluster using an existing PostgresCluster as its data source.
- It is now possible to properly customize the `pg_hba.conf` configuration file.

5.0.0

Changes

Beyond being fully declarative, PGO 5.0 has some notable changes that you should be aware of. These include:

- The minimum Kubernetes version is now 1.18. The minimum OpenShift version is 4.5. This release drops support for OpenShift 3.11.
- We recommend running the latest bug fix releases of Kubernetes.
- The removal of the `pgo` client. This may be reintroduced in a later release, but all actions on a Postgres cluster can be accomplished using `kubectl`, `oc`, or your preferred Kubernetes management tool (e.g. ArgoCD).
- A fully defined `status` sub-resource is now available within the `postgrescluster` custom resource that provides direct insight into the current status of a PostgreSQL cluster.
- Native Kubernetes eventing is now utilized to generate and record events related to the creation and management of PostgreSQL clusters.
- Postgres instances now use Kubernetes Statefulsets.
- Scheduled backups now use Kubernetes CronJobs.
- Connections to Postgres require TLS. You can bring your own TLS infrastructure, otherwise PGO provides it for you.
- Custom configurations for all components can be set directly on the `postgrescluster` custom resource.

Features

In addition to supporting the PGO 4.x feature set, the PGO 5.0.0 adds the following new features:

- Postgres minor version (bug fix) updates can be applied without having to update PGO. You only need to update the **image** attribute in the custom resource.
- Adds support for Azure Blob Storage for storing backups. This is in addition to using Kubernetes storage, Amazon S3 (or S3-equivalents like MinIO), and Google Cloud Storage (GCS).
- Allows for backups to be stored in up to four different locations simultaneously.
- Backup locations can be changed during the lifetime of a Postgres cluster, e.g. moving from "posix" to "s3".

References

CRD Reference

You can view the CRD Reference for your currently installed version using the links below:

CRD Versions:

- 5.8.x
- 5.7.x
- 5.6.x
- 5.5.x
- 5.4.x
- 5.3.x
- 5.2.x
- 5.1.x
- 5.0.x

5.8.x

Packages:

- postgres-operator.crunchydata.com/v1beta1

postgres-operator.crunchydata.com/v1beta1

Resource Types:

- CrunchyBridgeCluster
- PGUpgrade
- PostgresCluster
- PGAdmin

5.7.x

Packages:

- postgres-operator.crunchydata.com/v1beta1

postgres-operator.crunchydata.com/v1beta1

Resource Types:

- CrunchyBridgeCluster
- PGUpgrade
- PostgresCluster
- PGAdmin

5.6.x

Packages:

- postgres-operator.crunchydata.com/v1beta1

postgres-operator.crunchydata.com/v1beta1

Resource Types:

- CrunchyBridgeCluster
- PGUpgrade
- PostgresCluster
- PGAdmin

5.5.x

Packages:

- postgres-operator.crunchydata.com/v1beta1

postgres-operator.crunchydata.com/v1beta1

Resource Types:

- PGUpgrade
- PostgresCluster
- PGAdmin

5.4.x

Packages:

- postgres-operator.crunchydata.com/v1beta1

postgres-operator.crunchydata.com/v1beta1

Resource Types:

- PGUpgrade
- PostgresCluster

5.3.x

Packages:

- postgres-operator.crunchydata.com/v1beta1

postgres-operator.crunchydata.com/v1beta1

Resource Types:

- PGUpgrade
- PostgresCluster

5.2.x

Packages:

- postgres-operator.crunchydata.com/v1beta1

postgres-operator.crunchydata.com/v1beta1

Resource Types:

- PGUpgrade
- PostgresCluster

5.1.x

Packages:

- postgres-operator.crunchydata.com/v1beta1

postgres-operator.crunchydata.com/v1beta1

Resource Types:

- PGUpgrade

- PostgresCluster

5.0.x

Packages:

- postgres-operator.crunchydata.com/v1beta1

postgres-operator.crunchydata.com/v1beta1

Resource Types:

- PGUpgrade
- PostgresCluster

Components and Compatibility

Kubernetes Compatibility

PGO, the Postgres Operator from Crunchy Data, is tested on the following platforms:

- OpenShift
- Rancher
- Google Kubernetes Engine (GKE), including Anthos
- Amazon EKS
- Microsoft AKS
- VMware Tanzu

For additional information about supported versions of Kubernetes and OpenShift, see the Supported Platforms page.

Components Compatibility

The following table defines the compatibility between PGO and the various component containers needed to deploy PostgreSQL clusters using PGO.

The listed versions of Postgres show the latest minor release (e.g. 17.5) of each major version (e.g. 17). Older minor releases may still be compatible with PGO. We generally recommend to run the latest minor release for the [same reasons that the PostgreSQL community provides](#).

Note that for the 5.0.3 release and beyond, the Postgres containers were renamed to **crunchy-postgres** and **crunchy-postgres-gis**.

Architectures

Crunchy Postgres for Kubernetes is compatible with AMD and ARM architectures. Both AMD and ARM container builds are available for the various components discussed below.

ARM support was in added in Crunchy Postgres for Kubernetes 5.4 for Postgres version 13 and greater.

Base Images

Starting with v5.8.0, Crunchy Postgres for Kubernetes images are built on UBI 9 base images from Red Hat. Images built on UBI 8 are available for earlier versions of Crunchy Postgres for Kubernetes.

UBI 9 is distributed with Linux kernel version 5.14.0, which provides [support for the following architectures](#) at the minimum required version:

- AMD and Intel 64-bit architectures (x86-64-v2)
- The 64-bit ARM architecture (ARMv8.0-A)

Container Versions

In accordance with the [Crunchy Developer Program](#), the latest two major versions of Postgres are published to the Crunchy Data Developer registry and the Red Hat certified image catalog.

Also, please note that the PostgresCluster API-based pgAdmin solution currently utilizes pgAdmin 4.30, which does not support versions of Postgres greater than 14 or ARM architectures. The PGAdmin API-based solution (which uses a more recent version of pgAdmin, as shown below) should be utilized instead for full compatibility with ARM and all actively maintained versions of Postgres.

PGO	pgAdmin	pgBackRest	pgBouncer	Postgres	PostGIS
5.8.2	9.2	2.54.2	1.24	17,16,15,14,13	3.5,3.4,3.3,3.2,3.1
5.8.1	9.2	2.54.2	1.24	17,16,15,14,13	3.5,3.4,3.3,3.2,3.1
5.8.0	9.1	2.54.2	1.24	17,16,15,14,13	3.5,3.4,3.3,3.2,3.1
5.7.6	4.30,9.2	2.54.2	1.24	17,16,15,14,13	3.5,3.4,3.3,3.2,3.1,3.0
5.7.5	4.30,9.2	2.54.2	1.24	17,16,15,14,13	3.5,3.4,3.3,3.2,3.1,3.0
5.7.4	4.30,8.14	2.54.1	1.23	17,16,15,14,13	3.5,3.4,3.3,3.2,3.1,3.0
5.7.3	4.30,8.14	2.54.1	1.23	17,16,15,14,13	3.4,3.3,3.2,3.1,3.0
5.7.2	4.30,8.14	2.54.0	1.23	17,16,15,14,13	3.4,3.3,3.2,3.1,3.0
5.7.1	4.30,8.12	2.53.1	1.23	17,16,15,14,13,12	3.4,3.3,3.2,3.1,3.0,2.5
5.7.0	4.30,8.12	2.53.1	1.23	17,16,15,14,13,12	3.4,3.3,3.2,3.1,3.0,2.5
5.6.8	4.30,9.2	2.54.2	1.24	16,15,14,13	3.4,3.3,3.2,3.1,3.0
5.6.7	4.30,9.2	2.54.2	1.24	16,15,14,13	3.4,3.3,3.2,3.1,3.0

5.6.6	4.30,8.14	2.54.1	1.23	16,15,14,13	3.4,3.3,3.2,3.1,3.0
5.6.5	4.30,8.14	2.54.1	1.23	16,15,14,13	3.4,3.3,3.2,3.1,3.0
5.6.4	4.30,8.14	2.54.0	1.23	16,15,14,13	3.4,3.3,3.2,3.1,3.0
5.6.3	4.30,8.12	2.53.1	1.23	16,15,14,13,12	3.4,3.3,3.2,3.1,3.0,2.5
5.6.2	4.30,8.12	2.53.1	1.23	16,15,14,13,12	3.4,3.3,3.2,3.1,3.0,2.5
5.6.1	4.30,8.12	2.52.1	1.22	16,15,14,13,12	3.4,3.3,3.2,3.1,3.0,2.5
5.6.0	4.30,8.6	2.51	1.22	16,15,14,13,12	3.4,3.3,3.2,3.1,3.0,2.5
5.5.10	4.30,9.2	2.54.2	1.24	16,15,14,13	3.4,3.3,3.2,3.1,3.0
5.5.9	4.30,9.2	2.54.2	1.24	16,15,14,13	3.4,3.3,3.2,3.1,3.0
5.5.8	4.30,8.14	2.54.1	1.23	16,15,14,13	3.4,3.3,3.2,3.1,3.0
5.5.7	4.30,8.14	2.54.1	1.23	16,15,14,13	3.4,3.3,3.2,3.1,3.0
5.5.6	4.30,8.14	2.54.0	1.23	16,15,14,13	3.4,3.3,3.2,3.1,3.0
5.5.5	4.30,8.6	2.53.1	1.23	16,15,14,13,12	3.4,3.3,3.2,3.1,3.0,2.5
5.5.4	4.30,8.6	2.53.1	1.23	16,15,14,13,12	3.4,3.3,3.2,3.1,3.0,2.5
5.5.3	4.30,8.6	2.52.1	1.22	16,15,14,13,12	3.4,3.3,3.2,3.1,3.0,2.5
5.5.2	4.30,8.6	2.51	1.22	16,15,14,13,12	3.4,3.3,3.2,3.1,3.0,2.5
5.5.1	4.30,7.8	2.49	1.21	16,15,14,13,12	3.4,3.3,3.2,3.1,3.0,2.5
5.5.0	4.30,7.8	2.47	1.21	16,15,14,13,12,11	3.4,3.3,3.2,3.1,3.0,2.5,2.4
5.4.9	4.30	2.53.1	1.23	16,15,14,13,12	3.4,3.3,3.2,3.1,3.0,2.5
5.4.8	4.30	2.53.1	1.23	16,15,14,13,12	3.4,3.3,3.2,3.1,3.0,2.5
5.4.7	4.30	2.52.1	1.22	16,15,14,13,12	3.4,3.3,3.2,3.1,3.0,2.5
5.4.6	4.30	2.51	1.22	16,15,14,13,12	3.4,3.3,3.2,3.1,3.0,2.5
5.4.5	4.30	2.49	1.21	16,15,14,13,12	3.4,3.3,3.2,3.1,3.0,2.5
5.4.4	4.30	2.47	1.21	16,15,14,13,12,11	3.4,3.3,3.2,3.1,3.0,2.5,2.4
5.4.3	4.30	2.47	1.19	16,15,14,13,12,11	3.4,3.3,3.2,3.1,3.0,2.5,2.4
5.4.2	4.30	2.47	1.19	15,14,13,12,11	3.3,3.2,3.1,3.0,2.5,2.4
5.4.1	4.30	2.45	1.19	15,14,13,12,11	3.3,3.2,3.1,3.0,2.5,2.4
5.4.0	4.30	2.45	1.19	15,14,13,12,11	3.3,3.2,3.1,3.0,2.5,2.4
5.3.9	4.30	2.52.1	1.22	15,14,13,12	3.3,3.2,3.1,3.0,2.5
5.3.8	4.30	2.51	1.22	15,14,13,12	3.3,3.2,3.1,3.0,2.5
5.3.7	4.30	2.49	1.21	15,14,13,12	3.3,3.2,3.1,3.0,2.5
5.3.6	4.30	2.47	1.19	15,14,13,12,11	3.3,3.2,3.1,3.0,2.5,2.4
5.3.5	4.30	2.47	1.19	15,14,13,12,11	3.3,3.2,3.1,3.0,2.5,2.4
5.3.4	4.30	2.47	1.19	15,14,13,12,11	3.3,3.2,3.1,3.0,2.5,2.4
5.3.3	4.30	2.45	1.19	15,14,13,12,11	3.3,3.2,3.1,3.0,2.5,2.4
5.3.2	4.30	2.45	1.19	15,14,13,12,11	3.3,3.2,3.1,3.0,2.5,2.4
5.3.1	4.30	2.41	1.18	15,14,13,12,11	3.3,3.2,3.1,3.0,2.5,2.4
5.3.0	4.30	2.41	1.17	15,14,13,12,11	3.3,3.2,3.1,3.0,2.5,2.4
5.2.5	4.30	2.47	1.19	14,13,12,11	3.2,3.1,3.0,2.5,2.4,2.4
5.2.4	4.30	2.45	1.19	14,13,12,11	3.2,3.1,3.0,2.5,2.4,2.4
5.2.3	4.30	2.45	1.19	14,13,12,11	3.2,3.1,3.0,2.5,2.4,2.4
5.2.2	4.30	2.41	1.18	14,13,12,11	3.2,3.1,3.0,2.5,2.4,2.4

5.2.1	4.30	2.41	1.17	14,13,12,11,10	3.2,3.1,3.0,2.5,2.4,2.3
5.2.0	4.30	2.40	1.17	14,13,12,11,10	3.2,3.1,3.0,2.5,2.4,2.3
5.1.8	4.30	2.47	1.19	14,13,12,11	3.2,3.1,3.0,2.5,2.4,2.4
5.1.7	4.30	2.45	1.19	14,13,12,11	3.2,3.1,3.0,2.5,2.4,2.4
5.1.6	4.30	2.45	1.19	14,13,12,11	3.2,3.1,3.0,2.5,2.4,2.4
5.1.5	4.30	2.41	1.17	14,13,12,11	3.2,3.1,3.0,2.5,2.4,2.4
5.1.4	4.30	2.41	1.17	14,13,12,11,10	3.2,3.1,3.0,2.5,2.4,2.3
5.1.3	4.30	2.40	1.17	14,13,12,11,10	3.2,3.1,3.0,2.5,2.4,2.3
5.1.2	4.30	2.38	1.16	14,13,12,11,10	3.2,3.1,3.0,2.5,2.4,2.3
5.1.1	4.30	2.38	1.16	14,13,12,11,10	3.2,3.1,3.0,2.5,2.4,2.3
5.1.0	4.30	2.38	1.16	14,13,12,11,10	3.1,3.0,2.5,2.4,2.3
5.0.9	n/a	2.41	1.17	14,13,12,11,10	3.1,3.0,2.5,2.4,2.3
5.0.8	n/a	2.40	1.17	14,13,12,11,10	3.1,3.0,2.5,2.4,2.3
5.0.7	n/a	2.38	1.16	14,13,12,11,10	3.2,3.1,3.0,2.5,2.4,2.3
5.0.6	n/a	2.38	1.16	14,13,12,11,10	3.2,3.1,3.0,2.5,2.4,2.3
5.0.5	n/a	2.36	1.16	14,13,12,11,10	3.1,3.0,2.5,2.4,2.3
5.0.4	n/a	2.36	1.16	14,13,12,11,10	3.1,3.0,2.5,2.4,2.3
5.0.3	n/a	2.35	1.15	14,13,12,11,10	3.1,3.0,2.5,2.4,2.3

The latest Postgres containers include Patroni **4.0.4**.

Container Tags

Starting with Crunchy Postgres for Kubernetes v5.8.0, v5.7.5, v5.6.7, and v5.5.9, all container image tags follow the pattern:

- **<baseImage>-<softwareVersion>-<suffix>**

Software Version Tag

In cases where an image's contents might refer to multiple software versions, we prefer to include that information in the tag. For instance, a PostGIS enabled image with Postgres 17 and PostGIS 3.5 would have the **-<softwareVersion>-** part of its name as **-17-3.5-**.

Suffix Tag

The numeric suffix indicates when the image was built and what other images it is compatible with.

Newer images have a larger numeric suffix than older images.

For example, the suffix **-2513** indicates that the image is newer than the image with the suffix **-2480**.

Tags for older builds

Image builds predating v5.8.0, v5.7.5, v5.6.7, and v5.5.9 are tagged with the following pattern:

- **<baseImage>-<softwareVersion>-<buildVersion>**

Additionally, some older images may be tagged with one of the following patterns:

- **<baseImage>-<pgoVersion>-<buildVersion>** (Customer Portal only)

- `<baseImage>-<softwareVersion>-<pgoVersion>-<buildVersion>` (Customer Portal only)

Extensions Compatibility

The following table defines the compatibility between Postgres extensions and versions of Postgres they are available in. The "Postgres version" corresponds with the major version of a Postgres container.

The table also lists the initial PGO version that the version of the extension is available in.

Need an extension that's not listed? [Contact us](#) to discuss your use case.

Extension	Version	Postgres Version	Initial PGO Version
hypopg	1.4.1	17, 16, 15, 14, 13	5.7.4, 5.6.6, 5.5.8
orafce	4.14.2	17, 16, 15, 14, 13	5.8.2, 5.7.6, 5.6.8, 5.5.10
orafce	4.14.0	17, 16, 15, 14, 13	5.7.2
orafce	4.10.3	17, 16, 15, 14, 13, 12	5.6.1
orafce	4.9.4	16, 15, 14, 13, 12	5.5.2
orafce	4.9.1	16, 15, 14, 13, 12	5.5.1
orafce	4.7.0	16, 15, 14, 13, 12, 11	5.5.0
orafce	4.6.1	16, 15, 14, 13, 12, 11	5.4.3
orafce	4.2.6	15, 14, 13, 12, 11	5.4.0
orafce	3.25.1	15, 14, 13, 12, 11	5.3.0
orafce	3.25.1	14, 13, 12, 11, 10	5.2.1
orafce	3.24.0	14, 13, 12, 11, 10	5.1.3
orafce	3.22.0	14, 13, 12, 11, 10	5.0.8
pgAudit	17.0	17	5.7.0
pgAudit	16.0	16	5.5.0
pgAudit	1.7.0	15	5.4.0, 5.3.0
pgAudit	1.6.2	14	5.1.0, 5.0.6
pgAudit	1.6.1	14	5.0.4
pgAudit	1.6.0	14	5.0.3
pgAudit	1.5.2	13	5.1.0, 5.0.6
pgAudit	1.5.0	13	5.0.0
pgAudit	1.4.3	12	5.1.0
pgAudit	1.4.1	12	5.0.0
pgAudit	1.3.4	11	5.1.0, 5.0.6
pgAudit	1.3.2	11	5.0.0
pgAudit	1.2.4	10	5.1.0, 5.0.6
pgAudit	1.2.2	10	5.0.0
pgAudit Analyze	1.0.9	17, 16, 15, 14, 13, 12	5.4.3
pgAudit Analyze	1.0.8	14, 13, 12, 11, 10	5.0.3
pgAudit Analyze	1.0.7	13, 12, 11, 10	5.0.0
pg_cron	1.6.5	17, 16, 15, 14, 13	5.7.3
pg_cron	1.6.4	17, 16, 15, 14, 13, 12	5.7.0

pg_cron	1.6.2	16, 15, 14, 13, 12	5.5.1
pg_cron	1.6.0	16, 15, 14, 13, 12, 11	5.4.3
pg_cron	1.5.2	15, 14, 13, 12, 11	5.4.0
pg_cron	1.4.2	15, 14, 13	5.3.0
pg_cron	1.4.2	14, 13	5.2.1
pg_cron	1.4.1	14, 13, 12, 11, 10	5.0.5
pg_cron	1.3.1	14, 13, 12, 11, 10	5.0.0
pg_jobmon	1.4.1	17, 16, 15, 14, 13	5.7.4
pg_jobmon	1.4.1	16, 15, 14, 13	5.6.6, 5.5.8
pg_parquet	0.4.0	17, 16, 15, 14	5.8.2, 5.7.6, 5.6.8, 5.5.10
pg_parquet	0.2.0	17, 16, 15, 14	5.7.4
pg_parquet	0.2.0	16, 15, 14	5.6.6, 5.5.8
pg_partman	5.2.4	17, 16, 15, 14	5.7.4, 5.6.6, 5.5.8
pg_partman	5.2.2	17, 16, 15, 14	5.7.3
pg_partman	5.1.0	17, 16, 15, 14	5.5.2
pg_partman	5.0.1	16, 15, 14	5.5.1
pg_partman	5.0.0	16, 15, 14	5.5.0
pg_partman	4.7.4	16, 15, 14, 13, 12, 11	5.4.3
pg_partman	4.7.3	15, 14, 13, 12, 11	5.4.0
pg_partman	4.7.1	15, 14, 13, 12, 11	5.3.0
pg_partman	4.6.2	14, 13, 12, 11, 10	5.2.0
pg_partman	4.6.2	14, 13, 12, 11, 10	5.1.3, 5.0.8
pg_partman	4.6.1	14, 13, 12, 11, 10	5.1.1, 5.0.6
pg_partman	4.6.0	14, 13, 12, 11, 10	5.0.4
pg_partman	4.5.1	13, 12, 11, 10	5.0.0
pgnodemx	1.7	17, 16, 15, 14, 13, 12	5.7.0
pgnodemx	1.6	16, 15, 14, 13, 12, 11	5.4.3
pgnodemx	1.4	15, 14, 13, 12, 11	5.4.0
pgnodemx	1.3.0	14, 13, 12, 11, 10	5.1.0, 5.0.6
pgnodemx	1.2.0	14, 13, 12, 11, 10	5.0.4
pgnodemx	1.0.5	14, 13, 12, 11, 10	5.0.3
pgnodemx	1.0.4	13, 12, 11, 10	5.0.0
pgvector	0.8.0	17, 16, 15, 14, 13	5.7.2
pgvector	0.7.4	17, 16, 15, 14, 13, 12	5.7.0
pgvector	0.7.3	16, 15, 14, 13, 12	5.6.1
pgvector	0.7.0	16, 15, 14, 13, 12	5.5.2
pgvector	0.6.0	16, 15, 14, 13, 12	5.5.1
pgvector	0.5.1	16, 15, 14, 13, 12, 11	5.5.0
pgvector	0.4.4	15, 14, 13, 12, 11	5.4.0
set_user	4.1.0	17,16, 15, 14, 13, 12	5.7.0
set_user	4.0.1	15, 14, 13, 12, 11	5.4.0
set_user	3.0.0	14, 13, 12, 11, 10	5.0.3

set_user	2.0.1	13, 12, 11, 10	5.0.2
set_user	2.0.0	13, 12, 11, 10	5.0.0
TimescaleDB	2.19.3	17, 16, 15, 14	5.8.2, 5.7.6, 5.6.8, 5.5.10
TimescaleDB	2.18.2	17, 16, 15, 14	5.8.1, 5.7.5, 5.6.7, 5.5.9
TimescaleDB	2.17.2	17, 16, 15, 14	5.7.2
TimescaleDB	2.17.0	17, 16, 15, 14	5.7.0
TimescaleDB	2.15.3	16, 15, 14	5.6.1
TimescaleDB	2.14.2	16, 15, 14, 13	5.5.2
TimescaleDB	2.13.0	16, 15, 14, 13	5.5.1
TimescaleDB	2.12.2	15, 14, 13	5.5.0
TimescaleDB	2.11.2	15, 14, 13, 12	5.4.3
TimescaleDB	2.10.3	15, 14, 13, 12	5.4.0
TimescaleDB	2.8.1	14, 13, 12	5.3.0
TimescaleDB	2.6.1	14, 13, 12	5.1.1, 5.0.6
TimescaleDB	2.6.0	14, 13, 12	5.1.0
TimescaleDB	2.5.0	14, 13, 12	5.0.3
TimescaleDB	2.4.2	13, 12	5.0.3
TimescaleDB	2.4.0	13, 12	5.0.2
TimescaleDB	2.3.1	11	5.0.1
TimescaleDB	2.2.0	13, 12, 11	5.0.0
wal2json	2.6	17, 16, 15, 14, 13, 12	5.7.0
wal2json	2.5	15, 14, 13, 12, 11	5.4.0
wal2json	2.4	14, 13, 12, 11, 10	5.0.3
wal2json	2.3	13, 12, 11, 10	5.0.0

Geospatial Extensions

The following extensions are available in the geospatially aware containers (**crunchy-postgres-gis**):

Extension	Version	Postgres Version	Initial PGO Version
PostGIS	3.5	17	5.7.4
PostGIS	3.4	17, 16	5.4.3
PostGIS	3.3	16, 15, 14	5.4.3
PostGIS	3.3	15, 14	5.3.0
PostGIS	3.2	14	5.1.1
PostGIS	3.2	14	5.0.6
PostGIS	3.1	14, 13	5.0.0
PostGIS	3.0	13, 12	5.0.0
PostGIS	2.5	12, 11	5.0.0
PostGIS	2.4	11, 10	5.0.0
PostGIS	2.3	10	5.0.0
pgrouting	3.7.1	17	5.7.4

pgrouting	3.4.2	17, 16	5.4.3
pgrouting	3.3.4	16, 15, 14	5.4.3
pgrouting	3.3.1	15, 14	5.3.0
pgrouting	3.2.2	14	5.1.1
pgrouting	3.1.4	14	5.0.4
pgrouting	3.1.3	13	5.0.0
pgrouting	3.0.6	13	5.1.0
pgrouting	3.0.5	13, 12	5.0.0
pgrouting	2.6.3	12, 11, 10	5.0.0

Support

There are a few options available for community support of [PGO](#), the Postgres Operator from Crunchy Data:

- **If you believe you have found a bug** or have a detailed feature request: please open [an issue on GitHub](#). The Postgres Operator community and the Crunchy Data team behind the PGO is generally active in responding to issues.
- **For general questions or community support**, we welcome you to join our [community Discord](#) and ask your questions there.

In all cases, please be sure to provide as many details as possible in regards to your issue, including:

- Your Platform (e.g. Kubernetes vX.YY.Z)
- Operator Version (e.g. 5.8.2)
- A detailed description of the issue, as well as steps you took that lead up to the issue
- Any relevant logs
- Any additional information you can provide that you may find helpful

For production and commercial support of the PostgreSQL Operator, please [contact Crunchy Data](#) at info@crunchydata.com for information regarding an [Enterprise Support Subscription](#).